

1. TeamCity Documentation . . . . .	5
1.1 What's New in TeamCity 2018.1 . . . . .	6
1.2 Getting Started with TeamCity . . . . .	11
1.2.1 Continuous Integration with TeamCity . . . . .	12
1.2.2 Installation Quick Start . . . . .	14
1.2.3 Configure and Run Your First Build . . . . .	17
1.3 Concepts . . . . .	27
1.3.1 Build Artifact . . . . .	28
1.3.2 Build Configuration . . . . .	29
1.3.3 Composite Build Configuration . . . . .	33
1.3.4 Deployment Build Configuration . . . . .	34
1.3.5 Build Configuration Template . . . . .	34
1.3.6 Build Log . . . . .	38
1.3.7 Build Queue . . . . .	39
1.3.8 Build State . . . . .	41
1.3.9 Build Tag . . . . .	45
1.3.10 Change . . . . .	45
1.3.11 Change State . . . . .	46
1.3.12 Difference Viewer . . . . .	47
1.3.13 Project . . . . .	48
1.3.14 Build Agent . . . . .	49
1.3.15 Agent Home Directory . . . . .	51
1.3.16 Agent Requirements . . . . .	52
1.3.17 Agent Work Directory . . . . .	53
1.3.18 Authentication Modules . . . . .	53
1.3.19 Build Chain . . . . .	53
1.3.20 Build Checkout Directory . . . . .	56
1.3.21 Build Grid . . . . .	58
1.3.22 Build History . . . . .	58
1.3.23 Build Number . . . . .	58
1.3.24 Build Runner . . . . .	58
1.3.25 Build Working Directory . . . . .	59
1.3.26 Clean Checkout . . . . .	59
1.3.27 Clean-Up . . . . .	60
1.3.28 Code Coverage . . . . .	63
1.3.29 Code Duplicates . . . . .	63
1.3.30 Code Inspection . . . . .	64
1.3.31 Continuous Integration . . . . .	64
1.3.32 Dependent Build . . . . .	64
1.3.33 Guest User . . . . .	66
1.3.34 History Build . . . . .	67
1.3.35 Notifier . . . . .	67
1.3.36 Personal Build . . . . .	67
1.3.37 Pinned Build . . . . .	68
1.3.38 Pre-Tested (Delayed) Commit . . . . .	68
1.3.39 Remote Run . . . . .	69
1.3.40 Role and Permission . . . . .	69
1.3.41 Run Configuration Policy . . . . .	72
1.3.42 TeamCity Data Directory . . . . .	72
1.3.43 TeamCity Specific Directories . . . . .	75
1.3.44 User Account . . . . .	75
1.3.45 User Group . . . . .	76
1.3.46 Wildcards . . . . .	76
1.3.47 Already Fixed In . . . . .	77
1.3.48 First Failure . . . . .	78
1.3.49 Super User . . . . .	78
1.3.50 Identifier . . . . .	79
1.3.51 VCS root . . . . .	80
1.3.52 Remote Debug . . . . .	80
1.3.53 Favorite Build . . . . .	81
1.3.54 Agent Cloud Profile . . . . .	82
1.3.55 TeamCity Home Directory . . . . .	83
1.3.56 Revision . . . . .	84
1.3.57 Mapping TeamCity Concepts to Other CI Terms . . . . .	84
1.4 Supported Platforms and Environments . . . . .	85
1.4.1 Testing Frameworks . . . . .	90
1.4.2 Code Quality Tools . . . . .	90
1.5 Installation and Upgrade . . . . .	92
1.5.1 Installation . . . . .	92
1.5.1.1 Installing and Configuring the TeamCity Server . . . . .	93
1.5.1.1.1 CSRF Protection . . . . .	101
1.5.1.1.2 Running TeamCity Stack in AWS . . . . .	103

1.5.1.2 Setting up and Running Additional Build Agents .....	105
1.5.1.2.1 Build Agent Configuration .....	114
1.5.1.3 TeamCity Integration with Cloud Solutions .....	116
1.5.1.3.1 Setting Up TeamCity for Amazon EC2 .....	119
1.5.1.3.2 Setting Up TeamCity for VMware vSphere and vCenter .....	123
1.5.1.4 Disabling TeamCity Plugins .....	124
1.5.1.5 Installing Additional Plugins .....	125
1.5.1.6 Installing Agent Tools .....	126
1.5.2 Upgrade Notes .....	126
1.5.3 Upgrade .....	158
1.5.4 TeamCity Maintenance Mode .....	163
1.5.5 Setting up an External Database .....	164
1.5.5.1 Setting up TeamCity with MS SQL Server .....	168
1.5.5.2 Configuring UTF8 Character Set for MySQL .....	172
1.5.5.3 Using AWS Aurora Database Cluster .....	173
1.5.6 Migrating to an External Database .....	174
1.6 User's Guide .....	176
1.6.1 Managing your User Account .....	177
1.6.2 Subscribing to Notifications .....	177
1.6.3 Viewing Your Changes .....	181
1.6.4 Working with Build Results .....	182
1.6.5 Investigating and Muting Build Problems .....	186
1.6.6 Viewing Tests and Configuration Problems .....	188
1.6.7 Viewing Build Configuration Details .....	190
1.6.8 Statistic Charts .....	191
1.6.9 Search .....	193
1.6.10 Maven-related Data .....	196
1.7 Administrator's Guide .....	197
1.7.1 TeamCity Configuration and Maintenance .....	197
1.7.1.1 Configuring Authentication Settings .....	198
1.7.1.1.1 LDAP Integration .....	201
1.7.1.1.2 NTLM HTTP Authentication .....	210
1.7.1.1.3 Enabling Guest Login .....	212
1.7.1.1.4 Enabling Email Verification .....	212
1.7.1.2 TeamCity Data Backup .....	213
1.7.1.2.1 Creating Backup from TeamCity Web UI .....	214
1.7.1.2.2 Creating Backup via maintainDB command-line tool .....	215
1.7.1.2.3 Manual Backup and Restore .....	217
1.7.1.2.4 Backing up Build Agent's Data .....	219
1.7.1.2.5 Restoring TeamCity Data from Backup .....	219
1.7.1.3 Projects Import .....	222
1.7.1.4 Project Export .....	224
1.7.1.5 TeamCity Startup Properties .....	225
1.7.1.6 Configuring Server URL .....	226
1.7.1.7 Configuring TeamCity Server Startup Properties .....	226
1.7.1.7.1 TeamCity Tweaks .....	227
1.7.1.8 Setting up Google Mail and Google Talk as Notification Servers .....	228
1.7.1.9 Using HTTPS to access TeamCity server .....	228
1.7.1.10 TeamCity Disk Space Watcher .....	231
1.7.1.11 TeamCity Server Logs .....	231
1.7.1.12 Build Agents Configuration and Maintenance .....	234
1.7.1.12.1 Agent Pools .....	235
1.7.1.12.2 Configuring Build Agent Startup Properties .....	236
1.7.1.12.3 Viewing Agents Workload .....	237
1.7.1.12.4 Viewing Build Agent Details .....	238
1.7.1.12.5 Viewing Build Agent Logs .....	240
1.7.1.13 TeamCity Memory Monitor .....	241
1.7.1.14 Disk Usage .....	242
1.7.1.15 Server Health .....	243
1.7.1.16 Build Time Report .....	247
1.7.1.17 TeamCity Monitoring and Diagnostics .....	247
1.7.1.18 Uploading SSL Certificates .....	249
1.7.2 Several Nodes Setup .....	250
1.7.2.1 Configuring Read-Only Node .....	250
1.7.2.2 Configuring Running Builds Node .....	252
1.7.3 Configuring Cross-Server Projects Popup .....	255
1.7.4 Managing Projects and Build Configurations .....	255
1.7.4.1 Creating and Editing Projects .....	256
1.7.4.2 Configuring VCS Settings .....	260
1.7.4.2.1 Configuring VCS Roots .....	262
1.7.4.2.2 Working with Feature Branches .....	293
1.7.4.2.3 VCS Checkout Rules .....	299

1.7.4.2.4 VCS Checkout Mode . . . . .	300
1.7.4.3 Storing Project Settings in Version Control . . . . .	302
1.7.4.3.1 Kotlin DSL . . . . .	306
1.7.4.4 Creating and Editing Build Configurations . . . . .	323
1.7.4.4.1 Configuring General Settings . . . . .	325
1.7.4.4.2 NuGet . . . . .	329
1.7.4.4.3 Configuring Build Steps . . . . .	332
1.7.4.4.4 Adding Build Features . . . . .	396
1.7.4.4.5 Configuring Unit Testing and Code Coverage . . . . .	414
1.7.4.4.6 Build Failure Conditions . . . . .	437
1.7.4.4.7 Configuring Build Triggers . . . . .	439
1.7.4.4.8 Configuring Dependencies . . . . .	452
1.7.4.4.9 Configuring Build Parameters . . . . .	464
1.7.4.4.10 Configuring Agent Requirements . . . . .	476
1.7.4.5 Triggering a Custom Build . . . . .	477
1.7.4.6 Copy, Move, Delete Build Configuration . . . . .	479
1.7.4.7 Ordering Projects and Build Configurations . . . . .	480
1.7.4.8 Archiving Projects . . . . .	480
1.7.4.9 Ordering Build Queue . . . . .	480
1.7.4.10 Muting Test Failures . . . . .	482
1.7.4.11 Changing Build Status Manually . . . . .	483
1.7.4.12 Customizing Statistics Charts . . . . .	483
1.7.4.13 Configuring Artifacts Storage . . . . .	484
1.7.5 Licensing Policy . . . . .	485
1.7.5.1 Third-Party License Agreements . . . . .	488
1.7.6 Integrating TeamCity with Other Tools . . . . .	494
1.7.6.1 Integrating TeamCity with VCS Hosting Services . . . . .	494
1.7.6.2 Integrating TeamCity with Issue Tracker . . . . .	497
1.7.6.2.1 Bugzilla . . . . .	499
1.7.6.2.2 JIRA . . . . .	499
1.7.6.2.3 YouTrack . . . . .	499
1.7.6.2.4 GitHub . . . . .	500
1.7.6.2.5 Bitbucket Cloud . . . . .	500
1.7.6.2.6 Team Foundation Work Items . . . . .	501
1.7.6.3 Mapping External Links in Comments . . . . .	501
1.7.6.4 External Changes Viewer . . . . .	502
1.7.6.5 Integrating TeamCity with Docker . . . . .	502
1.7.6.5.1 Configuring Connections to Docker . . . . .	506
1.7.6.5.2 Docker Wrapper . . . . .	507
1.7.7 Managing User Accounts, Groups and Permissions . . . . .	508
1.7.7.1 Managing Users and User Groups . . . . .	508
1.7.7.2 Viewing Users and User Groups . . . . .	510
1.7.7.3 Managing Roles . . . . .	510
1.7.8 Customizing Notifications . . . . .	511
1.7.9 Assigning Build Configurations to Specific Build Agents . . . . .	515
1.7.10 Patterns For Accessing Build Artifacts . . . . .	516
1.7.11 Mono Support . . . . .	518
1.7.12 Maven Server-Side Settings . . . . .	519
1.7.13 Tracking User Actions . . . . .	520
1.7.14 Jenkins to TeamCity Migration Guidelines . . . . .	520
1.8 Installing Tools . . . . .	523
1.8.1 IntelliJ Platform Plugin . . . . .	523
1.8.1.1 IntelliJ Platform Plugin Compatibility . . . . .	524
1.8.2 Eclipse Plugin . . . . .	525
1.8.3 Visual Studio Addin . . . . .	526
1.8.4 Windows Tray Notifier . . . . .	527
1.8.5 Syndication Feed . . . . .	531
1.9 Extending TeamCity . . . . .	532
1.9.1 Build Script Interaction with TeamCity . . . . .	533
1.9.2 Accessing Server by HTTP . . . . .	545
1.9.3 REST API . . . . .	547
1.9.4 Including Third-Party Reports in the Build Results . . . . .	564
1.9.5 Custom Chart . . . . .	566
1.9.5.1 Edit Custom Chart Limitations . . . . .	572
1.9.6 Developing TeamCity Plugins . . . . .	572
1.9.6.1 Getting Started with Plugin Development . . . . .	573
1.9.6.2 Typical Plugins . . . . .	576
1.9.6.2.1 Build Runner Plugin . . . . .	576
1.9.6.2.2 Risk Tests Reordering in Custom Test Runner . . . . .	578
1.9.6.2.3 Custom Build Trigger . . . . .	579
1.9.6.2.4 Extending Notification Templates Model . . . . .	580
1.9.6.2.5 Issue Tracker Integration Plugin . . . . .	580

1.9.6.2.6 Version Control System Plugin . . . . .	582
1.9.6.2.7 Version Control System Plugin (old style - prior to 4.5) . . . . .	586
1.9.6.2.8 Custom Authentication Module . . . . .	592
1.9.6.2.9 Custom Notifier . . . . .	597
1.9.6.2.10 Custom Statistics . . . . .	597
1.9.6.2.11 Custom Server Health Report . . . . .	598
1.9.6.2.12 Extending Highlighting for Web diff view . . . . .	600
1.9.6.2.13 External Storage Implementation Guide . . . . .	606
1.9.6.3 Bundled Development Package . . . . .	608
1.9.6.4 Open API Changes . . . . .	609
1.9.6.5 Plugin Types in TeamCity . . . . .	614
1.9.6.6 Plugins Packaging . . . . .	615
1.9.6.7 Server-side Object Model . . . . .	620
1.9.6.8 Agent-side Object Model . . . . .	622
1.9.6.9 Extensions . . . . .	622
1.9.6.10 Web UI Extensions . . . . .	623
1.9.6.11 Plugin Settings . . . . .	628
1.9.6.12 Development Environment . . . . .	628
1.9.6.13 Developing Plugins Using Maven . . . . .	629
1.9.6.14 Plugin Development FAQ . . . . .	631
1.10 How To... . . . . .	631
1.11 Troubleshooting . . . . .	657
1.11.1 Common Problems . . . . .	657
1.11.2 Known Issues . . . . .	664
1.11.3 Reporting Issues . . . . .	672
1.11.4 Applying Patches . . . . .	681
1.11.5 Visual C Build Issues . . . . .	683
1.12 Getting Started with NUnit . . . . .	683
1.13 Getting started with PHP . . . . .	689
1.14 Continuous Delivery to Windows Azure Web Sites (or IIS) . . . . .	700

# TeamCity Documentation

Welcome to the documentation space for TeamCity 2018.x! If you are using an earlier TeamCity version, please refer to [documentation for your release](#).

## Get Started

- Getting Started with TeamCity
- TeamCity Concepts

## Install

- Install TeamCity Server
- Set up Additional Build Agents

## Administer

- Configure and Maintain TeamCity server
- Work with Projects and Build Configurations
- Manage Users and their Permissions
- Upgrade

## Extend and Develop

- Customize your TeamCity
- REST API
- Available TeamCity Plugins
- Develop your Own Plugin

## Find Answers

- How To...
- Troubleshooting
- Common Problems
- Known Issues
- Public Issue Tracker
- Community Forum
- Video User Guide

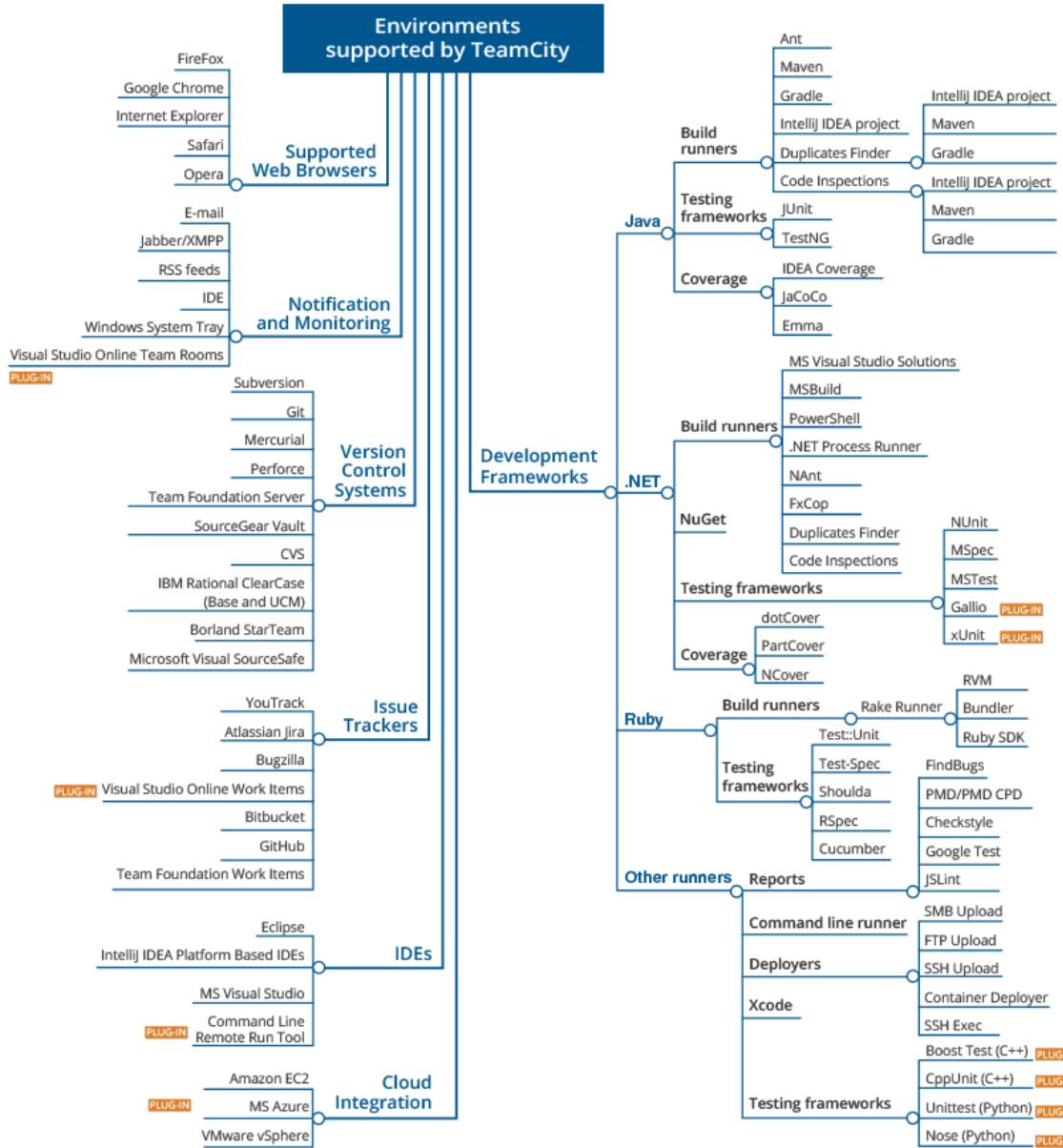
## Learn More and Contact Us

- TeamCity Official Site
- Official TeamCity Blog
- Feedback

## TeamCity 2018.x Supported Platforms and Environments

The core features of TeamCity are [platform-independent](#).

Use the clickable diagram below to have a "10,000-foot look" at TeamCity and the supported IDE's, frameworks, version control systems, and means of monitoring. See details at [Supported Platforms and Environments](#).



## Copyright and Trademark Notice

The software described in this documentation is furnished under a software license agreement. JetBrains, IntelliJ, IntelliJ IDEA, YouTrack and TeamCity are trademarks or registered trademarks of JetBrains, s.r.o. Windows is a registered trademark of Microsoft Corporation in the United States and other countries. Mac, Mac OS, macOS are trademarks of Apple Inc., registered in the U.S. and other countries. Linux is a registered trademark of Linus Torvalds. All other trademarks are the properties of their respective owners.

## What's New in TeamCity 2018.1

- Portable Kotlin DSL format
  - Create Project from URL recognizes Kotlin DSL settings
- Read-Only Node
- User Interface Changes
- Uploading SSL / HTTPS Certificates
- Inherited build steps configuration improvements
  - Ability to have pre- and post-steps in a template
  - Ability to redefine inherited build step settings

- Enforced settings
- Project Level NuGet Feed
- Docker Support
- Bundled Amazon S3 Artifact Storage
- PowerShell Core support
- Rerun build action improvements
- Shared Resources Improvements
- Other improvements
- Fixed Issues
- Previous Releases

## Portable Kotlin DSL format

TeamCity 2018.1 supports a much simpler format of Kotlin DSL settings - portable DSL. It allows sharing of the same Kotlin DSL scripts between different servers and even reusing the same Kotlin DSL script by several projects on the same server.

The portable Kotlin DSL has the following properties:

- in the portable format, TeamCity generates a single `settings.kts` file
- by default Kotlin scripts in the portable DSL do not have ids (both `uuid` and `id` are optional)
- the versioned settings themselves as well as the definition of a VCS root where the settings are stored are not available in the portable Kotlin DSL and cannot be changed via DSL

In this simplified format TeamCity also generates a single `settings.kts` file with all project settings. This is what this file looks like for a very simple project:

```
import jetbrains.buildServer.configs.kotlin.v2018_1.*
import jetbrains.buildServer.configs.kotlin.v2018_1.buildSteps.script
version = "2018.1"
project {
    buildType(HelloWorld)
}
object HelloWorld : BuildType({
    name = "Hello world"
    steps {
        script {
            scriptContent = "echo 'Hello world!''"
        }
    }
})
```

See details in our [documentation](#).

## Create Project from URL recognizes Kotlin DSL settings

Now, when you start creating a project from URL, TeamCity will scan the repository for presence of the `.teamcity/settings.kts` file and, if the file is found, it will offer to import the settings from Kotlin DSL:

## Create Project From URL

 The connection to the VCS repository has been verified

 The VCS repository contains **.teamcity/settings.kts** file with settings of some project. Would you like to import these settings?

- Import settings from **.teamcity/settings.kts**
- Do not import settings, create project from scratch

## Read-Only Node

TeamCity makes a step towards High Availability by introducing a read-only mode for the server. It is now possible to start a second, [read-only TeamCity server](#), that allows users read operations, such as view the build information, download artifacts, etc. during the downtime of the main server, e.g. during upgrade.

The Read-Only node uses the same Data Directory and the same external database as the main TeamCity Server, so a prerequisite of using the Read-Only node is to ensure that it has access to the data directory and database.

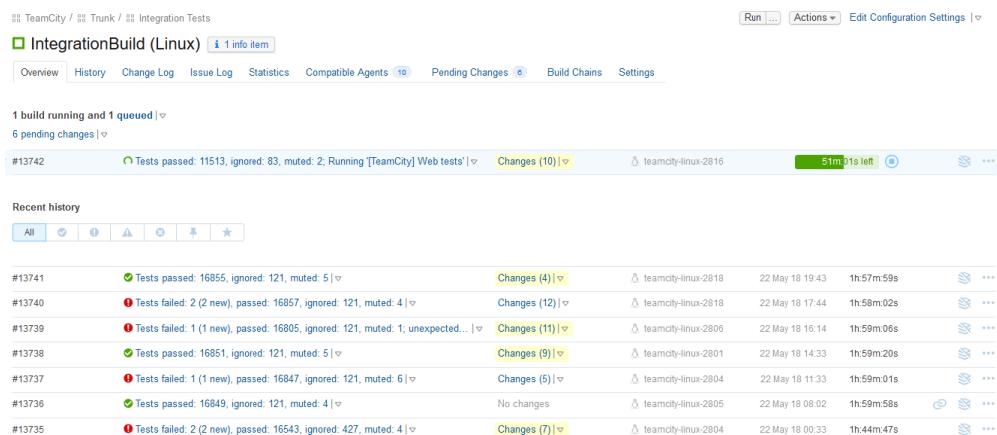
Using a read-only node in addition to the main TeamCity server, it's possible to set up a highly available TeamCity installation that will have zero read downtime, i.e. when the main server is unavailable or is performing an upgrade, requests will be routed to the read-only node. Such setup requires installing both the main server and the read-only node behind a reverse proxy, that should be configured to proxy requests to the main server while it's available and to the read-only one in other cases.

Note that an additional server in a high-availability set-up uses the license from the main server and does not require a separate license.

Details are available in the dedicated [documentation section](#).

## User Interface Changes

The build configuration home page has been redesigned (internally it now uses the REST API to show its data)



The screenshot shows the TeamCity interface for the 'IntegrationBuild (Linux)' configuration. At the top, there are navigation links for 'Overview', 'History', 'Change Log', 'Issue Log', 'Statistics', 'Compatible Agents', 'Pending Changes' (with 10 pending changes), 'Build Chains', and 'Settings'. Below this, a summary indicates '1 build running and 1 queued' and '6 pending changes'. A specific build entry for '#13742' is highlighted, showing 'Tests passed: 11513, ignored: 83, muted: 2; Running [TeamCity] Web tests' and 'Changes (10)'. The build status is 'teamcity-linux-2816' and it has '51m31s left'. The build history table lists recent builds from '#13741' to '#13735', each with details about test results, changes, and duration. Artifacts icons are visible next to the build numbers in the history table.

Build ID	Status	Test Details	Changes	Duration	Agent	Actions
#13741		Tests passed: 16855, ignored: 121, muted: 5	Changes (4)	22 May 18 19:43	1h:57m:59s	
#13740		Tests failed: 2 (2 new), passed: 16857, ignored: 121, muted: 4	Changes (12)	22 May 18 17:44	1h:58m:02s	
#13739		Tests failed: 1 (1 new), passed: 16805, ignored: 121, muted: 1; unexpected...	Changes (11)	22 May 18 16:14	1h:59m:06s	
#13738		Tests passed: 16951, ignored: 121, muted: 5	Changes (9)	22 May 18 14:33	1h:59m:20s	
#13737		Tests failed: 1 (1 new), passed: 16847, ignored: 121, muted: 6	Changes (5)	22 May 18 11:33	1h:59m:01s	
#13736		Tests passed: 16849, ignored: 121, muted: 4	No changes	22 May 18 08:02	1h:59m:58s	
#13735		Tests failed: 2 (2 new), passed: 16543, ignored: 427, muted: 4	Changes (7)	22 May 18 00:33	1h:44m:47s	

Besides, TeamCity uses the new artifacts icon to indicate the presence of artifacts in a build.

## Uploading SSL / HTTPS Certificates

TeamCity now allows [uploading a trusted certificate](#), e.g. be a self-signed certificate, or a certificate signed by a not well-known certificate authority (CA). The uploaded certificate becomes trusted by TeamCity, which means TeamCity will be able to open a connection by HTTPS or SSL protocols to a resource with this certificate. Currently the trusted certificates storage is global for the whole server and affects all of the server projects.

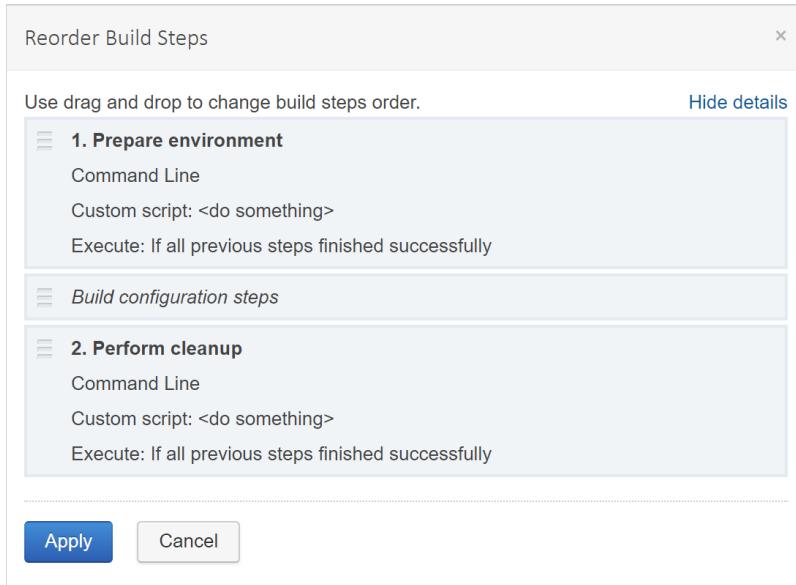
See details in [our documentation](#).

## Inherited build steps configuration improvements

### Ability to have pre- and post-steps in a template

There is sometimes a need to define a common build step in a template, so that this step will be executed either before all build configuration steps or after them.

Starting with this version, for a given template it is possible to define such steps and then define their placement in respect to the build configuration steps. All build configuration steps are represented as a placeholder in the Reorder Build Steps dialog. The template steps can be placed before or after this placeholder.



Note: you still can have a completely custom order of steps in a build configuration inherited from a template.

### Ability to redefine inherited build step settings

Starting with this version it is possible to change inherited build steps right in the build configuration.

## Enforced settings

TeamCity now provides the ability to enforce settings for all of the build configurations in a project hierarchy. For instance, with help of this feature it is possible to enforce agent side checkout everywhere, or make sure that all build configurations have some strict execution timeout.

To enforce some settings in the project hierarchy, a template with these settings must be created. After that, a system administrator can set this template as the enforced settings template in the project:

Name: \*

Project ID: \*  [Regenerate ID](#)

Description:

Default template: [TeamCity / Trunk / Default Template](#)

Enforced settings template: [TeamCity / Trunk / Default Template](#)

<No enforced settings template selected>

<No enforced settings template selected>

TeamCity

Vagrant Environment

Trunk

Default Template

DistMiddleStep

dotnet-listeners tests

JPS Template

To some extent, the enforced settings template works similarly to the default template - i.e. all of its settings are inherited in build configurations of the project hierarchy. The difference is that these inherited settings cannot be disabled or overridden anyhow. In addition to that, only system administrator can associate a project with a specific enforced settings template -

project administrator permissions are not sufficient. On the other hand, the template itself can be edited by a project administrator who can administer the project where the template is defined.

If the enforced settings template is specified in a project and another template is assigned as the enforced settings in a subproject, the subproject's template will have the higher priority.

Currently it is not possible to enforce VCS roots, build steps, triggers, dependencies and requirements. Let us know if you see valid use cases when this would be useful.

## Project Level NuGet Feed

In previous versions of TeamCity, NuGet feed was global for the whole server. Not only that, but once the feed was enabled, TeamCity started indexing of all of the .nupkg files published as artifacts. On a large server this could lead to a really large feed, delays with packages appearance, increased disk usage for the feed metadata and sometimes very low performance which was one of the most common complaints in the past.

It is now possible to enable a [NuGet feed at the project level](#), which means each project can have its own NuGet feed. By default, TeamCity no longer adds all .nupkg artifacts to the project feed; instead, you can add the [NuGet packages indexer](#) build feature to build configurations whose artifacts should be indexed. If you prefer to have a single feed for the whole server, you can enable it at the Root project level and add the packages indexer build feature to build configurations which produce relevant packages. Alternatively, you can enable automatic packages indexing on the project NuGet Feed page. In this case, as it was before, all of the .nupkg artifacts produced by this project will be indexed automatically.

## Docker Support

- [Docker](#) command runner with support for build, push and some other docker commands has replaced the Docker Build runner
- The Docker wrapper now supports .NET CLI and PowerShell runners, which means you can easily run these steps in a Docker container
- The Docker wrapper can now use Gradle and Maven provided by the Docker images in the corresponding build steps
- The [Docker Compose](#) supports multiple [Docker Compose YAML file\(s\)](#), space-separated paths to the docker-compose.yml files should be relative to the [checkout directory](#).
- AWS ECR (Elastic Container Registry) is now supported

## Bundled Amazon S3 Artifact Storage

TeamCity now comes [bundled with Amazon S3 plugin](#), which allows uploading to, downloading, and removing artifacts from S3. Resolution of artifact dependencies as well as clean-up of artifacts is handled by the plugin. The artifacts located externally are displayed in the TeamCity web UI.

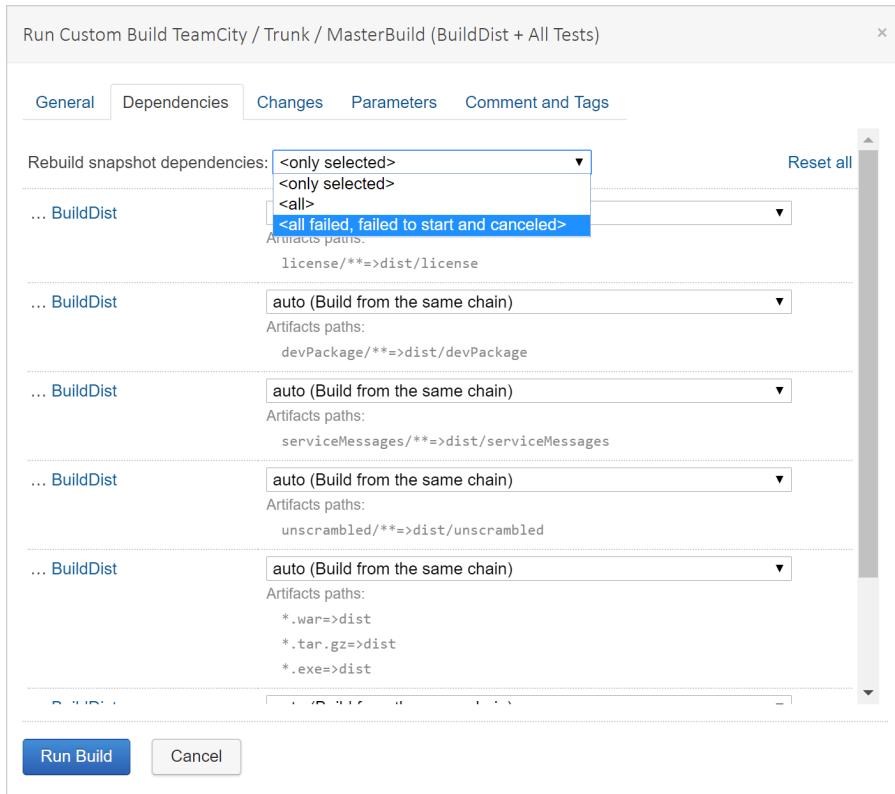
## PowerShell Core support

- Cross-platform PowerShell ([PowerShell Core](#)) is now supported on Windows, MacOS and Linux
- Side-by-side installations of PowerShell Desktop and PowerShell Core are supported under Windows

## Rerun build action improvements

Now when rerunning a build, TeamCity preserves all the custom parameters of this build and its dependencies as of the time of the original run.

Besides that, on the dependencies tab, there is now an additional option to rerun the failed/failed to start or cancelled dependencies of the build:



## Shared Resources Improvements

- You can view the resources used by the build on the Shared Resources tab of the [Build Results](#) page if at least one resource lock is defined in a build configuration. The tab displays the resources and their type, including the locks used by the build for each resource.
- Shared resources can now be locked not only for regular builds, but for composite builds as well. A lock on the specified resource will be acquired when a composite build starts (when the first build of the composite build chain starts); the lock will be released when the composite build finishes (the last build in its chain is finished). The locks acquired on composite builds affect only these composite builds and are not propagated to their individual parts.

## Other improvements

- Browsing of artifacts inside archives is now supported for all external artifacts storages too (previously this feature was available only if artifacts were located in the internal artifacts storage).
- Java 10 can now be used to run an agent, though Java 8 is still recommended.
- Gerrit Commit Status Publisher now allows an admin to configure custom values for success and failure instead of +1 and -1 respectively. This label is also configurable. Note that since TeamCity 2018.1 Gerrit 2.6. is the minimal version supported by Commit Status Publisher.
- The build chains page has got a new grouping option: it groups together in a single node all of the build configurations that are not related to the current context.
- Duplicates finder (.NET) and Inspections (.NET) runners have been renamed to Duplicates finder (ReSharper) and Inspections (ReSharper) respectively. You can now select the platform bitness for InspectCode when using Inspections (ReSharper).
- The bundled ReSharper CLT and dotCover have been updated to version 2018.1.2
- The new option to copy build configurations build numbers has been added to the copy project dialog

## Fixed Issues

[Full list of fixed issues](#)

## Previous Releases

[What's New in TeamCity 2017.2](#)

# Getting Started with TeamCity

JetBrains TeamCity is a powerful and user-friendly Continuous Integration and Deployment server that works out of the box.

This guide provides basic information on TeamCity features and capabilities and includes instruction on the evaluation TeamCity setup. Details on [installation](#) and more complex [production](#) configuration adjusted to your needs are available in the [Installation Guide](#).

## Before you start

Check the full list of [Supported platforms and environments](#).

### 1. Learn about CI with TeamCity

Understand the idea behind Continuous integration, learn [basic TeamCity concepts](#) and [build lifecycle](#).

### 2. Install and Start TeamCity

Get [information](#) required to download, install and start TeamCity on Windows, on Linux and OS X.

### 3. Run your First Build

Create your [first project](#) in TeamCity and configure and run your first build.

## Continuous Integration with TeamCity

On this page:

- [What is Continuous Integration?](#)
- [What is TeamCity?](#)
  - [What can you do with TeamCity?](#)
- [Basic TeamCity concepts](#)
  - [Basic CI Workflow in TeamCity](#)

### What is Continuous Integration?

Continuous Integration is a software development practice in which developers commit code changes into a shared repository several times a day. Each commit is followed by an automated build to ensure that new changes integrate well into the existing code base and to detect problems early. To learn more about continuous integration basics, please refer to [Martin Fowler's article](#).

### What is TeamCity?

JetBrains TeamCity is a user-friendly continuous integration (CI) server for developers and build engineers [free of charge](#) with the [Professional Server License](#) and easy to set up!

### What can you do with TeamCity?

- Run parallel builds simultaneously on different platforms and environments
- Optimize the code integration cycle and be sure you never get broken code in the repository
- Review on-the-fly test results reporting with intelligent tests re-ordering
- Run code coverage and duplicates finder for Java and .NET
- Customize statistics on build duration, success rate, code quality, and custom metrics
- and much more.

To learn more about major TeamCity features, refer to the [official JetBrains site](#).  
The complete list of supported platforms and environments can be found [here](#).

### Basic TeamCity concepts

This section explains the main concepts. The complete list can be found [here](#).

The TeamCity build system comprises the server and Build Agents.

Concept	Description
Build Agent	<p>A piece of software that actually executes a build process. It is installed and configured separately from the TeamCity server, i.e. the agent can be installed on a separate machine (physical or virtual, and it can run the same operating system (OS) as the server or a different OS).</p> <p>Build Agents in TeamCity can have different platforms, operating systems, and pre-configured environments that you may want to test your software on. Different types of tests can be run under different platforms simultaneously so the developers get faster feedback and more reliable testing results.</p> <div style="border: 1px solid #f0e68c; padding: 5px; margin-top: 10px;">  It is possible for the server and an agent to coexist on the same computer, but for production purposes we recommend installing them on different machines for a number of reasons, the server performance being the most important.         </div>
TeamCity Server	<p>The server itself does not run either builds or tests: the server's job is to monitor all the connected build agents, distribute <a href="#">queued builds</a> to the agents based on compatibility requirements, and report the results. All information on the build results (build history and all the build-associated data except for artifacts and build logs), VCS changes, agents, build queue, user accounts and user permissions, etc. are stored in a <a href="#">database</a>.</p> <div style="border: 1px solid #f0e68c; padding: 5px; margin-top: 10px;">  It is possible for the server and an agent to coexist on the same computer, but for production purposes we recommend installing them on different machines for a number of reasons, the server performance being the most important.         </div>
Project	A TeamCity Project corresponds to a software project, or a specific version/release of a software project. A project is a collection of build configurations.
Build Configuration	A combination of settings defining a build procedure. The settings include VCS Roots, Build Steps, Build Triggers described below.
VCS Root	A collection of version control settings (paths to sources, username, password, <a href="#">checkout mode</a> and other settings) that defines how TeamCity communicates with a version control (SCM) system to monitor changes and get sources for a build.
Build Step	A task to be executed. Each build step is represented by a <a href="#">build runner</a> providing integration with a specific build tool (like Ant, Gradle, MSBuild, etc), a testing framework (e.g. NUnit), or a code analysis engine. Thus, in a single build you can have several steps and sequentially invoke test tools, code coverage, and, for instance, compile your project.
Build Trigger	A rule which initiates a new build on certain events. For example, a <a href="#">VCS trigger</a> will automatically start a new build each time TeamCity detects a change in the configured <a href="#">VCS roots</a> .
Change	Any modification of the source code which you introduce. If a change has been committed to the version control system, but not yet included in a build, it is considered pending for a certain build configuration.
Build	Refers to both: the actual process of creating an application version and the version itself. After the build process is triggered, it is put into the <a href="#">build queue</a> and is started when there are agents available to run it. After the build is finished, the build agent sends <a href="#">Build Artifacts</a> to the server.
Build Queue	A list of builds that were <a href="#">triggered</a> and are waiting to be started. TeamCity will distribute them to <a href="#">compatible</a> build agents as soon as the agents become idle. A queued build is assigned to an agent at the moment when it is started on the agent; no pre-assignment is made while the build is waiting in the build queue.
Build Artifacts	Files produced by a build, for example, installers, WAR files, reports, log files, etc, when they become available for download.

## Basic CI Workflow in TeamCity

To understand the data flow between the server and the agents, what is passed to the agents, how and when TeamCity gets the results, let's take a look at a simple build lifecycle.

- The TeamCity server detects a change in your VCS Root and stores it in the database.
- The build trigger sees the change in the database and adds a build to the queue.
- The server finds an idle compatible build agent and assigns the queued build to this agent.
- The agent executes the Build Steps. While the build steps are being executed, the build agent reports the build progress to the TeamCity server sending all the log messages, test reports, code coverage results, etc. to the server on the fly, so you can monitor the build process in real time.

- e. After finishing the build, the build agent sends [Build Artifacts](#) to the server.

Now you can proceed with [TeamCity installation](#).

## Installation Quick Start

This page covers the evaluation setup of a TeamCity server with a default build agent running on the same machine for the most popular operating systems.

 Note that for production purposes it is recommended to set up the TeamCity server and Agent on separate machines.

 You can also use the [TeamCity Server](#) and [TeamCity Build Agent Docker](#) images. The instructions are available on the corresponding Docker Hub pages.

On this page:

- [Download TeamCity](#)
- [Install TeamCity](#)
  - on Windows
  - on Linux and OS X
- [Start TeamCity for the First Time](#)

### Download TeamCity

Download TeamCity to install the free Professional Edition, the full-featured TeamCity bundled with 3 build agents with a limit of 100 build configurations since TeamCity 2017.2 (20 in earlier versions). After evaluation, you can switch to the Enterprise edition: [Licensing Policy](#) provides details. The pricing is available on the [JetBrains site](#).

[DOWNLOAD](#)

Earlier versions are available on the [Previous Releases Downloads](#) page.

### Install TeamCity

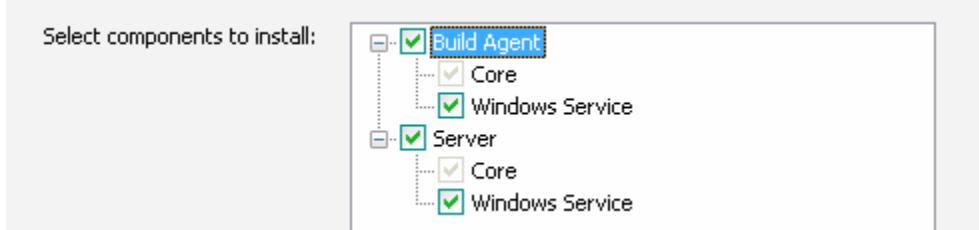
on Windows

Run the .exe file and follow the instructions of the TeamCity Setup wizard. The TeamCity web server and one build agent will be installed on the same machine.

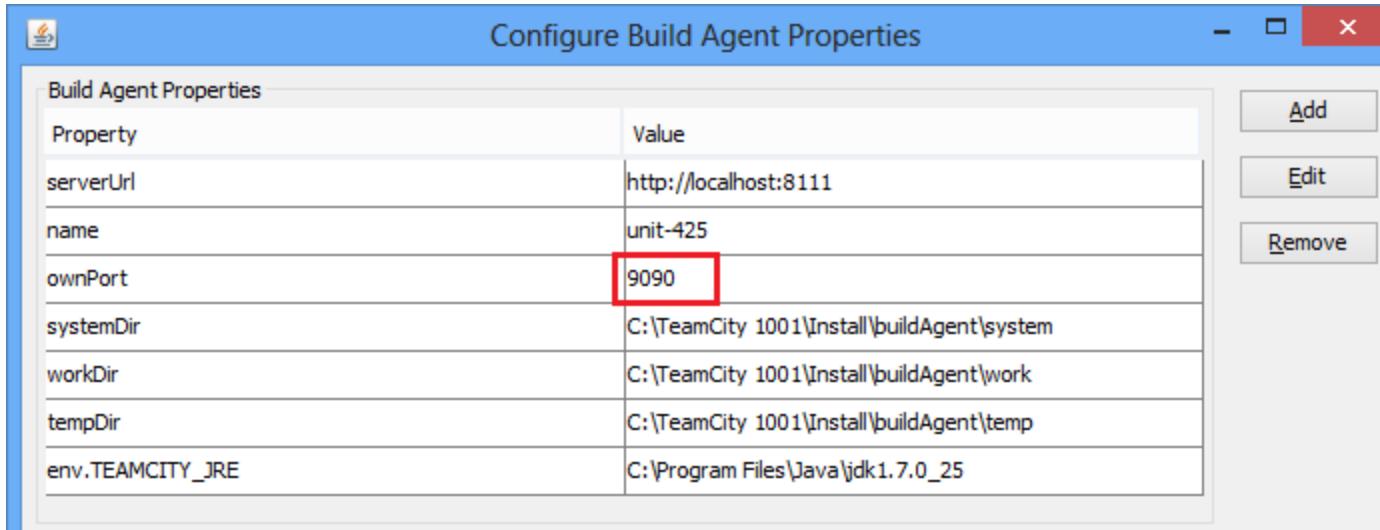
 Note that for [production purposes](#) it is recommended to set up the TeamCity server and Agent on separate machines.

During installation, you can configure the following:

1. The [TeamCity home](#) directory where TeamCity will be installed.
2. Whether the TeamCity server and agent will run as Windows services.



3. The server port: 80 is the default port, which can be already used by other applications (e.g. Skype). Change the server port if it is already in use. In the example below we've set the port to 8111.
4. The agent port: 9090 is the default for incoming connections from the server. If the port is already in use, you'll be asked to change it by setting the ownPort property to a different value.



If the TeamCity server is installed as a Windows service, follow the [usual procedure](#) of starting and stopping services.

Otherwise, to start/stop the TeamCity server and one default agent at the same time, use the `runAll` script, provided in the <`TeamCity home`>/bin directory, e.g.

- To start the server and the default agent, use

```
runAll.bat start
```

- To stop the server and the default agent, use

```
runAll.bat stop
```

If you did not change the default port (80) during the installation, the TeamCity web UI can be accessed via <http://localhost> in a web browser running on the same machine where the server is installed. Otherwise use <http://localhost:<port>> (<http://localhost:8111> in our case) to access the TeamCity server and build agent running on the same computer.

on Linux and OS X

1. Make sure you have [JRE or JDK](#) installed. Since TeamCity 10.0, Oracle Java 1.8 JDK is required for this test installation. Open a command-line terminal and run the following command:

```
java -version
```

2. Make sure the `JAVA_HOME` environment variable is pointing to the Java installation directory. Run the following command in the command-line terminal:

```
echo $JAVA_HOME
```

3. Use the `TeamCity<version number>.tar.gz` archive to manually install TeamCity bundled with Tomcat servlet container. Unpack the archive: e.g. under Linux use

```
tar xfz TeamCity<version number>.tar.gz
```

The archive can be used for installation on Windows as well.

4. The TeamCity web server and one build agent will be installed on the same machine.

 Note that for [production purposes](#) it is recommended to set up the TeamCity server and Agent on separate machines.

To start/stop the TeamCity server and one default agent at the same time, use the `runAll` script, provided in the `<TeamCity home>/bin` directory, e.g.

- To start the server and the default agent, use

```
runAll.sh start
```

- To stop the server and the default agent, use

```
runAll.sh stop
```

By default, TeamCity runs on <http://localhost:8111> and has one registered build agent that runs on the same computer. If another application uses the same port as the TeamCity server, the TeamCity server (Tomcat server) will not start with the "Address already in use" errors in the server logs or server console.

To change the server port, in the `<TeamCity Home>/conf/server.xml` file, change the port number in the `<Connector>` XML node, e.g.:

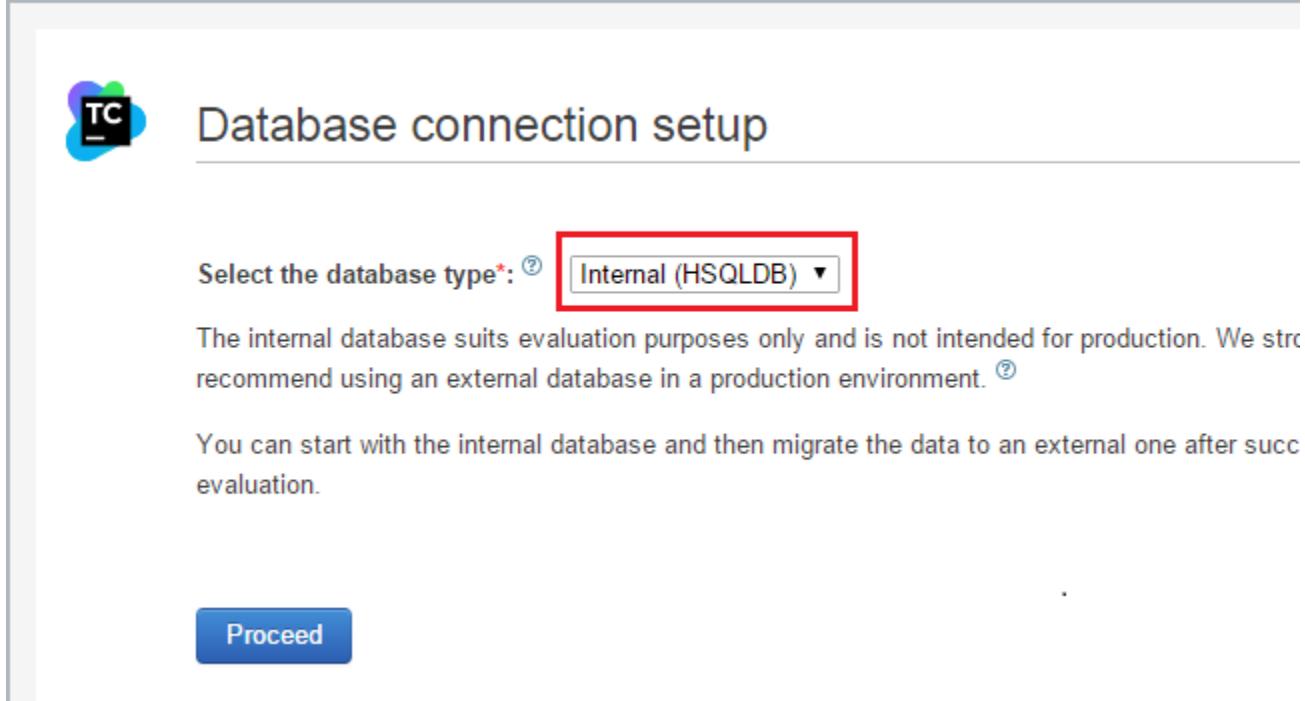
```
<Connector port="8111" ...
```

## Start TeamCity for the First Time

On the first TeamCity start, do the following:

1. Review the location of the TeamCity Data Directory, where all the configuration information is stored. Click Proceed.
2. TeamCity stores build history, users, build results and some run time data in an SQL database and allows you to select the database type.

For now, keep the default internal database. Click Proceed.



It'll take some time for TeamCity to configure the necessary components.

3. On the next screen, accept the License Agreement to proceed with the launch. Click Continue.
4. TeamCity displays the Create Administrator Account page. Specify the administrator credentials and click Create Account.

You are logged to TeamCity: now you can configure your user settings and [create](#) and [run your first build](#).

## Configure and Run Your First Build

After you installed and started TeamCity as described [here](#), the server is accessible locally on the default port (on Windows as <http://localhost/>, and on Linux/ OS X as <http://localhost:8111/>) and has one registered build agent that runs on the same computer.

Now we can get building! This section describes how you can:

- [Create your first project](#)
  - [Create a project from a repository URL](#)
  - [Create a project pointing to GitHub.com repository](#)
  - [Create a project manually](#)
- [Run your first build](#)
- [Tweak your build configuration settings](#)
  - [Artifacts](#)
  - [Automatic Build Trigger](#)
  - [Build Number Format](#)

### Create your first project

In TeamCity there is the default <Root project> containing all other projects in TeamCity. To create a project, use the Administration link at the top right corner and click Create project. The Create project page is displayed.

There are several options to create a project:

- [From a repository URL \(default\)](#)
- [From GitHub \(and similarly from Bitbucket Cloud repository and from Visual Studio Team Services\)](#)
- [Manually](#)

### Create a project from a repository URL

This is the fastest way to create your first build.

1. On the Create project page, click the From a repository URL option and paste the repository URL of your project into the field. The URL formats are listed [here](#). If required, specify your repository credentials.

**Create Project**

- From a repository URL
- From GitHub
- From Bitbucket Cloud
- From Visual Studio Team Services
- Manually

**Parent project:** <Root project>

**Repository URL:** https://github.com/Julia-Alexandrova/Sample-Project

**Username:**

**Password:**

**Proceed**

Click Proceed and follow the wizard.

- TeamCity will do the rest for you: it will identify the type of the VCS repository, test the connection and auto-configure VCS repository settings, as well as suggest the project and build configuration names.

**Create Project From URL**

✓ The connection to the VCS repository has been verified

**Project name:** Sample Project (2)

**Build configuration name:** Build

**VCS root:** (Git) https://github.com/Julia-Alexandrova/Sample-Project

**Proceed** **Cancel**

- Click Proceed.

Next TeamCity will scan your VCS repository and autodetect your build steps. Check the boxes of the steps and use them in your build configuration.

**Build**

Build Step	Parameters Description
Maven	Path to POM: pom.xml Goals: clean test

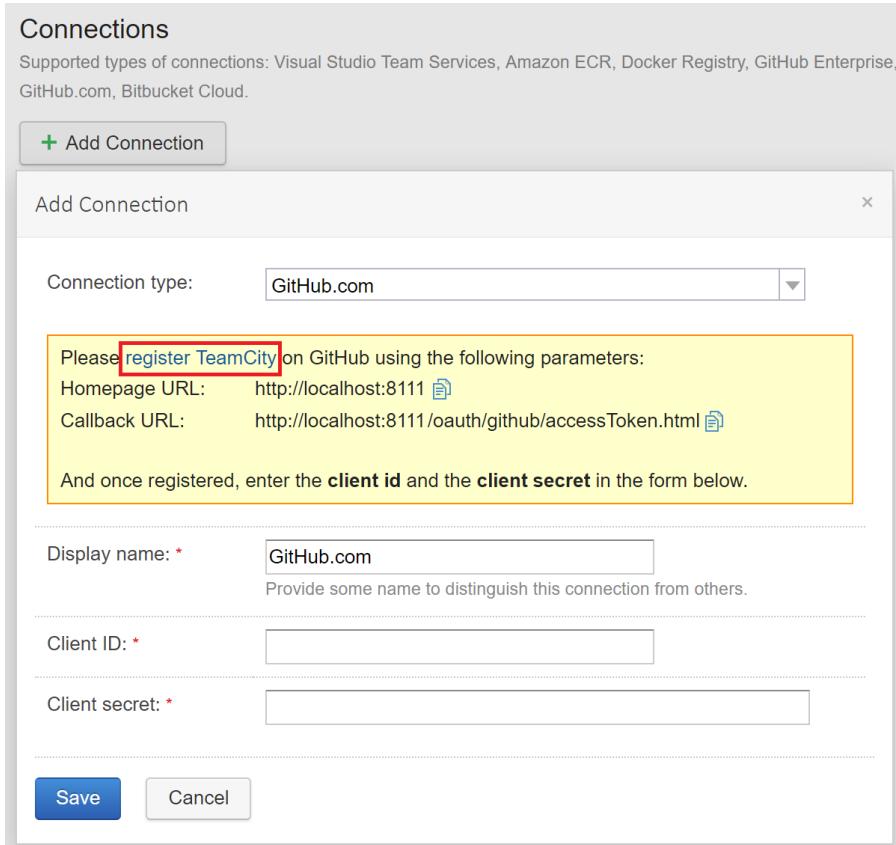
**Use selected** **Refresh**

The selected build step is added to the the build configuration.

4. Congratulations! You've configured your first build containing one build step. Now you can [run your build](#) and [tweak its settings](#) if necessary.

Create a project pointing to GitHub.com repository

1. Click the Create project button and select [Pointing to GitHub.com](#) repository.
2. The Connections page with the Add connection dialog opens. This provides the parameters to be used when registering your TeamCity application in GitHub service. Click the register TeamCity link.



3. The GitHub page opens. You need to register TeamCity as an [OAuth application](#) in GitHub. The following steps are performed in your GitHub account:
4. Log into your GitHub account. On the Register a new OAuth application page specify the name (and an optional

description), the homepage URL and the callback URL as provided by TeamCity. Use the  icon to copy the required parameters.

← → C GitHub, Inc. [US] | github.com/settings/applications/new

Search or jump to... / Pull requests Issues Marketplace Explore

## Register a new OAuth application

**Application name**  
TeamCity  
Something users will recognize and trust

**Homepage URL**  
http://localhost:8111  
The full URL to your application homepage

**Application description**  
Application description is optional  
This is displayed to all users of your application

**Authorization callback URL**  
http://localhost:8111/oauth/github/accessToken.html  
Your application's callback URL. Read our [OAuth documentation](#) for more information.

**Buttons:** Register application Cancel

1. Click Register application.
2. Scroll down and click Update application. The page is updated with Client ID and the client secret information for your TeamCity application.
3. Continue configuring the connection in TeamCity: on the Add Connection page, specify the Client ID and the client secret.

## Connections

Supported types of connections: Visual Studio Team Services, Amazon ECR, Docker Registry, GitHub Enterprise, GitHub.com, Bitbucket Cloud.

+ Add Connection

### Add Connection

Connection type: GitHub.com

Please register TeamCity on GitHub using the following parameters:  
Homepage URL: <http://localhost:8111>   
Callback URL: <http://localhost:8111/oauth/github/accessToken.html> 

And once registered, enter the **client id** and the **client secret** in the form below.

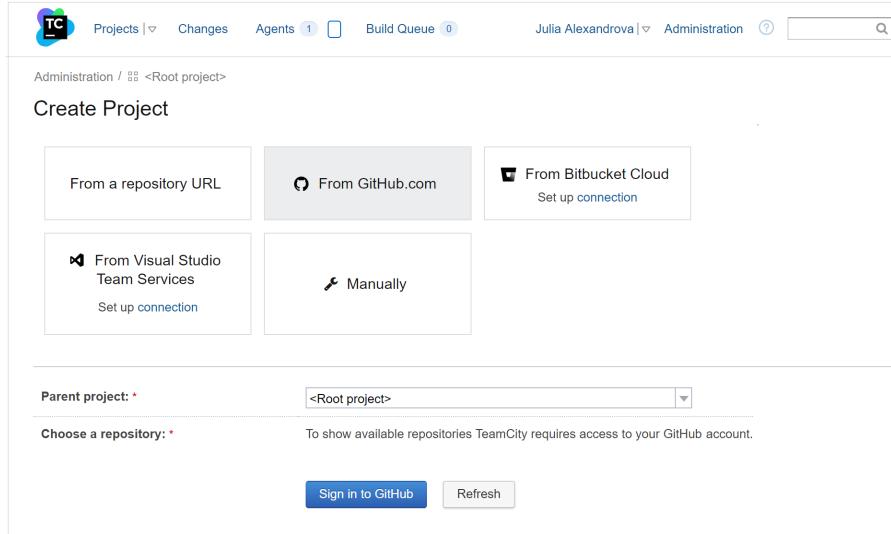
Display name: \* GitHub.com  
Provide some name to distinguish this connection from others.

Client ID: \* absd1234abcd1234

Client secret: \* 

**Save** **Cancel**

4. Save your settings.
5. Next you need to authorize TeamCity in the VCS: go to the Project administration, click Create project, select From GitHub and click Sign in to GitHub:



The screenshot shows the TeamCity Administration interface. At the top, there are navigation links: Projects, Changes, Agents (1), Build Queue (0), Julia Alexandrova, Administration, and a search bar. Below this, the page title is "Administration / <Root project>". The main section is titled "Create Project" and contains five options for creating a project: "From a repository URL", "From GitHub.com" (which is selected and highlighted in grey), "From Bitbucket Cloud" (with a "Set up connection" link), "From Visual Studio Team Services" (with a "Set up connection" link), and "Manually". Below these options, there is a "Parent project:" dropdown set to "<Root project>" and a "Choose a repository:" field with a note: "To show available repositories TeamCity requires access to your GitHub account." At the bottom of the form are two buttons: "Sign in to GitHub" (blue) and "Refresh" (grey).

6. On the page that opens, authorize the TeamCity application. The authorized application will be granted full control of private repositories and the Write repository hooks permission in GitHub.
7. Now you have a connection configured and you can continue with creating your project in TeamCity: All the repositories available to the user will be listed. Start typing to filter the list and select the required repository:

Administration / <Root project>

Create Project

From a repository URL

From GitHub.com

From Bitbucket Cloud  
Set up connection

From Visual Studio Team Services  
Set up connection

Manually

Parent project: \* <Root project>

Choose a repository: \* Found 188 repositories  
filter repositories>

Your repositories (44)

- App (<https://github.com/Julia-Alexandrova/App>)
- Calculator (<https://github.com/Julia-Alexandrova/Calculator>)
- dev (<https://github.com/Julia-Alexandrova/dev>)
- dotnet-components (<https://github.com/Julia-Alexandrova/dotnet-components>)
- Eclipse\_Sample1 ([https://github.com/Julia-Alexandrova/Eclipse\\_Sample1](https://github.com/Julia-Alexandrova/Eclipse_Sample1))

- TeamCity will verify the repository connection. If the connection is verified, the Create Project page opens. TeamCity will display the project and build configuration name. If required, modify the names and click Proceed. Next TeamCity will scan your VCS repository and autodetect your build steps (it may take some time). Check the boxes of the steps and use them in your build configuration.
- The selected build step is added to the the build configuration. Congratulations! You've configured the GitHub connection and your first build containing one build step. Now you can run your build and tweak its settings if necessary.

When the connection is configured, a small GitHub icon becomes active in several places of the UI where a repository URL can be specified (create project from URL, create a build configuration from URL, create VCS root from URL, create Git VCS root, create GitHub issue tracker for the current project and all of its subprojects), making it easier to create these entities in TeamCity.

## Create a project manually

You can create a project manually if auto-detection of settings is not suitable for you.

- Use the Administration link in the top right corner to go to the Administration area and click Create project and select Manually. Specify the project's name, ID (autogenerated, modifiable) and an optional description. Click Create:

Administration / <Root project>

## Create Project

- From a repository URL
- From GitHub Set up connection
- From Bitbucket Cloud Set up connection
- From Visual Studio Team Services Set up connection
- Manually

**Parent project:** \* <Root project>

**Name:** \* Sample project

**Project ID:** \* SampleProject  
This ID is used in URLs, REST API, HTTP requests to the server, and configuration settings in the TeamCity Data Directory.

- When a project is created, TeamCity prompts to populate it with build configurations. Click Create build configuration. They can be created automatically (similarly to creating projects) or manually as described below. If you select to do it manually, specify the build configuration name, **ID** (autogenerated, modifiable) and an optional description.

Administration / <Root project> / Sample Project

## Create Build Configuration

- From a repository URL
- From GitHub Set up connection
- From Bitbucket Cloud Set up connection
- From Visual Studio Team Services Set up connection
- Manually

**Parent project:** Sample Project

**Name:** \* Build

**Build configuration ID:** \* SampleProject\_Build  
This ID is used in URLs, REST API, HTTP requests to the server, and configuration settings in the TeamCity Data Directory.

**Description:**

Show advanced options

**Create**

- Next TeamCity offers to create and attach a new VCS Root: to be able to create a build, TeamCity has to know where the source code resides, and a **VCS root** is a collection of VCS settings (paths to sources, login, password, and other settings) that defines how TeamCity communicates with a version control (SCM) system to monitor changes and get sources for a build. Each build configuration has to have at least one VCS root attached to it.  
A VCS root can be created **automatically** or manually. To create it manually, select the type of VCS from the drop-down (Git in the example below), specify the required information (name and URL), test your connection and click Create.

## New VCS Root

Type of VCS

Type of VCS: Git

VCS Root

VCS root name: \* Git  
A unique name to distinguish this VCS root from other roots.

VCS root ID: \* SampleProject\_Git  
VCS root ID must be unique across all VCS roots. VCS root ID can be used in parameter references to VCS root parameters and REST API.

General Settings

Fetch URL: \* https://github.com/Julia-Alexandrova/Sample-Project  
It is used for fetching data from the repository.

Push URL: It is used for pushing tags to the remote repository. If blank, the fetch url is used.

Default branch: \* refs/heads/master  
The main branch or tag to be monitored

Authentication Settings

Authentication method: Test Connection  
Connection successful!

Show advanced options

Create Test connection Cancel

**i** If your project resides in several version control systems, you can attach as many VCS Roots to it as you need. For example, if you store a part of your project in Perforce, and the rest in Git, you need to create and attach 2 VCS roots - one for Perforce, another for Git. Learn more about configuring VCS roots.

After you have created a VCS root, you can instruct TeamCity to exclude some directories from checkout, or map some paths (copy directories and all their contents) to a location on the build agent different from the default one. This can be done by means of [Checkout rules](#). You can also specify whether you want TeamCity to checkout the sources [on the agent or server](#). Note that the agent-side checkout is [supported not for all VCSs](#), and in case you want to use it, you need to have a version control client installed on at least one agent.

After the VCS Root is created, the build configuration settings are displayed on the left.

- Now you can configure [Build steps](#) by selecting the corresponding setting on the left. You can either instruct TeamCity to automatically detect the build steps after scanning your repository or configure build steps manually as described in this example.

Click Add build step and select a build runner from the drop-down.

The screenshot shows the 'New Build Step' configuration screen. On the left, there's a sidebar with various settings like General Settings, Version Control Settings, and Build Steps. Under 'Build Steps', 'Build Step: Maven' is selected. The main area shows a dropdown menu titled 'Runner type:' with many options listed, including Docker, Docker Compose, Gradle, and Maven. The 'Maven' option is currently selected and highlighted in blue.

5. Fill in the required fields and save the build step
6. Congratulations! You've configured your first build containing one build step. Now you can [run](#) your build and [tweak](#) its [settings](#) if necessary.

## Run your first build

Your build currently has one build step and you can now launch your first build by clicking Run in the upper right corner of the TeamCity web UI:

The screenshot shows the 'Build' configuration page for a project named 'Sample Project'. The 'Build Step: Maven' section is selected. A yellow banner at the top says 'Build step settings updated.' Below it, there's a table with one row: '1. Maven' with 'Path to POM: pom.xml', 'Goals: clean test', and 'Execute: If all previous steps finished successfully'. There are buttons for 'Add build step' and 'Auto-detect build steps'.

You will be redirected to the build result page, where you can watch the build progress and review its results upon the build finishing. You can also access your build configuration settings from this page and edit them as required:

The screenshot shows the 'Build (1)' result page. It displays a summary table with the following data:

Result:	4 tests passed: 4, ignored: 13	Agent:	munit-048
Time:	20 Jun 18 17:40:15 - 17:40:27 (12s)	Triggered by:	you on 20 Jun 18 17:40

Below the table, there are sections for '13 tests ignored' and '4 tests passed (all tests)'. An info icon in the bottom right corner provides tips about adding and reordering build steps.

**i** You can add as many steps as you like and reorder them if required. You can add steps manually or ask TeamCity to detect them automatically. TeamCity will also suggest settings, such as triggers, failure conditions, and build features. Depending on the build configuration settings, it may prompt some additional options. You can follow the suggestions and add the settings to configure your build.

You can always tweak the settings after running your first build.

## Tweak your build configuration settings

You might want to configure the following settings:

- artifacts
- automatic build trigger
- build number

## Artifacts

If your build produces installers, WAR files, reports, log files, etc. and you want them to be published on the TeamCity server after finishing the build, you can specify the paths to such artifacts in the General Settings section of Build configuration settings.

As you have a finished build, the build agent has checked out the sources already and the Artifact paths field has the checkout directory browser. You can select artifacts from the tree:

The screenshot shows the 'Artifact paths' configuration dialog. It includes fields for 'Name' (Build), 'Build configuration ID' (SampleProject\_Build), and 'Description'. The 'Artifact paths' field contains the path 'target\surefire-reports'. A tooltip explains the syntax: 'newline- or comma-separated paths in the form of [+:]source [> target] to include as build artifacts. Ant-style wildcards are supported, e.g. use \*\*/\* > target\_directory1 into the target\_directory.' Below the path, a tree view shows the directory structure: 'target' containing 'classes', 'surefire', 'test-classes', and 'project'. The 'surefire' folder is expanded, showing 'test-classes' and 'project' with their respective sizes: 264 B and 388 B. There are also 'pom.xml' and 'classpath' entries. Buttons for 'Save' and 'Cancel' are at the bottom.

TeamCity will place the paths to them into the input field and you can modify them as needed:

This screenshot shows the same configuration dialog as above, but the 'Artifact paths' field now contains the modified path 'target\surefire-reports => surefire-reports.zip'. The rest of the interface is identical to the previous screenshot.

Save your settings. Now when we run a build, TeamCity will put the required reports into an archive.

The build configuration home page lists all builds that were run and enables you to view the available artifacts:

This screenshot shows the build configuration home page for 'Sample Project'. It displays the 'Recent history' section with two builds: '#2' and '#1'. Build '#2' is shown with a green checkmark and the message 'Tests passed: 4, ignored: 13'. It was run on '20 Jun 18 18:12' and has 'No changes'. The 'Artifacts' section shows a single artifact: 'surefire-reports.zip' (15.36 KB). Build '#1' is also listed with similar details. Navigation links like 'Overview', 'History', 'Change Log', 'Issue Log', 'Statistics', 'Compatible Agents', 'Pending Changes', 'Settings', and 'Maven' are visible at the top and bottom of the page.

You can also view and download artifacts from the build results page:

This screenshot shows the build results page for build '#2' (20 Jun 18 18:12). The 'Artifacts' tab is selected, displaying the artifact 'surefire-reports.zip' (15.36 KB). Other tabs like 'Overview', 'Changes', 'Tests', 'Build Log', 'Parameters', and 'Maven Build Info' are also present. At the bottom, there are links for 'All history' and 'Last recorded build'.

More details are available in the dedicated section of our documentation.

## Automatic Build Trigger

Automatic build triggering on detecting a change in the version control is essential to any CI. TeamCity will add a [VCS trigger](#) automatically when creating a project / build configuration from a URL or repository. You can also do it manually using the [Edit Build Configuration Settings](#) drop-down, [Triggers](#) page:

The screenshot shows the 'Build Configuration Home' page for a 'Sample Project'. In the 'Triggers' section, a modal window titled 'Add New Trigger' is open. The dropdown menu lists several trigger types: 'Please choose a trigger', 'VCS Trigger' (which is selected and highlighted in blue), 'Schedule Trigger', 'Finish Build Trigger', 'Branch Remote Run Trigger', 'Maven Artifact Dependency Trigger', 'Maven Snapshot Dependencies Trigger', 'NuGet Dependency Trigger', and 'Retry Build Trigger'.

## Build Number Format

Each build in TeamCity has a build number, which is a string identifier composed according to the pattern specified on the General settings page of your build configuration (the field is available on clicking the show advanced options link). You can leave the default value here, in which case the build number format will be maintained by TeamCity and will be resolved into a next integer value on each new build start. More details are available in the [dedicated section](#) of our documentation.

Happy building!

## Concepts

This section lists basic TeamCity terms and their definitions, that will help you successfully start and work with TeamCity:

- [Agent Cloud Profile](#)
- [Agent Home Directory](#)
- [Agent Requirements](#)
- [Agent Work Directory](#)
- [Already Fixed In](#)
- [Authentication Modules](#)
- [Build Agent](#)
- [Build Artifact](#)
- [Build Chain](#)
- [Build Checkout Directory](#)
- [Build Configuration](#)
- [Build Configuration Template](#)
- [Build Grid](#)
- [Build History](#)
- [Build Log](#)
- [Build Number](#)
- [Build Queue](#)
- [Build Runner](#)
- [Build State](#)
- [Build Tag](#)
- [Build Working Directory](#)
- [Change](#)
- [Change State](#)
- [Clean Checkout](#)
- [Clean-Up](#)
- [Code Coverage](#)
- [Code Duplicates](#)
- [Code Inspection](#)
- [Composite Build Configuration](#)
- [Continuous Integration](#)
- [Dependent Build](#)
- [Deployment Build Configuration](#)
- [Difference Viewer](#)

- Favorite Build
- First Failure
- Guest User
- History Build
- Identifier
- Mapping TeamCity Concepts to Other CI Terms
- Notifier
- Personal Build
- Pinned Build
- Pre-Tested (Delayed) Commit
- Project
- Remote Debug
- Remote Run
- Revision
- Role and Permission
- Run Configuration Policy
- Super User
- TeamCity Data Directory
- TeamCity Home Directory
- TeamCity Specific Directories
- User Account
- User Group
- VCS root
- Wildcards

## Build Artifact

TeamCity contains an integrated lightweight builds artifact repository. The artifacts are stored either on the [server-accessible file system](#) or on an [external storage](#).

Build artifacts are files produced by a build. Typically these include distribution packages, WAR files, reports, log files, etc. When creating a build configuration, you specify the paths to the artifacts of your build on the [General Settings](#) page.

Upon the build finish, TeamCity searches for artifacts in the build [checkout directory](#) according to the specified [artifact path](#) or [path patterns](#). The matching files are then uploaded ("published") to the TeamCity server, where they become available for download through the web UI or can be used in other builds using [artifact dependencies](#).

To download artifacts of a build, use the artifacts icon  on the project or build configuration overview page, on the TeamCity pages that list the builds, or see at the [Artifacts](#) tab of the build results page. You can browse artifacts inside archives is supported.

You can automate artifacts downloading as described in the [Patterns For Accessing Build Artifacts](#) section.

In case of the built-in storage, TeamCity keeps artifacts on the disk in a directory structure that can be accessed directly (for example, by configuring the Operating System to share the directory over the network). The storage format is described in the [TeamCity Data Directory#artifacts](#) section. The artifacts are stored on the server "as is" without additional compression, etc. By default, the artifacts are stored under the `<TeamCity data directory>/system/artifacts` directory which [can be changed](#).

Build artifacts can also be uploaded to the server while the build is still running. To instruct TeamCity to upload the artifacts, the build script should be modified to send [service messages](#).

## Hidden Artifacts

In addition to user-defined artifacts, TeamCity also generates and publishes some artifacts for internal purposes. These are called [hidden artifacts](#).

For example, for Maven builds, TeamCity creates the `maven-build-info.xml` file that contains Maven-specific data collected during the build. The content of the file is then used to visualize the Maven data on the [Maven Build Info](#) tab in the build results.

- Hidden artifacts are placed under the `.teamcity` directory in the root of the build artifacts.
- Hidden artifacts are not listed on the Artifacts tab of the build results by default. However, below the list of the artifacts there's a link that allows you to view hidden artifacts if any. When hidden artifacts are displayed, clicking the Download all link will result in downloading all artifacts including hidden ones.
- Artifacts dependencies do not download hidden artifacts unless they explicitly have `".teamcity"` in the pattern.
- Hidden artifacts are not deleted by artifacts clean-up unless `".teamcity"` is explicitly specified in the pattern.

You can configure publishing some builds artifacts under `.teamcity` directory to make them hidden.

Some of the hidden artifacts are:

- `maven-build-info.xml.gz` - Maven build data. Used to display data on [Maven Build Info](#) build's tab.

- properties directory - holds properties calculated for the build on the agent. There are properties actual before the build and after the build. These are displayed on the build Properties tab.
- .NETCoverage - raw .Net coverage data (e.g. used to open dotCover data in VS addin)
- coverage\_idea - raw IntelliJ IDEA coverage data (e.g. used to open coverage in IDEA)

## Artifacts Cache on Agent

All artifacts published by a build are stored in the agent's artifacts cache in the `<Build Agent home>\system\artifacts_cache` directory, which helps speed up artifact dependencies in some cases. However, depending on the size of artifacts, **clean-up** and other settings, artifacts caching may cause low disk space on the agent. You can **configure** storing published artifacts in the agent cache.

See also:

<a href="#">Concepts: Dependent Build</a>
<a href="#">Administrator's Guide: Configuring General Settings</a>   <a href="#">Configuring Dependencies</a>   <a href="#">Patterns For Accessing Build Artifacts</a>

# Build Configuration

A Build Configuration is a collection of settings used to start a build and group the sequence of the builds in the UI. Examples of build configurations are distribution, integration tests, prepare release distribution, "nightly" build.

A build configuration belongs to a **project** and contains builds. You can explore details of a build configuration on its [home page](#) and modify its settings on the [editing page](#).

It is recommended to have a separate build configuration for each sequence of builds (that is performing a specified task in a dedicated environment). This allows for proper features functioning, like detection of new problems/failed tests, first failed in/fixed in tests status, automatically removed investigations, etc.

To tackle an increased number of build configurations you can use [Build Configuration Templates](#) and project-level [parameters](#).

In this section:

- [Build Configuration Settings](#)
- [Build Configuration Types](#)
- [Build Configuration State](#)
  - Pausing / Activating several build configurations of a project
  - Pausing / Activating a single build configuration
- [Build Configuration Status](#)
  - Status Display for Set of Build Configurations

## Build Configuration Settings

Build configuration settings include:

- [General settings](#)
- [Version control settings](#), defining how the source code is retrieved from VCS, where it is checked out to, etc.
- [Build steps](#), i.e. actions that are run sequentially: e.g. running msbuild, a script, unit tests, etc.
- [Triggers](#), which are rules defining when to start a new build
- [Failure conditions](#) specifying when a build will be marked as failed
- [Additional build features](#)
- [Dependencies](#):

- for [snapshot dependencies](#), TeamCity will run all dependent builds on the sources taken at the moment the build they depend on starts
- For [artifact dependencies](#), before a build is started, all artifacts this build depends on will be downloaded and placed in their configured target locations and then will be used by the build.
- [Parameters](#) which allow sharing settings
- Agent requirements specifying whether a [build configuration](#) can run on a particular [build agent](#).



Build configuration settings and build behavior may vary depending on the type of build configuration.

## Build Configuration Types

The following build configuration types exist in TeamCity:

- regular build configuration, defining actions and rules to apply to the source code. All the settings above are applicable.
- [deployment build configuration](#), which deploys artifacts of other builds to some environment (since TeamCity 2017.2)
- [composite build configuration](#), which aggregates results from several other builds combined by snapshot dependencies and presents them in a single place (since TeamCity 2017.2)

## Build Configuration State

A build configuration is characterized by its state visible in the UI which can be paused or active. By default, when created all configurations are active and can be paused manually as described below or automatically if the project is [archived](#).

If a build configuration is paused, its [automatic build triggers](#) are disabled until the configuration is activated. Still, you can start a build of a paused configuration manually or automatically as a part of a [Build Chain](#). Besides, information on paused build configurations is not displayed on the [Changes](#) page.

Since TeamCity 10.0, it is possible to manually pause all or selected build configurations for a project.

### Pausing / Activating several build configurations of a project

To pause several build configurations of a project, do the following:

On the Project Settings page:

1. Click the Actions button, select the Pause/Activate triggers.
2. In the dialog that opens select the box next to the project to pause all its build configurations or check the boxes of the configurations selectively: the checked build configurations will be paused.
3. Add an optional comment
4. To remove the builds of the paused configurations from the [build queue](#) en masse, check the Cancel already queued builds if build configuration is paused box.
5. Click Apply.

To activate several build configurations of a project, do the following:

On the Project Settings page:

1. Click the Actions button, select the Pause/Activate triggers.
2. In the dialog that opens clear the box next to the project to activate all its build configurations or clear the boxes of the configurations selectively: the unselected build configurations will be activated.
3. Add an optional comment
4. Click Apply.

### Pausing / Activating a single build configuration

To pause or activate an individual build configuration, do one of the following:

- On the Build Configuration Settings or Home Page page: click the Actions button, select the Pause triggers..., add your comment (optional) and click Pause. For a paused configuration, click the Activate button at the top of the settings page. To remove the builds of the paused configuration from the [build queue](#), check the Cancel already queued builds box.

## Build Configuration Status

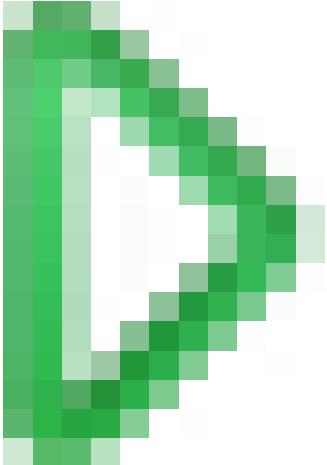
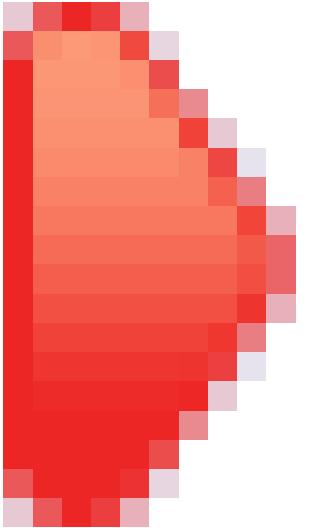
In general, a build configuration status reflects the status of its last finished build.

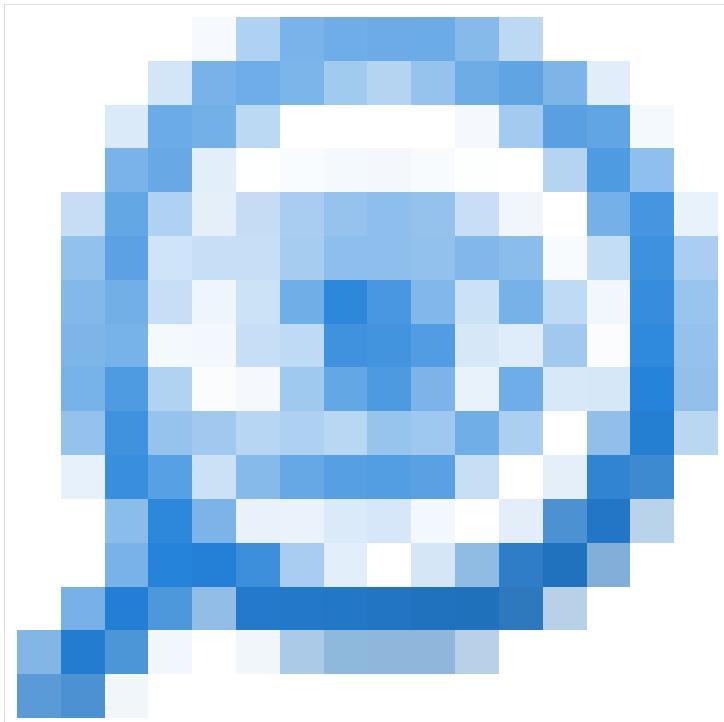


Personal builds do not affect the build configuration status.

You can view the status of all build configurations for all/particular project on the Projects Overview page or Project Home Page, when the details are collapsed.

Build configuration status icons:

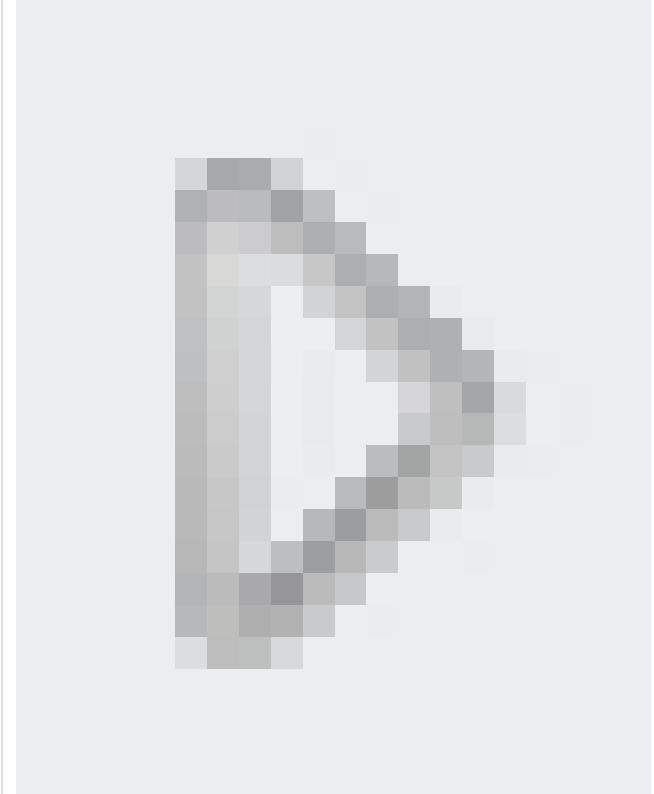
Icon	Description
	The last build on default branch executed successfully.
	The last build on default branch executed with errors or one of the currently running builds is failing. The build configuration status will change to "failed" when there's at least one current running and failing build, even if the last finished build was successful.



Indicates that someone has started investigating the problem already fixed it. (see [Investigating and Muting Build Problem](#))

no icon

There were no finished builds for this configuration, the status unknown. If none of the build configurations in a project have finished builds, the



is displayed next to a project name



The build configuration is paused; no builds are triggered for it. Click on the link next to the status to view by whom it was paused, and activate configuration if needed.

## Status Display for Set of Build Configurations

It is possible to filter out the build configurations whose status you want to be displayed in TeamCity or externally.

To display the status of selected build configurations in TeamCity:

- configure visible projects on the Projects Overview page to display the status of build configurations belonging to these projects only
- implement a custom Java plugin for TeamCity to make the page available as a part of the TeamCity web application

To display the status for a set of build configurations externally (e.g. on your company's website, wiki, Confluence or any other web page), you can:

- use the external status widget
- use the build status icon
- use any of the available visualization plugins
- implement a separate page or application which will get the build configuration status via the TeamCity REST API

See also:

[Administrator's Guide: Creating and Editing Build Configurations](#)

## Composite Build Configuration

TeamCity provides the Composite type of build configuration.

The purpose of the composite build configuration is to aggregate results from several other builds combined by snapshot dependencies and present them in a single place. To some extent, a composite build can be viewed as a build which consists of several parts which can be executed in parallel on different agents. All these parts will have a synchronized snapshot of the source code and the results can be seen in a single place.

When [creating a build configuration](#), you can specify Composite as its type and choose the build configurations whose builds results the current composite one will aggregate.

There are important differences between composite builds and regular builds with snapshot dependencies:

- A composite build does not occupy an agent; as a result, some settings which require an agent (e.g. build steps or requirements) are not applicable
- A composite build often acts as a single build regardless of the number of dependencies:
  - A composite build is shown as running at the time when the first dependency of the build chain starts, and is shown as finished only when the last dependency finishes
  - If a composite build is stopped or removed from the queue, all its dependencies, i.e. the whole build chain, is stopped / removed too
  - It is possible to limit the number of running composite builds, but in case with a composite build it will affect the build and all its dependencies: if the limit is set to 1 and a composite build is already running, another composite build with all its dependencies will wait in the queue until the current one finishes.
  - A composite build aggregates results from dependencies, e.g. it aggregates all of the tests and shows all failed/muted or ignored tests of the chain in a single place, the same applies to code coverage or results of code inspection/code duplicates analysis
  - The status line of the composite builds reflects the current chain state: the number of running/queued/finished or failed dependencies
  - The progress indicator of the composite build reflects all of the dependencies, so it actually shows when the whole chain is going to finish

- A composite build does not have its own artifacts. However, you can make it display the artifacts published by its parts (dependencies) via artifact dependencies.

## Clean-up of Composite Build Artifacts

Composite builds only display the artifacts published by its parts via artifact dependencies, which means composite builds do not have their own artifacts (except for internal artifacts). To enforce a certain clean-up policy for artifacts of a composite build, you need to have the same clean-up rules for all its parts.

## Deployment Build Configuration

TeamCity provides the Deployment type of build configuration. Build configurations which perform deploying to some environment can be marked with this type: these are usually build configurations which have snapshot or artifact dependencies on the builds, whose results they deploy.

You can [create a build configuration](#) and use the [General Settings](#) tab to set its type to Deployment.

Once a configuration is marked as Deployment, TeamCity changes behavior in the following way:

- The Run button caption for this configuration changes to Deploy
- If there is a build used as a dependency in a Deployment configuration, then the Deployments section will be shown on the build results page of this build, allowing you to quickly deploy the build.
- The Change status page (the page where you can see which configurations are affected by the change, and which builds have been executed with this change) has the Deployments tab that shows builds in Deployment configurations where this change was deployed for the first time.
- For Deployment configurations, TeamCity always shows the latest started build regardless of the changes it contains; unlike regular build configurations, for which the build with the latest changes is displayed.
- When setting the build configuration type to "Deployment", several settings are automatically changed to reflect the best practices: "Limit the number of simultaneously running builds" is set to "1" and "allow triggering personal builds" option is turned off.

## Build Configuration Template

On this page:

- [Overview](#)
- [Creating build configuration template](#)
- [Defining default template for project](#)
  - [Related settings changes](#)
- [Associating build configurations with templates](#)
  - [Associating build configuration with multiple templates](#)
    - [Related settings changes](#)
- [Detaching build configurations from template](#)
- [Redefining settings inherited from template](#)
  - [Using parameter reference](#)
    - [Example of configuration parameters usage](#)
  - [Ability to have pre- and post-steps in a template](#)
- [Enforcing settings inherited from template](#)

### Overview

Build Configuration Templates allow you to eliminate duplication of build configuration settings. If you want to have several similar (not necessarily identical) build configurations and be able to modify their common settings in one place without having to edit each configuration, create a build configuration template with those settings. Modifying template settings affects all build configurations associated with this template.

It is possible to define a default template in a project for all build configurations in this project and its subprojects. The [section below](#) contains details.

Build configuration templates support project hierarchy: once created, available templates include the ones from the current project and its parents. On copying a project or a build configuration, the templates that do not belong to the target project or one of its parents are automatically copied. You can associate a build configuration with a template only if the template belongs to the current project or one of its parents.

### Creating build configuration template

There are several ways to create a build configuration template:

- Manually, like a [regular build configuration](#).
- Extract from an existing build configuration: there is the Extract template option available from the Actions button at

the top right corner of the screen. Note that if you extract a template from a build configuration, the original configuration automatically becomes [associated](#) with the newly created template.

## Defining default template for project

Default templates allow affecting all build configurations in this project and its subprojects.

You can associate all the build configurations of the project with a default template using the General Settings page of the project administration, by selecting a template from the Default template drop-down. The option is available if at least one template is defined in the project or its parent. All new build configurations will inherit the default template settings.

The settings of the existing configurations will be preserved.

When defined, the default template affects all build configurations and subprojects of this project unless other default templates are defined in subprojects. With default templates, you can easily modify all project's build configurations, for example:

- add a specific build feature to all build configurations of a project,
- switch all build configurations to some specific checkout mode,
- provide a default failure condition.

## Related settings changes

The project configuration schema accommodates for default templates.

- XML: The project-config.xml file contains the `<default-template ref="...." />` element.
- DSL: Project configuration contains the `defaultTemplate = "..."` method. See a sample project configuration with a default template configured:

```
object Project : Project({  
    uuid = "2b241ffb-9019-4e60-9a3a-d5475ab1f312"  
    extId = "ExampleProject"  
    parentId = "_Root"  
    name = "Example Project"  
    defaultTemplate = "ExampleProject_MyDefaultTemplate"  
    ...  
    features {  
        ...  
    }  
    ...  
})
```

## Associating build configurations with templates

- You can [create new build configurations based on a template](#)
- You can associate/attach any number of existing build configurations with/to a template: there's the Attach to template... option available from the Actions button at the top right corner of the screen.

 When you associate an existing build configuration with a template, the build configuration inherits all the settings defined in the template, and if there's a conflict, the template settings supersede the settings of the build configuration (except dependencies, parameters, and requirements). The settings inherited from a template can be overridden.

You can associate a build configuration to a template only if the template belongs to the current project or one of its parents. A template which has at least one associated build configuration cannot be deleted, the associated build configurations need to be detached first.

It is also possible to associate a build configuration with multiple templates.

## Associating build configuration with multiple templates

A build configuration can be attached to multiple templates using the "Attach to template..." action menu. In the Actions menu, the Manage templates action allows users to:

- change the order of templates, which affects the overlapping settings priority and the build step order: the priority is

given to the settings from the template higher in the list. It affects such entities as parameter names, setting ids (for build steps, triggers, features, artifact dependencies and requirements), VCS roots or snapshot dependency source build configurations ids if they overlap between templates attached to a build configuration.

- detach the build configuration from some of the templates (the user marks those to be detached and then has to apply their changes)
- detach the build configuration from all templates using correspondingly named button on the same dialog window

You can view all the templates attached to a build configuration on the Build Configuration Settings page.

The settings from all templates the build configuration is attached to are inherited and you can view where they are inherited from in the view/edit Build Configuration Settings pages.

When a build configuration is detached from some of its templates, all the effective (i.e. not overridden in the config and not overlapped by some higher priority template) settings inherited from them are copied to the configuration. The copying build configuration logic is the same as extracting a template. On moving a build configuration/project, the logic checks all templates to which the build configuration is attached.

### Related settings changes

- XML: If a build configuration is attached to a single template, the resulting config XML format stays the same as it was before ("ref" attribute of the "settings" element). If it is attached to a number of templates, then references to them are stored in a separate element under "settings" node, as follows:

```
<inherits>
<ref id="Template1_ExternalId" />
<ref id="Template2_ExternalId" />
.....
</inherits>
```

- DSL: Kotlin DSL is extended, so within a build type definition users can use the `templates(vararg)` method accepting either external ids or DSL template instances (but not a mix of them, so if both templates defined inside and outside DSL are used in the same configuration, the external ids of both must be used). The older `template(...)` method and property cannot be used multiple times within the same build type definition to indicate that it is inherited from multiple templates - following earlier implementation, each time this method is used, it overrides the previous template external id. It is preserved for backward compatibility.

### Detaching build configurations from template

When you detach a build configuration from a template using the Detach from template option available from the Actions button at the top right corner of the build configuration settings screen, all settings from the template will be copied to the build configuration and enabled for editing.

### Redefining settings inherited from template

A build configuration associated with a template inherits all its settings (marked as inherited in the UI). Inherited settings cannot be deleted from an associated build configuration, but the inherited build steps, triggers, build features, failure conditions, artifact dependencies, and agent requirements can be disabled in it.

Modifying settings in the template will influence all configurations associated with this template; however, it is possible to redefine most settings in an associated build configuration.

Since TeamCity 2018.1, it is possible to redefine all build configuration settings (e.g. build steps, [parameters](#), [build options](#), etc.) the only exceptions being snapshot dependencies and checkout rules which cannot be redefined.

 Modified settings are highlighted with a yellow border and the Reset button appears on the right of the modified settings enabling you to revert the changes to the original settings of the template.

 Note that if you redefine an inherited parameter in a build configuration, it will be saved with the inherited name and the new value. If you then rename this parameter in the template, you will end up with two parameters in the build configuration: the one with the original name and the redefined value, and the renamed one inherited from the template.

### Using parameter reference

Beside redefining settings as described above, you can redefine individual fields of the inherited settings via parameter references.

To introduce a configuration parameter reference, use the `%ParameterName%` syntax in the template text field. Once introduced, this parameter appears on the Parameters page of the build configuration template marked as requiring a value. You can either specify the parameter's default value or leave it without any value. You can then define the actual value for the parameter in a build configuration associated with the template.

See also [Configuring Build Parameters](#).

### Example of configuration parameters usage

Assume that you have two similar build configurations that differ only by checkout rules. For instance, checkout rules for the first configuration should contain `'+:release_1_0 => .'` and for the second one `'+:trunk => .'`. All other settings are equal. It would be useful to have one template to associate with both build configurations, but this means changing the checkout rules in each build configuration separately.

To do so, perform the following steps:

1. Extract a template from one of those configurations.
2. In the template settings, navigate to Version Control Settings, open the Checkout rules dialog for the VCS root and enter there: `%checkout.rules%`
3. For the inherited build configuration, open the configuration settings page and on the Parameters page specify the actual value for the `checkout.rules` configuration parameter.
4. For the second build configuration, use the Associate with template option from Actions and choose the template. Specify an appropriate value for the `checkout.rules` parameter right in the "Associate with Template" dialog. Click "Associate".

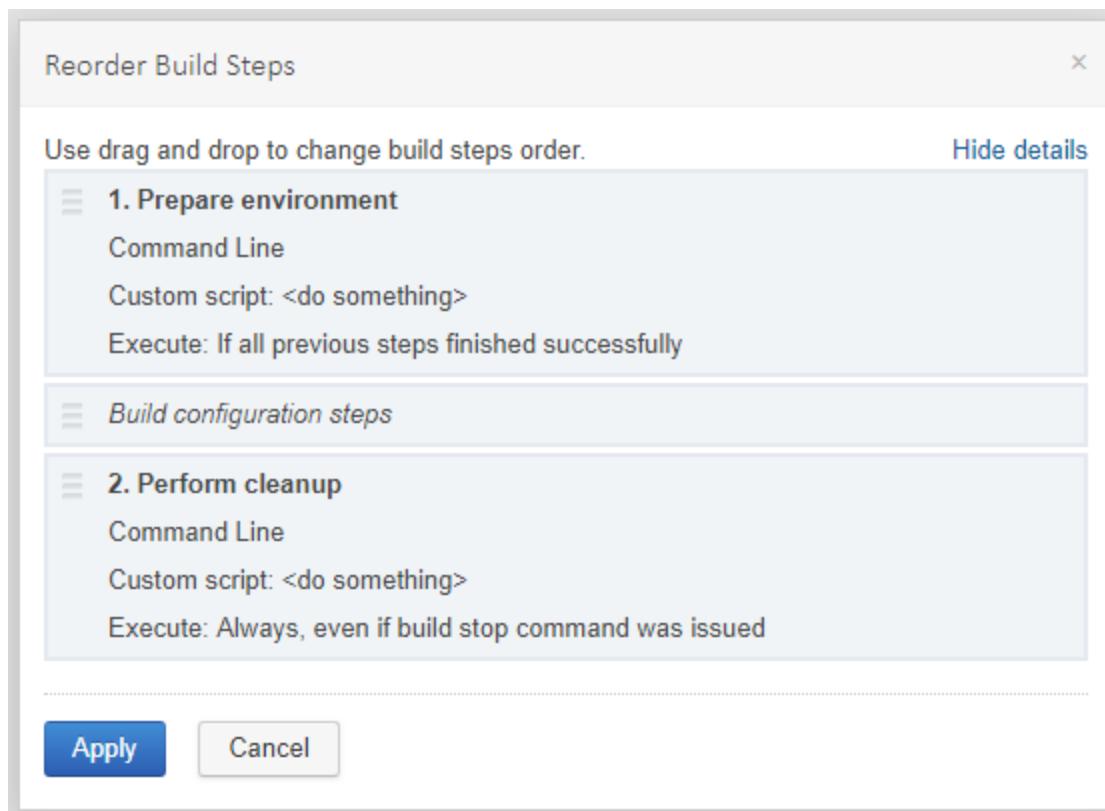
As a result, you'll have two build configurations with different checkout rules, but associated with one template.

This way you can create a configuration parameter and then reference it from any build configuration, which has a text field.

### Ability to have pre- and post-steps in a template

There is sometimes a need to define a common build step in a template, so that this step will be executed either before all build configuration steps or after them.

Since TeamCity 2018.1, for a given template it is possible to define such steps and then define their placement in respect to the build configuration steps. All build configuration steps are represented as a placeholder in the Reorder Build Steps dialog. The template steps can be placed before and/or after this placeholder.



You can still customise the order of build steps in a template-based build configuration if necessary:

- using the TeamCity Web UI, it is possible to change the placement of the build configuration steps in respect to the template steps
- using [versioned settings](#), it is possible to change not only the placement of the build configuration steps in respect to the template steps, but also to reorder the steps of the template itself.

## Enforcing settings inherited from template

Since TeamCity 2018.1, if you want to enforce some settings on all the build configurations in the project so that other users could not redefine them, TeamCity provides this ability for all of the build configurations in a project hierarchy. For instance, using enforced settings it is possible to set [agent side checkout](#) everywhere, or make sure that all build configurations have some strict [execution timeout](#). Currently it is possible to enforce build features, options, and parameters.

To enforce some settings in the project hierarchy, create a template with these settings. After that, a system administrator can set this template as the enforced settings template in the project:

The screenshot shows the 'Edit Project' dialog for a project named 'Trunk'. In the 'Enforced settings template:' dropdown, the option '<No enforced settings template selected>' is highlighted. A tooltip below the dropdown states: 'The template is attached to all build configurations in this project and its subprojects unless redefined in a subproject.' Below the dropdown, a list of available templates is visible, including 'TeamCity', 'Vagrant Environment', 'Trunk', 'Default Template', 'DistMiddleStep', 'dotnet-listeners tests', and 'JPS Template'. At the bottom left are 'Save' and 'Cancel' buttons.

The enforced settings template is similar to the default template as all of its settings are inherited in build configurations of the project hierarchy. The difference is that these inherited settings cannot be disabled or overridden.

The system administrator role is required to associate a project with a specific enforced settings template. The template itself can be edited by a project administrator who can administer the project where the template is defined.

If the enforced settings template is specified in a project and a different template is assigned as the enforced settings in a subproject, the template of the subproject will have a higher priority.

## See also:

[Administrator's Guide: Creating and Editing Build Configurations | Configuring Build Parameters](#)

## Build Log

A build log is an enhanced console output of a build. It is represented by a structured list of the events which took place during the build. Generally, it includes entries on TeamCity-performed actions and the output of the processes launched during the build.

TeamCity captures the processes output and stores it in an internal format that allows for hierarchical display.

On this page:

- [Viewing Build Log](#)
- [Build Log Size](#)
  - [Partial Build Log Display](#)
- [ANSI-style Coloring in Build Log](#)

### Viewing Build Log

The log of a specific build is available for browsing at the [Build Results page](#).

The Tree view is the most capable view provided in the web UI.

By default, all messages are displayed. Using the View drop-down, you can switch from all messages to viewing errors separately, or you can choose Important messages to see the log filtered by "error" and "warning" statuses. You can also use the "Verbose" view level and download a raw build log using the corresponding link.

Since TeamCity 10.0, it is possible to enable the dark theme in the build log by selecting the Use console view check-box.

You can download a full build log in the textual form or as a .zip archive from the Build Results page by clicking



. Alternatively, you can use the following URL:

<http://teamcity:8111/httpAuth/downloadBuildLog.html?buildId=...>.

It is also possible to download the build log as a .zip file using the corresponding link in the UI or via the following URL:

<http://teamcity:8111/httpAuth/downloadBuildLog.html?buildId=&archived=true>.

## Build Log Size

It is recommended to keep the build log small and tune build scripts not to print too much into the output. Large build logs are hard to view in the browser and are loading TeamCity infrastructure piping build messages from the agent to the server while the build is running.

It is recommended to print into the output only the messages required to understand the build progress and build failures. The rest of the information should be streamed into a log file and the file should be published as a build artifact. A "good" build log size is megabytes at most.

## Partial Build Log Display

Since TeamCity 2017.1, when opening large build logs, TeamCity displays a part of it to avoid browser hanging. You can view the full build log on clicking the corresponding link.

The display threshold is set to 7M characters by default and can be adjusted using the `teamcity.buildLog.sizeThreshold.chars` internal property (not applicable to running build logs and logs links with states (e.g. direct links to messages)).

## ANSI-style Coloring in Build Log

TeamCity build logs render clickable hyperlinks and support ANSI-style escape color codes by default: if your tool produces a colored console output, you will see in the build log in TeamCity. See the related feature request for the [full list of supported sequences](#).

To disable the coloring, set the `teamcity.buildLog.ansiColoring.enabled=false` internal property.

# Build Queue

The build queue is a list of builds that were triggered and are waiting to be started. TeamCity will distribute them to compatible build agents as soon as the agents become idle. A queued build is assigned to an agent at the moment when it is started on the agent; no pre-assignment is made while the build is waiting in the build queue.

When a build is triggered, first it is placed into the build queue, and, when a compatible agent becomes idle, TeamCity will run the build. On this page:

- [Build Queue Optimization by TeamCity](#)
- [Build Queue Tab](#)
  - [Agent Selection for Queued Build](#)
  - [Ordering Build Queue](#)
  - [Pausing/Resuming Build Queue](#)

## Build Queue Optimization by TeamCity

By default, TeamCity optimizes the build queue as follows:

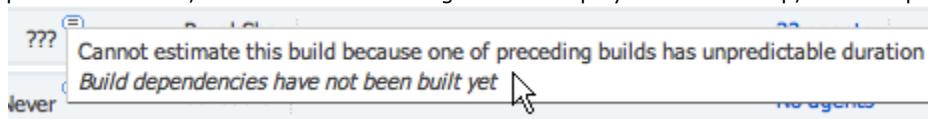
- if a similar build exists in the queue, a new build (on the same change set and with the same custom properties) will not be added
- if an automatically triggered build chain has more changes than a build chain that is already queued, the latter will be replaced with the automatically triggered build chain, if such replacement will not delay obtaining the build chain results (based on the [estimated duration](#))
- while a build chain is in the queue, TeamCity tries to replace the queued builds with equivalent started builds.

This default behavior can be manually disabled via the corresponding option in the [VCS Build Trigger](#) and [Schedule Build Trigger](#).

## Build Queue Tab

The list of builds waiting to be run can be viewed on the Build Queue tab. This tab displays the following information:

- The number of the build in the queue which is a link to the [build results](#) page
- The branch (if available)
- The build configuration name in the following format: <project name>::<build configuration name>, where the project and build configuration names are the links to the corresponding overview pages;
- Time to start: the estimated wait duration. Hovering the mouse cursor over the estimated time value shows a tooltip with the following information:
  - the expected start/finish time,
  - the link to the planned agent page.
  - If the current build is a part of a build chain and the builds it depends on are not finished yet, a corresponding note will be displayed. For some builds, like the builds that have never been run before, TeamCity can't estimate possible duration, so the relevant message will be displayed in the tooltip, for example:



- Triggered by - a brief description of [the event that triggered the build](#).
- Can run on - the number of agents compatible with this build configuration. You can click an agent's name link to open the [Agents page](#), or use the down arrow to quickly view the list of compatible agents in the pop-up window.

## Agent Selection for Queued Build

When there are several idle agents that can run a queued build, TeamCity tries to select the fastest one as follows:

1. If no builds have previously run on agents, the [CPU rank](#) is used to select an agent.
2. If builds have previously run on agents, the estimated build duration for the given build configuration is used to select an agent. The estimate is made based on the heuristics of the latest builds in the history of the build configuration; for estimating, the execution time of the more recent builds has more weight than that of the earlier builds. [Personal](#) and [Canceled](#) builds are not taken into account, neither are any individual builds whose duration differs significantly from the rest of the builds for this build configuration.

## Ordering Build Queue

You can do the following:

- [reorder](#) the builds in the queue manually
- [remove](#) build configurations or personal builds from the queue
- If you have System Administrator permissions, you can [assign different priorities to build configurations](#), which will affect their position in the queue.

## Pausing/Resuming Build Queue

The build queue can be paused manually or automatically.

Users with [Enable / disable agent](#) permission (included in the [Agent Manager](#) role by default) can manually Pause/Resume the Build Queue (since pausing the queue is equivalent to disabling all the agents on the server).

The build queue can be paused automatically if the TeamCity Server runs out of disk space. The queue will be automatically resumed when sufficient space is available.

When the queue is paused, every page in TeamCity will contain a message containing information on the reasons for pausing.

See also:

[Concepts: Build Chain](#)  
[Administrator's Guide: Ordering Build Queue](#)

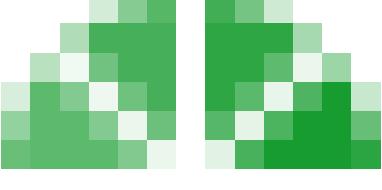
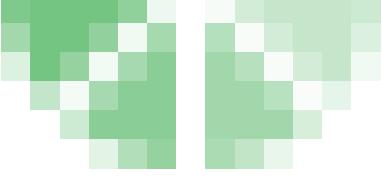
## Build State

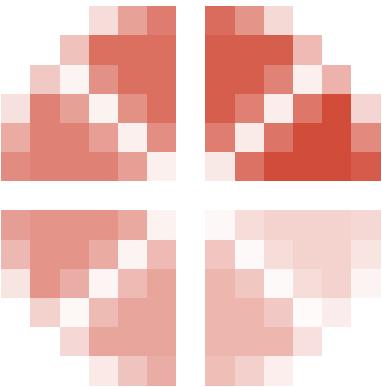
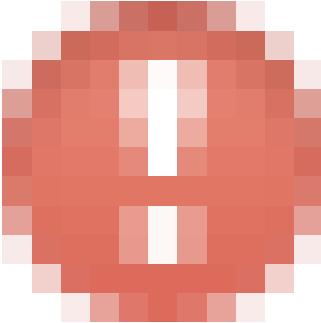
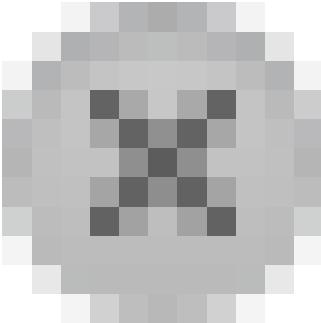
The build state icon appears next to each build under the expanded view of the build configuration on the Projects page.

On this page:

- [Build States](#)
  - [Canceled/Stopped build](#)
- [Personal Build States](#)
- [Hanging and Outdated Builds](#)
- [Failed to Start Builds](#)

### Build States

Icon	State	Description
	running successfully	A build is running successfully.
	successful	A build finished successfully in all specified build configurations.

	running and failing	A build is failing.
	failed	A build failed at least in one specified build configuration.
	cancelled	A build was cancelled.

#### Canceled/Stopped build

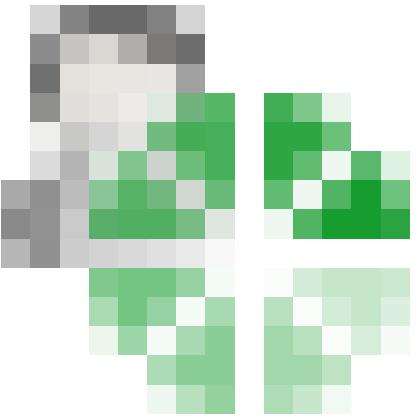
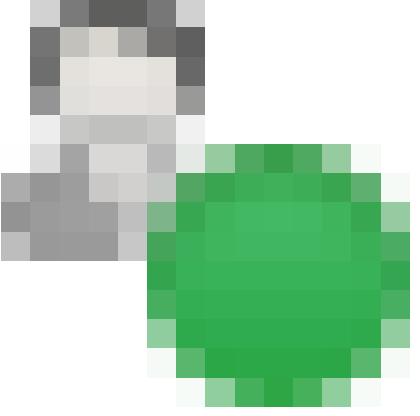
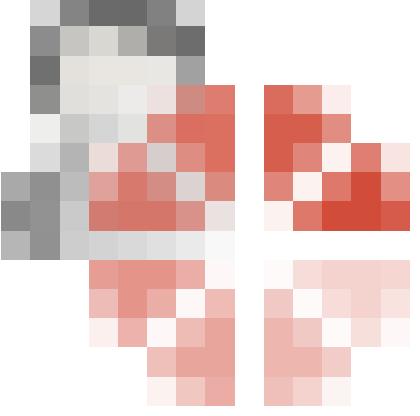
Stopping a running build results in the build status displayed as cancelled. You can stop a running build from the [build results page](#), [build configuration home page](#) or using the Stop option from the Actions drop-down.

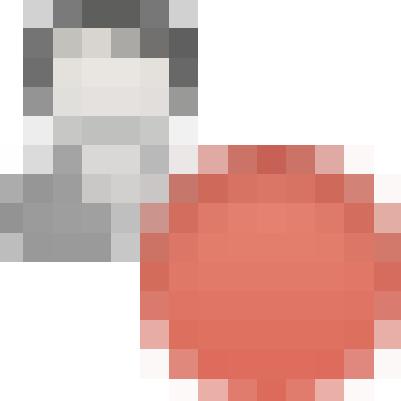
When a build is started, the build process calls the runner process and listens to its output. The stop command kills the runner process, then the build process stops.



It is possible to configure your build so that it will continue executing build steps after the build was stopped. To do it, you can add a build step with the Always, even if build stop command was issued option selected. See [Configuring Build Steps](#).

## Personal Build States

Icon	State	Description
	running successfully	A personal build is running successfully.
	successful	A personal build has completed successfully for all specified build configurations.
	running and failing	A personal build is running with errors.

	failed	A personal build failed at least in one specified build configuration.
---	--------	--

## Hanging and Outdated Builds

TeamCity considers a build as hanging when its run time significantly exceeds estimated average run time and the build did not send any messages since the estimation exceeded.

A running build can be marked as Outdated if there is a build which contains more changes but it is already finished.

Hanging and outdated builds appear with the  icon. Move the cursor over the icon to view a tooltip that displays additional information about the warning.

## Failed to Start Builds

Builds which failed to start, i.e. did not get to the point of launching the first build step are marked with the  icon. It may be caused by a VCS repository being down when the build starts, or the inability to resolve artifact dependencies and so on. Such build status is often an indication of a configuration error and should usually be addressed by a build engineer rather than a developer if there is such roles separation.

If such an error occurs, TeamCity:

- doesn't send build failed notification (unless you have subscribed to "the build fails to start" notification)
- doesn't associate pending changes with this build, i.e. the changes will remain pending, because they were not actually tested
- doesn't show such build as the last finished build on the overview page
- such builds will not affect the build configuration status and the status of developer changes
- shows a "configuration error" stripe for a build configuration with such a build

See also:

[Concepts: Build Configuration Status | Change | Change State](#)  
[User's Guide: Viewing Your Changes](#)

## Build Tag

Build tags are labels that can help you to:

- organize history of your builds
- quickly navigate to the builds marked with a specific tag
- search for a build with a particular tag
- create an artifact dependency on a build with a particular tag

You can assign any number of tags for a single build, for example, "EAP" or "release" using the Edit tags dialog by entering several tags separated by a space, comma, semi-colon, etc.

Clicking a tag filters out all builds in the history: only the builds marked with the tag are displayed. Additionally, you can search for builds with particular tags using the search field.

To tag a build:

The TeamCity Web UI provides the following ways to tag a particular build:

- using the [Build Configuration Home Page](#):
  - go to either the Overview or History tab
  - go to the build history table
  - use the Tags column for the desired build and add/modify the build tag.
- using the [Build Results Page](#) for the particular build:
  - click the Actions button
  - select Tag... and add/modify the build tag
- using the Queued build page:
  - click the Actions button
  - select Tag... and add/modify the build tag
- using the [Run Custom Build](#) dialog
  - go to the Comments and Tags tab and add/modify the build tag
- Using the [Pin/Unpin build](#) dialog, where you can tag the build in addition to pinning it. For builds with snapshot dependencies, there is an option to pin and tag the build as well as its snapshot dependencies.

## Change

Any modification of the source code which you introduce. If a change has been committed to the version control system, but not yet included in a build, it is considered pending for a certain build configuration.

TeamCity suggests several ways to view changes:

- The [Changes](#) page shows the list of changes made by TeamCity users and how they have affected different builds. By default, your changes are displayed. The page has a users selector enabling you to view changes made by any other TeamCity user the same way you see your own changes.
- Pending changes are accessible from the Projects page, build configuration page, or the [build results](#) page.

Viewing and analyzing changes involves the following possibilities:

- Observing actual changes that are already included in the build using the Changes link on the Projects page or on the Changes tab of [the build results](#).
- Observing pending changes in the Pending Changes tab of the the Build Configuration Home Page.
- Viewing the [revision](#) of the sources corresponding to each change
- Navigating to the related issues in a bug tracking system.
- Navigating to the source code and viewing differences.

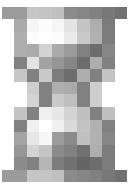
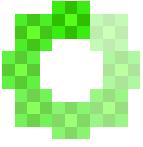
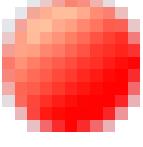
- Starting an investigation of a failed build, if your changes have caused a build failure.

See also:

[Concepts: Revision, Build Configuration](#)

[User's Guide: Investigating and Muting Build Problems](#)

## Change State

Icon	State	Description
	pending	<p>The change is scheduled to be integrated into a build that is currently in the build queue.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 5px;"> <span style="color: #0070C0;">i</span> Even if a change has been successfully integrated into a build, the change will appear as pending when it is scheduled to be added to a different build.         </div>
	running successfully	<p>The change is being integrated into a build that is running successfully.</p>
	successful	<p>The change was integrated into build that finished successfully.</p>
	running and failing	<p>The change is being integrated into a build that is failing.</p>
	failed	<p>The change was integrated into a build that failed.</p>

### Personal Change States

Icon	State	Description
------	-------	-------------

	pending	The change is scheduled to be integrated into a personal build that is currently in the build queue.
	running successfully	The change is being integrated into a personal build that is running successfully.
	successful	The change was integrated into a personal build that finished successfully.
	running and failing	The change is being integrated into a personal build that is failing.
	failed	The change was integrated into a personal build that failed.

See also:

[Concepts: Build Configuration Status | Build State | Change](#)  
[User's Guide: Viewing Your Changes](#)

## Difference Viewer

TeamCity Difference Viewer allows reviewing the differences between two versions of a file modified in the source control and navigating between these differences. You can access the viewer from almost any place in the TeamCity UI where the changes lists appear, for example, the Projects page, Build Configuration Home Page, or the [Changes](#) tab of the build results page. Comparing images in the GIF, PNG or JPG file formats are also supported.

Clicking the name of a modified file opens the viewer:

```
1 package jetbrains.buildServer.serverSide.impl;
2
3 import com.intellij.openapi.diagnostic.Logger;
4 import java.util.*;
5 import java.util.concurrent.ConcurrentHashMap;
6 import java.util.concurrent.ConcurrentMap;
7 import jetbrains.buildServer.buildTriggers.*;
8 import jetbrains.buildServer.log.Loggers;
9 import jetbrains.buildServer.serverSide.*;
10 import jetbrains.buildServer.serverSide.dependency.BuildTypeDependencyGraph;
11 import jetbrains.buildServer.serverSide.systemProblems.SystemProblem;
12 import jetbrains.buildServer.serverSide.systemProblems.SystemProblemNotification;
13 import jetbrains.buildServer.serverSide.systemProblems.SystemProblemTicket;
14 import jetbrains.buildServer.util.EventDispatcher;
15 import jetbrains.buildServer.util.ItemProcessor;
16 import jetbrains.buildServer.util.SystemTimeService;
17 import jetbrains.buildServer.util.TimeService;
18 import org.jetbrains.annotations.NotNull;
19
20 /**
21 * @author Pavel.Sher
22 */
23 public class BuildTriggersChecker implements BuildTriggersProcessor {
24     private final static Logger LOG = Logger.getInstance(BuildTriggersChecker.class);
```

The window heading displays the file modifications summary:

- the file name along with its status,
- the changes author,
- the comment on the changes list.

To move between changes, use the next and previous change arrows and the red and green bars on the versions separator.

If you want to switch to your IDE and explore a change in detail, click the Open in the IDE button in the upper-right corner of the window or select it in the pop-up next to the file name. The file opens, and you will navigate to this particular change.

See also:

Concepts: Project | Build Configuration

## Project

A project in TeamCity is a collection of [build configurations](#). TeamCity project can correspond to a software project, a specific version/release of a project or any other logical group of the build configurations.

The project has a name, an [ID](#), and an optional description.

In TeamCity, user [roles and permissions](#) are managed on per-project basis.

On this page:

- [Project Hierarchy](#)
  - [Settings Propagation](#)
  - [Root Project](#)

### Project Hierarchy

Projects can be nested and organized into a tree allowing for hierarchical display and settings propagation. The hierarchy is defined by the project administrators and is the same for all the TeamCity users.

You can view the hierarchy on the overview page, in the Projects popup, and in breadcrumbs.

## Settings Propagation

The projects hierarchy is used in the following ways:

Settings defined on a project level are propagated to all the subprojects (recursively). These include:

- Parameters
- Clean-up rules
- Versioned Settings (Settings for synchronizing the project settings with version control)
- Since TeamCity 10.0 it is possible to export a project with all its subprojects and external dependencies using the [Settings Export](#) page.

Entities defined in a project become available to all the build configurations residing under the project and its subprojects. These include:

- VCS Roots
- Build Configuration Template
- Issue Trackers
- Shared Resources
- SSH keys
- Maven Settings
- Meta-Runners

For example, if you want to share a VCS root among several projects, you have to move it to the common parent of all these projects. If a VCS root must be shared among all projects, it must be created in the <Root project>.

A setting referencing a project affects the project and all its subprojects. These include:

- User and User group roles
- Investigations
- Muted Problems
- Notification rules

Please note that associating a project with an [agent pool](#) is not propagated to its subprojects and affects only the build configurations residing directly in the project.

## Root Project

TeamCity always has a <Root project> as the top of the projects hierarchy. The root project has most of the properties of a usual project and the settings configured in the root project are available to all the other projects on the server.

The root project is special in the following ways:

- it is present by default and cannot be deleted.
- it is the top-level project, so it has no parent project.
- it can have no build configurations.
- it does not appear in the user-level UI and is mostly present as an entity in Administration UI only.

See also:

[Concepts: Build Configuration](#)

[Administrator's Guide: Managing Projects and Build Configurations | Creating and Editing Projects | Creating and Editing Build Configurations | Project Export](#)

## Build Agent

A TeamCity Build Agent is a piece of software which listens for the commands from the TeamCity server and starts the actual build processes. It is [installed and configured](#) separately from the TeamCity server. An agent can be installed on the same computer as the server or on a different machine (the latter is a preferred setup for server performance reasons); an agent can run the same operating system (OS) as the TeamCity server or a different OS.

On this page:

- [Build Agent Status](#)
- [Agent Upgrade](#)

A TeamCity build agent contains [two processes](#):

- Agent Launcher — a Java process that launches the agent process
- Agent — the main process for a Build Agent; runs as a child process for the agent launcher

An Agent typically checks out the source code, downloads artifacts of other builds and runs the build process. An agent can run a single build at a time. The number of agents basically limits the number of parallel builds and environments in which your build processes are run.

An Agent can run builds of any compatible build configuration.

The TeamCity server monitors all the connected agents and assigns queued builds to the agents based on [compatibility requirements](#), [Agent Pools](#), Build Configuration restrictions configured for an agent and the selection algorithm described [here](#).

## Build Agent Status

In TeamCity, a build agent can have following statuses:

Status	Description
Connected/ Disconnected	An agent is connected if it is registered on the TeamCity server and responds to server commands, otherwise it is disconnected. This status is determined automatically.
Authorized/ Unauthorized	Agents are manually authorized via the web UI on the Agents page (except for the agents from the machines launched by the <a href="#">cloud integrations</a> ). Only authorized build agents can run builds. The number of authorized agents at any given time cannot exceed the number of <a href="#">agent licenses</a> entered on the server. When an agent is unauthorized, a license is freed and a different build agent can be authorized. Purchase additional licenses to expand the number of agents that can concurrently run builds. When a new agent is registered on the server for the first time, it is unauthorized by default and requires manual authorization to run the builds.  <div style="border: 1px solid #ccc; padding: 5px; border-radius: 5px;"><span style="color: #0072bc; font-size: 1.5em;">i</span> If a build agent is installed and running on the same computer as the TeamCity build server, it is authorized automatically.</div>
Enabled/ Disabled	Agents are manually enabled/disabled via the <a href="#">web UI</a> . The TeamCity server only distributes builds to agents that are enabled.  <div style="border: 1px solid #ccc; padding: 5px; border-radius: 5px;"><span style="color: #0072bc; font-size: 1.5em;">i</span> Agent disabling does not affect (stop) the build which is currently running on the agent.</div> Disabled agents can still run builds, when the build is assigned to a special agent (e.g. by <a href="#">Triggering a Custom Build</a> ). This feature is generally used to temporarily remove agents from the build grid to investigate agent-specific issues.

All agents connected to the server must have unique agent names.

Only users with certain roles can manage agents. See [Role and Permission](#) for more information.

For a build agent configuration, refer to the [Build Agent Configuration](#) section.

## Agent Upgrade

A TeamCity agent is upgraded automatically when necessary. The process involves downloading new agent files from the TeamCity server and restarting the agent on the new files. In order to successfully accomplish this, the user under whose account the agent runs should have [enough](#) permissions.

Typically, an agent upgrade happens when:

- the server is [upgraded](#)
- an agent plugin is [added](#) or [updated](#) on the server
- a new tool is [installed](#)

See also:

Concepts: [Build Grid](#) | [Agent Work Directory](#) | [Role and Permission](#)  
Installation and Upgrade: [Installing and Running Build Agents](#) | [Setting up and Running Additional Build Agents](#) |  
Administrator's Guide: [Agent Pools](#) | [Assigning Build Configurations to Specific Build Agents](#) | [Licensing Policy](#)

# Agent Home Directory

The Build Agent Home Directory is the directory where the agent is installed.

The [Build Agent](#) can be installed into any directory.

If you use the TeamCity [.tar.gz distribution or .exe distribution](#) opting to install a Build Agent, the agent will be placed into <TeamCity Home>/buildAgent.

The default directory suggested by the .exe agent installation is C:\BuildAgent.

The agent stores all related data under its directory and the only place that requires installation/uninstallation into an OS is integrating into [the automatic start system](#) (e.g. service settings under Windows).

## Agent Directories

The agent consists of:

- agent binaries (stored under `bin`, `launcher` and `lib` directories). The binaries can be automatically updated from the server to match the server version.
- agent plugins and tools (stored under `plugins` and `tools` directories). These are parts of agent binary installation and are managed by the agent itself, updating automatically whenever necessary from the TeamCity server.
- agent configuration (stored under `conf` and `launcher\conf` directories). This is a unique piece of information defining the agent settings and behavior.
- [agent work directory](#) (stored under the `work` directory by default, configurable via agent configuration).
- agent auxiliary data (stored under `system`, `temp`, `backup`, `update` directories). The data necessary during agent running.
- agent logs (stored under `logs` directory) - The directory storing internal agent logs that might be necessary for agent issues investigation.

## Agent Files Modification

The agent configuration directory is the only one designed to have files that can be edited by the user.

All the other directories should not be edited by the user.

The content of the agent work directory can be deleted (but only entirely). This will result in a [clean checkout](#) for all the affected builds.

The content of directories storing agent auxiliary data can be deleted (but only entirely and while the agent is not running). Deletion of data can result in extra actions during next builds on this agent, but this is meant to have only a performance impact and should not affect consistency.

## Important Agent Files and Directories

- `/bin`
  - `agent.bat` — batch script to start/stop the build agent from the console under Windows
  - `agent.sh` — shell script to start/stop the build agent under Linux/Unix
  - `service.install.bat` — batch file to install the build agent as a Windows service. See also [related section](#).
  - `service.start.bat` — starts the build agent using the installed build agent service
  - `service.stop.bat` — stops the installed build agent service
  - `service.uninstall.bat` — batch file to uninstall the currently installed build agent Windows service
- `/conf/` — this folder contains all configuration files for the build agent
  - `buildAgent.properties` — [main configuration file](#). This file is generated by the TeamCity server .exe installer and build agent .exe installer.
  - `buildAgent.dist.properties` — sample configuration file. You can rename it to 'buildAgent.properties' to create an initial agent configuration file.
  - `teamcity-agent-log4j.xml` — build agent logging settings. For details, please refer to comments inside the file or to the [log4j manual](#)
- `/launcher/conf/`
  - `wrapper.conf.template` — sample configuration file to be used as a template for creating the original configuration
  - `wrapper.conf` — current build agent Windows service configuration. This is a Java Service Wrapper configuration java properties file. For details, please see comments inside the file or Java Service Wrapper documentation.
- `/logs`
  - `launcher.log` — log of the build agent launcher
  - `teamcity-agent.log` — main build agent log
  - `wrapper.log` — log of the Java Service Wrapper. Available only if the build agent is running as a windows

- service
  - teamcity-build.log — log from the build
  - upgrade.log — log from the build agent upgrade process
  - teamcity-vcs.log — agent-side checkout logs
- /system
  - .artifacts\_cache — cache for all build's artifacts; can be configured
- /temp — temporary folder; the path can be overridden in the `buildAgent.properties` file
  - agentTmp — temporary folder that is used by the build agent to store build-related files during the build. Is cleaned after each build.
  - buildTmp — temporary folder that is set as the default temp directory for the build process and is cleaned after each build
  - globalTmp — temporary folder that is used by the build agent for its own temporary files. Is cleaned on the agent restart.

## Agent Requirements

Agent requirements are used in TeamCity to specify whether a [build configuration](#) can run on a particular [build agent](#) besides [Agent Pools](#) and [configured Build Configuration restrictions](#).

When a build agent registers on the TeamCity server, it provides information about its configuration, including its environment variables, system properties and additional settings specified in the `buildAgent.properties` file.

The administrator can specify required environment variables and system properties for a build configuration on the [Build Configuration Settings | Agent Requirements page](#). For instance, if a particular build configuration must run on a build agent running Windows, the administrator specifies this by adding a requirement that the `teamcity.agent.jvm.os.name` system property on the build agent must contain the `Windows` string.

If the properties and environment variables on the build agent do not fulfill the requirements specified by the build configuration, then the build agent is incompatible with this build configuration. The [Agent Requirements page](#) lists both compatible and incompatible agents. Multiple agent requirements for a single parameter can be added now. The conditions are treated as 'and' to determine compatible agents.

It is possible to disable a configured requirement using the corresponding option in the requirements list.

Sometimes the build configuration may become incompatible with a build agent if the build runner for this configuration cannot be initialized on the build agent. For instance, .NET build runners do not initialize on UNIX systems.

## Implicit Requirements

Any reference (name in %-signs) to an unknown parameter is considered an "implicit requirement". That means that the build will only run on the agent which provides the parameters named.

Otherwise, the parameter should be made available for the build configuration by defining it on the build configuration or project levels.

For instance, if you define a build runner parameter as a reference to another property: `%env.JDK_16%/lib/*.jar`, this will implicitly add an agent requirement for the referenced property, that is, `env.JDK_16` should be defined. To define such properties on agent you may either specify them in the `buildAgent.properties` file, or set the environment variable `JDK_16` on the build agent, or you can specify the value on the [Parameters page](#) of a build configuration (in the latter case, the same value of the property for all build agents will be used).

See also:

<a href="#">Concepts: Build Agent   Build Configuration</a> <a href="#">Administrator's Guide: Assigning Build Configurations to Specific Build Agents   Configuring Build Agent Startup Properties</a>   <a href="#">Configuring Build Parameters</a>   <a href="#">Configuring Agent Requirements</a>
--

## Agent Work Directory

Agent work directory is the directory on a build agent that is used as a containing directory for the default checkout directories. By default, this is the `<Build agent home>/work` directory.

To modify the default directory location, see `workDir` parameter in [Build Agent Configuration](#).

 Please note that TeamCity assumes full control over the directory and can [delete subdirectories](#) if they do not correspond to checkout directories of the recent builds.

For more information on handling the directories inside the agent work directory, please refer to [Build Checkout Directory](#) section.

See also:

[Concepts: Build Agent | Build Checkout Directory](#)

## Authentication Modules

There are two types of authentication modules in TeamCity:

- Credentials Authentication Module authenticates users with a login/password pair specified on the login page.
- HTTP Authentication Module authenticates users with some information from certain HTTP request.

You can enable several credentials authentication modules and several HTTP authentication modules simultaneously.

On an attempt to login via the login page, TeamCity asks all the available credentials authentication modules in the order they are specified and the first one that can authenticate the user, authenticates him/her. And for any HTTP request, if there is no authenticated user yet, TeamCity asks all enabled HTTP authentication modules in the order they are specified and the first one that can authenticate the user, authenticates him/her (if no HTTP authentication module can authenticate the user for the specified HTTP request, TeamCity redirects the user to the login page).

TeamCity supports the following credentials authentication modules:

- Built-in (cross-platform): Users and their passwords are maintained by TeamCity. New users are added by the TeamCity administrator (in the Administration area) or they can register themselves if the user registration at the first login is allowed by the administrator.
- Microsoft Windows domain (cross platform): All NT domain users that can log on to the machine running the TeamCity server, can also log in to TeamCity using the same credentials. i.e. to log in to TeamCity users should provide domain and user name (`DOMAIN\username`) and their domain password.
- LDAP server (cross-platform): Authentication is performed by directly logging into LDAP with credentials entered into the login form.

The following HTTP authentication modules are supported:

- Basic HTTP (cross-platform): Allows to access certain web server pages and perform actions from various scripts.
- NTLM HTTP (only for Windows servers): Allows to login using NTLM HTTP protocol. Depending on the client's web browser and operating system can provide an ability to login without typing the user's credentials manually.

Please refer to [Configuring Authentication Settings](#) for specific authentication modules configuration. See also [Accessing Server by HTTP](#) page for details about accessing server from your scripts using basic HTTP authentication.

See also:

[Administrator's Guide: Accessing Server by HTTP | LDAP Integration | Configuring Authentication Settings](#)  
[Extending TeamCity: Custom Authentication Module](#)

## Build Chain

On this page:

- [Common Use Case](#)
- [Configuring Build Chains](#)
- [Stopping/Removing From Queue Builds from Build Chain](#)
- [Build Chains Visual Representation](#)
  - [Dependencies page of build configuration settings](#)
  - [Build Chains tab of project home and build configuration home page](#)
  - [Dependencies tab of build results page](#)

A build chain is a sequence of builds interconnected by [snapshot dependencies](#). Thus, all the builds in a chain use the same snapshot of the sources. Sometimes the build chain is called a "pipeline".

## Common Use Case

The most common use case for specifying a build chain is running the same test suite of your project on different platforms. For example, before a release build you want to make sure the tests are run correctly under different platforms and environments. For this purpose, you can instruct TeamCity to run tests, then an integration build first, and a release build after that.

Let's see how the build chain mechanism works in details. On triggering a dependent build of the Release build configuration, TeamCity does the following:

1. Resolves a chain of all build configurations that the Release build configuration snapshot depends on.
2. Checks for changes for all dependent build configurations and synchronizes them when the first build in the build chain enters a build queue.
3. Adds all the builds that need building with specific revisions to the build queue.

## Configuring Build Chains

To specify dependencies in your build configuration:

1. On the Build Configuration Settings page, select Dependencies
2. On the Dependencies page, click the Add new snapshot dependency link.

See also [Build Dependencies Setup](#) for details and an example.

## Stopping/Removing From Queue Builds from Build Chain

If a build being stopped or removed from the build queue is a part of [Build Chain](#), there is a message below the comment field: This build is a part of a build chain.

If there are other running or queued parts of the build chain (i.e. other running builds or queued builds, which are connected with the build under the action), these builds will be listed below under the label: Stop other parts::

If a user has access rights to stop a build in the list, there is a checkbox near it. The checkbox is selected by default if stopping the current build will definitely cause the build in the list to fail (for instance, if the listed build depends on the original build being stopped).

If user has no access right to stop a build from the list, the checkbox is not visible.

Selecting the checkbox marks the selected build for a stop/removal from queue.

If a user has no access right to view a build which is a part of the build chain, this build is not visible to the user at all. If there is at least one such build, there is a warning displayed: You don't have access rights to all its parts. The stripe is shown right under the message "This build is a part of a build chain".

In case when all other parts of the build chain cannot be viewed by the current user, we show a yellow stripe with warning: You don't have access rights to see its other parts.

If there are no running or queued builds for the build chain (i.e. all other parts of the build chain have finished), no additional information is shown.

## Build Chains Visual Representation

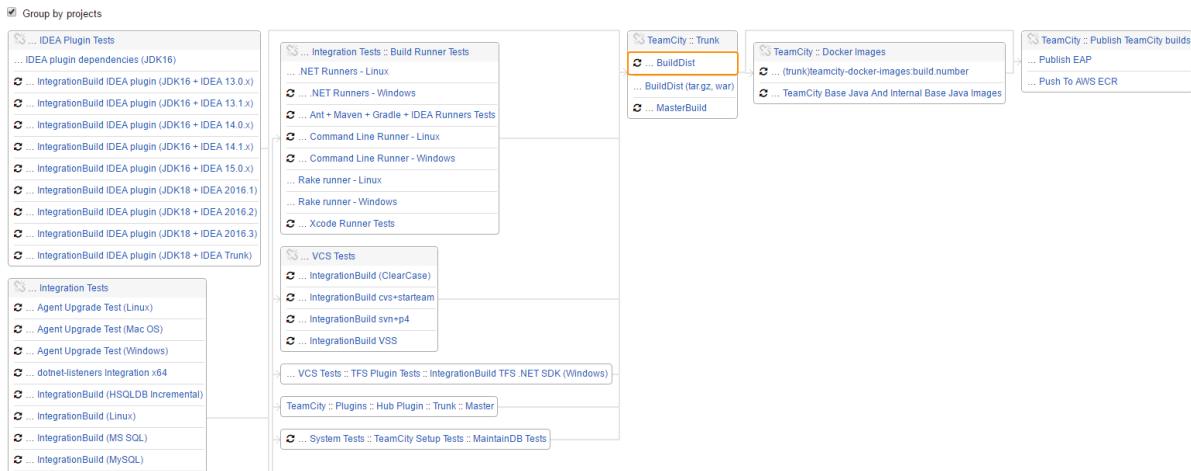
Basically, each build chain is a [directed acyclic graph](#), i.e. it cannot have cycles.

Build Chains are visible in various places in the TeamCity Web UI:

- [Dependencies page of build configuration settings](#)
- [Build Chains tab of project home and build configuration home page](#)
- [Dependencies tab of build results page](#)

## Dependencies page of build configuration settings

If a build configuration is a part of a build chain, the corresponding information is displayed in the Build Configuration settings > Dependencies page | Snapshot dependencies. Clicking the build chain link opens the preview of the build chain and its configuration in a separate window. The preview shows builds of the chain; the builds with automatic triggering configured are marked with the  icon:

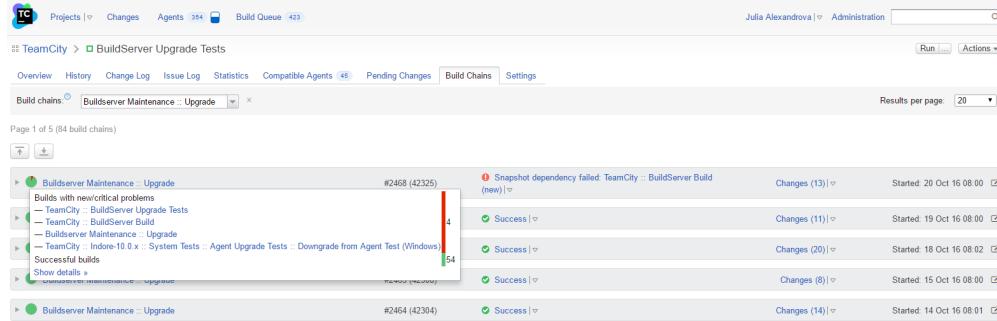


## Build Chains tab of project home and build configuration home page

You can review build chains on both project and build configuration pages: each of those pages has a Build Chains tab appearing when snapshot dependencies are configured.

The tab displays the list of build chains that contain builds of this project or this build configuration with the ability to filter them. Note that build chains are sorted so that the build chain with the last finished build appears at the top of the list.

The pie-chart icon displays the ratio of the statuses for builds that are parts of the chains. On hovering over the pie chart, the details are displayed:



The screenshot shows the 'Build Chains' tab for the 'BuildServer Upgrade Tests' project. There are 84 build chains listed. One chain, 'Buildserver Maintenance : Upgrade', is expanded, showing its structure. The expanded view includes a tree view of the build chain and a table of individual build details. The table columns include the build name, ID, status, changes, start time, and duration. The status column uses icons to represent build statuses like success, failure, and pending.

Build	Status	Changes	Started	Duration
Snapshot dependency failed: TeamCity : BuildServer Build (new)   ↗	Failure	Changes (13)   ↗	20 Oct 16 08:00	00:00:00
Success   ↗	Success	Changes (11)   ↗	19 Oct 16 08:00	00:00:00
Success   ↗	Success	Changes (20)   ↗	18 Oct 16 08:02	00:00:00
Success   ↗	Success	Changes (8)   ↗	15 Oct 16 08:00	00:00:00
Success   ↗	Success	Changes (14)   ↗	14 Oct 16 08:01	00:00:00

When a chain is expanded, the following information is also available:

- all builds this build chain is comprised of
- status of these builds: not triggered, in queue, running or finished and its details
- the chain displays builds in order of actual execution, i.e. builds that start first are on the left.

Clicking a build in a chain highlights the selected build and all its direct dependencies. This page

- provides a compact representation of chains: if several top builds triggered the same chain of dependent builds, TeamCity displays one build chain with several "top builds".
- has additional display options: "Group by projects" and "Hide details".
- transitively highlights all the downstream/upstream builds when a build is selected in a build chain.

From this page you can also:

- Continue a chain, if there are yet "not triggered" builds. Click the Run button and a new build will be started on the chain revisions and associated with builds from this chain.
- Click  to open the custom build dialog with build chain revisions preselected. This action can be used if you want to

re-run some build in the chain.

## Dependencies tab of build results page

If dependencies are configured, you can view their details build results page, the Dependencies tab. This tab also displays indirect dependencies, e.g. if a build A depends on a build B which depends on builds C and D, then these builds C and D are indirect dependencies for build A.

The tab also displays artifacts downloaded and delivered by the builds of the chain. It also allows grouping/ungrouping builds and highlighting the builds reused from previous chains ([suitable builds](#)).

See also:

[Concepts: Dependent Build](#)

[Administrator's Guide: Configuring Dependencies | Build Dependencies Setup](#)

## Build Checkout Directory

On this page:

- [Checkout Process](#)
- [Custom checkout directory](#)
- [Automatic Checkout Directory Cleaning](#)

The build checkout directory is a directory on the TeamCity agent machine where all of the sources of all builds are checked out into.

- If you use the [agent-side checkout mode](#), the build agent checks out the sources into this directory before the build.
- In case you use the [server-side checkout mode](#), the TeamCity server sends incremental patches to the agent to update only the files changed since the last build in the given checkout directory
- With the [manual checkout](#) mode, no sources will be checked out but the default build checkout directory will still be created to check out the sources via a build script. The directory will not be cleaned automatically unless as its expiration period is configured as described below.

The sources are placed into the checkout directory according to the mapping defined in the [VCS Checkout Rules](#).

The checkout directory is configured in the Checkout Settings section on the [Version Control Settings](#) page; the default Auto (recommended) value is strongly advised, but it is possible to configure a custom checkout directory as described below.

If you want to investigate an issue and need to know the directory used by the build configuration, you can get the directory from the build log, or you can refer to the [Agent Work Directory](#)/directory.map generated file which lists build configurations with the directories they used last.

In your [build script](#) you can refer to the effective value of the build checkout directory via the `teamcity.build.checkoutDir` property provided by TeamCity.

 By default, this is also the directory where builds will run.

## Checkout Process

For checkout handled by TeamCity (the [server-side](#) or [agent-side](#) checkout mode), TeamCity keeps track of the last revision checked out into each checkout directory on the agent and for the new build applies an incremental patch from the last used revision to the revision of the current build.

The revisions used can be looked up on the [Changes](#) tab of the build results page.

Incremental checkouts mean that any files not created or modified by TeamCity (e.g. by the previous build scripts) are preserved in their modified state (unless dedicated VCS root-specific reset options are used).

That is why it is recommended to:

- make sure the builds perform a clean procedure as the first step of the build for all the files that affect the build and might have been produced by previous builds. Typical files are compilation output, tests reports, build produce artifacts.
- make sure the builds never modify or delete the files under version control

If TeamCity detects that it cannot build an incremental patch, a [clean checkout](#) is enforced. It can also be enforced manually or configured to be performed on each build.

## Custom checkout directory

In most cases, the default Auto (recommended) setting will cover your needs. With this default checkout directory TeamCity ensures best performance and consistent incremental sources updates. The name of the default automatically created directory is generated as follows: <Agent Work Directory>/<VCS settings hash code>. The VCS settings hash code is calculated based on the set of VCS roots, their checkout rules and VCS settings used by the build configuration (checkout mode). Effectively, this means that the directory is shared between all the build configurations with the same VCS settings.

If for some reason you need to specify a custom checkout directory (for example, the process of creating builds depends on some particular directory), make sure that the following conditions are met:

- the checkout directory is not shared between build configurations with different VCS settings (otherwise TeamCity will perform [clean checkout](#) each time another build configuration is built in the directory);
- the content of the directory is not modified by processes other than those of a single TeamCity agent (otherwise TeamCity might be unable to ensure consistent incremental sources update). If this cannot be eliminated, make sure to turn on the clean build checkout option for all the participating build configurations. This rule also applies to two TeamCity agents sharing the same working directory. As one TeamCity agent has no knowledge of another, the other agent is appearing as an external process to it.



It is not recommended that a custom checkout directory contain the [agent's working directory](#) because of the potential undesirable side effects.

Note that content of the checkout directory can be deleted by TeamCity under certain circumstances.

## Automatic Checkout Directory Cleaning

With the [server-side](#) and [agent-side checkout](#) modes, checkout directories are automatically deleted from the disk if not used (no builds were run on the agent using the directory as the checkout directory) for a specified period of time (8 days by default) or when another build requires more free disk space than available. With [manual checkout](#) mode, automatic directory cleaning is not performed unless the directory expiration period is configured.

It is recommended to use the [Free disk space](#) build feature to ensure the build gets enough disk free space on the build agent.

There are also obsolete manual controls for old checkout directories cleanup:

The time frame for automatic directory expiration can be changed by specifying a new value (in hours) by either of the following ways:

- '`teamcity.agent.build.checkoutDir.expireHours`' agent property in the [buildAgent.properties](#) file;
- '`system.teamcity.build.checkoutDir.expireHours`' [Build Configuration](#) property

Setting the property to "0" will cause deleting the checkout directories right after the build finishes.

Setting the property to "never" will let TeamCity know that the directory should never be deleted by TeamCity.

Setting the property to "default" will enforce using the default value.

Expiration-based directory cleaning is performed in the background when the build agent is idle (no builds are running).

See also:

[Administrator's Guide: Configuring VCS Settings](#)

## Build Grid

A build grid is a pool of computers ([Build Agents](#)) used by TeamCity to simultaneously create builds of multiple projects. The build grid employs currently-unused resources from multiple computers, any of which can run multiple builds and/or tests at a time, for single or multiple projects across your company.

See also:

[Concepts: Build Agents | Build Queue](#)

## Build History

Build history is a record of the past builds produced by TeamCity.

To view the build history, click the Projects tab, expand the desired project and build configuration, and click a build result link. In the Build history section of the [Build Results Home Page](#) page, click previous and next links to browse through, or click All history link to open the history page.

## Build Number

Each build in TeamCity is assigned a build number, which is a string identifier composed according to the pattern specified in the build configuration setting on the [General settings](#) page.

This number is displayed in the UI and passed into the build as a [predefined property](#).

A build number can be:

- [Used to download artifacts](#)
- [Referenced as a property](#)
- [Shared for builds connected by a dependency](#)
- [Used in artifact dependencies](#)
- [Set with help of service messages](#)

See also:

[Administrator's Guide: Build number format](#)

## Build Runner

Build runner is a part of TeamCity that allows integration with a specific build tool (Ant, MSBuild, Command line, etc.). In a build configuration, a build runner defines how to run a build and report its results.

Each build runner has two parts:

- the server-side settings that are configured through the web UI
- the agent-side part that executes a build on an agent

TeamCity comes bundled with the following runners:

- .NET CLI (dotnet)
- .NET Process Runner
- Ant
- Command Line — run an arbitrary command line
- Deployers
  - Container Deployer
  - FTP Upload
  - SMB Upload
  - SSH Upload
  - SSH Exec
- Duplicates Finder (ReSharper)
- Duplicates Finder (Java)
- FxCop
- Gradle
- Inspections (IntelliJ IDEA) — a set of IntelliJ IDEA inspections
- Inspections (ReSharper) — a set of ReSharper inspections
- IntelliJ IDEA Project, and its earlier version: Ipr (deprecated)
- Maven
- MSBuild
- MSpec
- NAnt
- NuGet-related runners
- NUnit
- PowerShell
- Rake
- Simple Build Tool (Scala)
- Visual Studio (.sln) — Microsoft Visual Studio 2005/2008/2010/2012/2013/2015 solutions
- Visual Studio 2003 — Microsoft Visual Studio 2003 solutions
- Visual Studio Tests (with integrated MSTest)
- Xcode Project

Technically, build runners are implemented as plugins.

Build runners are configurable in the [Build Steps](#) section of the [Create/Edit Build Configuration](#) page.

See also:

[Administrator's Guide: Configuring Build Steps](#)

## Build Working Directory

The build working directory is the directory set as current for the build process. By default, this is the same directory as the [Build Checkout Directory](#).

If the build script needs to run from a location other than the checkout directory, you can specify the location explicitly using the Working Directory field of the Advanced options in the Build Runner settings.

 Not all build runners provide the working directory setting.

The path entered in the Working Directory field can be either absolute or relative to the build checkout directory. When using this option, all of the other paths should still be entered relative to the checkout directory.

See also:

[Concepts: Build Checkout Directory](#)

## Clean Checkout

Clean Checkout (also referred to as "Clean Sources") is an operation that ensures that the next build will get a copy of the sources fetched all over from the VCS. All the content of the [Build Checkout Directory](#) is deleted and the sources are re-fetched from the version control.

### Enforcing Clean Checkout

Clean checkout is recommended if the checkout directory content was modified by an external process via adding new or modifying, or deleting existing files.

You can enforce clean sources action for a build configuration from the Build Configuration home page (Actions drop-down in the top right corner), or for an agent from the [Agent Details](#) page using the Miscellaneous section, the Clean sources on this agent option. The action opens a list of agents/build configurations to clean sources for.

 If you set specific folder as the Build Checkout Directory (instead of using default one), you should remember that all of the content of this directory will be deleted during clean checkout procedure.

TeamCity maintains an internal cache for the sources to optimize communications with the VCS server. The caches are reset during the [cleanup time](#). To resolve problems with sources update, the caches may need to be reset manually using the [Diagnostics | Caches](#) tab in the Web UI or by deleting the <TeamCity Data Directory>/system/caches directory.

## Automatic Clean Checkout

You can also enable automatic cleaning the sources before every build, if you check the option Clean all files before build on the [Create/Edit Build Configuration](#)> Version Control Settings page. If this option is checked, TeamCity performs a full checkout before each build.

If clean checkout is not enabled, TeamCity updates the sources in the checkout directory incrementally to the required state.

TeamCity tries to detect if the sources in the checkout directory are not corresponding to the expected state and triggers clean checkout in such cases to ensure sources are appropriate. This means that under certain circumstances TeamCity can detect clean checkout is necessary even if it is not enabled in the VCS settings and not requested by the user from web UI. In such cases all the content of the checkout directory is deleted and it is re-populated by the sources from scratch. If any details are available on the decision, they are added into the build log before checkout-related logging.

Here is the summary of cases when TeamCity performs automatic clean checkout:

- if it is enabled using the Clean all files in the checkout directory before the build option in the "Version Control Settings" of the build configuration.
- build checkout directory was not found or is empty (either the build configuration is started on the agent for the first time or the directory has disappeared since the last build). This also covers
  - particularly when no builds were run in a specific checkout directory for a configured (or default) time and the directory became empty. See more at [automatic checkout directory cleaning](#).
  - particularly when there were not enough [free space on disk](#) in one of the earlier builds and the directory was deleted
- a user invoked "Enforce clean checkout" action from the web UI for a build configuration or agent
- the build was triggered via Custom Run Build dialog with "Clean all files in checkout directory before build" option selected or by a trigger with corresponding option
- the build was triggered by a Schedule trigger with "Clean all files in checkout directory before build" option enabled or as a part of a build chain where the topmost build was triggered with the setting in the schedule trigger while "apply to all snapshot dependencies" was also selected
- VCS settings of the build configuration were changed
- the previous build in this directory was of a build configuration with different VCS settings (can only occur if the same checkout directory is specified for several build configurations with individual VCS settings and VCS Roots)
- the previous build in this directory was built on more recent revisions than the current one (can only occur for [history builds](#))
- there was a critical error while applying or rolling back a patch during the previous build, so TeamCity cannot ensure that checkout directory contains known versions of files
- [Build Files Cleaner \(Swabra\)](#) is enabled with corresponding options and it detected that clean checkout is necessary.
- Custom checkout directory contains agent-specific parameters, such as %teamcity.agent.work.dir% (pre-8.1)

## Clean-Up

Clean-up in TeamCity is a feature allowing automatic deletion of data belonging to old builds.

Project-related clean-up settings are configured in the Project Settings.

The general clean-up configuration is available in the server Administration | Clean-up Settings.

It is recommended to configure clean-up rules to remove obsolete builds and their artifacts, purge unnecessary data from database and caches in order to free disk space, remove builds from the TeamCity UI and reduce the TeamCity workload.

Clean-up deletes the data stored under [TeamCity Data Directory/system](#) and in the database. Also, during the clean-up time the server performs various maintenance tasks (e.g. resets VCS full patch caches).

On this page:

- [Server Clean-up Settings](#)
  - [Manual Clean-up Launch](#)
- [Project Clean-up Rules](#)
  - [Clean-up for Dependent Builds](#)
  - [Clean-up in Build Configurations with Feature Branches](#)
  - [Clean-up of Personal Builds](#)
  - [Deleted Build Configurations Cleanup](#)

## Server Clean-up Settings

The server settings are configured on the [Administration | Server Administration | Clean-up Settings](#).

The build history clean-up is run as a background process, which means that now there is no server maintenance down-time.

 If you use the HSQL database, there is a short period of server unavailability when the HSQL database is being compacted.

Depending on the amount of data to clean up, the process may take significant time, during which the server might be less performant. Therefore, it is recommended to schedule clean-up to run during off-peak hours. By default, TeamCity will start cleaning up daily at 3.00 AM. It is also possible to [run it manually](#).

You can also specify the time limit for the clean-up process. In case not all the data is purged within the time-frame specified, the remaining data will be removed during the next clean-up process.

With clean-up enabled, TeamCity will keep the server [audit records](#) for a year (365 days) by default.

### Manual Clean-up Launch

The Previous clean-up section of the server clean-up settings enables you to:

- review the information on the previous server clean-up date and duration helping you decide whether to launch the clean-up process at a given moment
- run clean-up manually using the Start clean-up now button.

During clean-up, TeamCity reports the progress. If you need, you can stop the clean-up process and the remaining data will be removed during the next clean-up.

## Project Clean-up Rules

Clean-up rules are configured per project and define when and what data to clean.

To manage the rules, use [Project-Settings | Clean-up Rules](#). The Clean-up Rules page allows assigning different rules to a project, and the templates or build configurations within this project.

The following inheritance rules apply:

- if a clean-up rule is assigned to a project, it becomes default for all configurations or subprojects in this project
- if a clean-up rule is assigned to a template, it becomes default for all configurations inherited from this template, but if a clean-up rule is assigned to both a template and a project, the rule from the project will override the rule from the template
- if a clean-up rule is assigned to a build configuration, it will override the clean-up rule from a project or a template.

In each rule, you can define a number of successful builds to preserve, and/or the period of time to keep builds in history (e.g. keep builds for 7 days).

The following clean-up levels are available:

- Artifacts (all other data including build logs is preserved. [Hidden Artifacts](#) are also preserved);
- History (all the build data is deleted except for builds statistics values that are visible in the [statistics charts](#));
- Everything (no build data remains in TeamCity).  
Each level includes the one(s) listed above it.

By default, everything is kept forever. When you select custom settings, for each of the items above you can specify:

- the number of days. Builds older than the number of days specified will be cleaned with the specified level. The starting point is the date of the last build, not the current date. A day is equivalent to a 24-hour period, not a calendar day;

- the number of successful builds. Only builds older than the last matching successful build will be cleaned with the level specified (all the failed builds between the preserved successful ones are kept). This rule is only taken into account if there are successful builds in the build configuration.

When both conditions are specified, only the builds which must be cleaned according to all rules will be actually removed: TeamCity finds the oldest build to preserve according to each of the rules and then cleans all builds older than the oldest one of the two found.

For the Artifacts level you can also specify the patterns for the artifact names:

- Artifact patterns. The artifacts matching the specified pattern will be in/excluded from the clean-up. Use newline-delimited rules following [Ant-like pattern](#).

Examples:

- to clean-up artifacts with 'file' as a part of the name, use the following syntax: +:\*\*/file\*.\*.
- to exclude \*.jar artifacts with 'file' as a part of the name from clean-up, use -:\*\*/file\*.jar.

There are builds that preserve all their data and are not affected during cleanup. These are:

- [pinned builds](#);
- builds used as a source for [artifact dependency](#) in other builds when the "Prevent clean-up" option for dependency artifacts is enabled. See [Clean-Up for Dependent Builds](#) below. Such builds are marked with  icon in the build history list;
- builds used as [snapshot dependency](#) in other not yet deleted builds;
- builds of build configurations that were deleted less than one day ago.

### Clean-up for Dependent Builds

The settings in the Dependencies section of the Edit Clean Up Rules dialog affect clean-up of artifacts in builds that the builds of the current build configuration depend on.

TeamCity always preserves builds which are used as [snapshot dependencies](#) in other builds. These builds are not deleted from builds history by the clean-up procedure until dependent builds are deleted. Artifacts of these builds can be deleted based on the option below.

TeamCity can optionally preserve builds and their artifacts which are used in other builds by [artifact dependencies](#).

- Use default choice uses the option configured in the default cleanup rule.
- Prevent clean-up choice protects builds (and their artifacts) which were used as a source of artifact or snapshot dependencies for the builds of the current build configuration.
- Do not prevent clean-up (default) choice makes cleanup-related processing of the dependency builds disregard the fact that they are used by the builds of the current build configuration.

Example:

Say, a build configuration A has an artifact dependency on B. If Prevent clean-up option is used for A, the builds of B that provide artifacts for the builds of A will not be processed while cleaning the builds, so the builds and their artifacts will be preserved.

### Clean-up in Build Configurations with Feature Branches

If a build configuration has builds from several [branches](#), before applying clean-up rules, TeamCity splits the build history of this configuration into several groups. TeamCity creates one group per each [active branch](#), and a single group for all builds from inactive branches. Then clean-up rules are applied to each group independently.

### Clean-up of Personal Builds

Clean-up rules are applied separately for the non-personal builds and then for the personal builds. That is, if you have a rule to preserve 3 successful builds, 3 non-personal builds and 3 personal builds are preserved (in each branch group as per description above).

### Deleted Build Configurations Cleanup

When a project or a build configuration is deleted, the corresponding builds data is removed during the cleanup, but only if 5 days (432,000 seconds) have passed since the deletion. To change the timeout, set the `teamcity.deletedBuildTypes.cleanupTimeout` [internal property](#) to the required number of seconds to protect the data from deletion.

See also:

[Concepts: Dependent Build](#)

## Code Coverage

Code coverage is a number of metrics that measure how your code is covered by unit tests. TeamCity supports the following coverage engines out of the box:

- Java, see [Configuring Java Code Coverage](#)
  - IntelliJ IDEA coverage (bundled)
  - EMMA open-source toolkit (bundled)
  - JaCoCo open-source (bundled)
- .NET: see [Configuring .NET Code Coverage](#)
  - JetBrains dotCover (bundled)
  - NCover 1.x, 3.x
  - PartCover

For importing reports from other coverage tools, see the related [notes](#).

For importing coverage results in TeamCity, see [this section](#).

To get the code coverage information displayed in TeamCity for the supported tools, you need to configure it in the dedicated section of a [build runner](#) settings page. The following build runners include code coverage support:

- Ant
- IntelliJ IDEA Project
- Maven
- MSBuild
- NAnt
- NUnit
- MSpec
- MSTest
- .NET Process Runner
- Ipr (deprecated)

Note that currently the Maven2 runner supports [IntelliJ IDEA](#) and [JaCoCo](#) coverage engines.

The code coverage results can be viewed on the Overview tab of the [Build Results page](#); detailed report is displayed on the dedicated [Code Coverage](#) tab.

The chart for code coverage is also available on the [Statistics](#) tab of the build configuration.

For the details on configuring code coverage, refer to the dedicated pages: [Configuring Java Code Coverage](#), [Configuring .NET Code Coverage](#).

See also:

[Concepts: Build Runner](#)

[Administrator's Guide: Configuring Java Code Coverage](#) | [Configuring .NET Code Coverage](#) | [Integrating with External Reporting Tools](#)

## Code Duplicates

Code Duplicates are repetitive blocks of code. The Duplicates Finder [build runners](#) search for similar code fragments and provide a comprehensive report on repetitive blocks of code discovered in your code base.

See also:

[Administrator's Guide: Duplicates Finder \(Java\)](#) | [Duplicates Finder \(ReSharper\)](#)

## Code Inspection

TeamCity comes with code analysis tools capable of inspecting your source code on the fly, finding and reporting common problems and anti-patterns.

The following inspections tools are bundled with TeamCity:

- [Inspections \(IntelliJ IDEA\)](#): runs code analysis based on [IntelliJ IDEA inspections](#). More than 600 Java, HTML, CSS, JavaScript inspections are performed by this runner.
- [Inspections \(ReSharper\)](#): gathers results of [JetBrains ReSharper Code Analysis](#) in your C#, VB.NET, XAML, XML, ASP.NET, JavaScript, CSS and HTML code.

Inspection results are reported in the [Code Inspection tab](#) of the build results page.

TeamCity can also [be integrated with external reporting tools](#).

See also:

[Concepts: Build Runner](#)  
[Administrator's Guide: Inspections \(IntelliJ IDEA\)](#) | [Inspections \(ReSharper\)](#)

## Continuous Integration

Continuous integration is a software engineering term describing a process that completely rebuilds and tests an application frequently. Generally it takes the form of a server process or daemon that:

- Monitors a file system or version control system (e.g. CVS) for changes.
- Runs the build process (e.g. a make script or Ant-style build script).
- Runs test scripts (e.g. JUnit or NUnit).

Continuous integration is also associated with [Extreme Programming](#) and other agile software development practices.

Following the principles of Continuous Integration, TeamCity allows users to monitor the software development process of the company, while improving communication and facilitating the integration of changes without breaking any established practices.

## Dependent Build

In TeamCity one build configuration can depend on one or more configurations. Two types of dependencies can be specified:

- [Snapshot Dependency](#)
- [Artifact Dependency](#)

An artifact dependency is just a way to get artifacts produced by one build into another. Without a corresponding Snapshot dependency, it is mainly used when the build configurations are not related in terms of sources. For example, one build provides a reusable component for others.

A snapshot dependency influences the way builds are processed and implies that the builds are deeply related, one build being a logic part of another.

### Snapshot Dependency

Snapshot Dependency is a powerful concept that allows expressing source-level dependencies between build configurations in TeamCity.

The primary goal is to allow complex build procedures via creating different build configurations linked with snapshot dependencies. This in particular allows dividing a single monolith build into a set of interlinked builds ([Build Chain](#)) with flexible reuse rules. TeamCity follows the declarative style of defining the build structure on this level (declaring dependencies rather than adding build triggers) as it allows for more flexible and powerful features.

See [Build Dependencies Setup](#) for a description of typical snapshot dependencies usages and related blog posts: [April, 2012](#), [March, 2016](#)

A snapshot dependency of build configuration A on build configuration B ensures that each build of A has a "suitable" build of B before build of A can start. Both builds of A and B use the same sources snapshot (revision of the sources being the same or taken at the same time if the VCS roots are different) when they belong to the same chain.

The build results page of a build with snapshot dependencies allows reviewing all the dependency builds and their errors, if any.

A snapshot dependency alters the builds behavior in the following way:

- when a build is queued, so are the builds from all the Build Configurations it snapshot-depends on, transitively; TeamCity then determines the revisions to be used by the builds ("checking for changes" process).
- if some of the build configurations already have started builds with matching changes ("suitable builds") and the snapshot dependency has the "Do not run new build if there is a suitable one" option ON, TeamCity optimizes the queued builds by using an already finished builds instead of the queued ones. Corresponding queued builds are then silently removed. This procedure can be performed several times, because, while builds of the chain remain in the queue, new builds may start and finish;
- all builds linked via snapshot dependencies are started by TeamCity with explicit specification of the sources revision. The revision is calculated so that it corresponds to the same moment in time (for the same VCS root it is the same revision number). For a queued build chain (all builds linked with a snapshot dependency), the revision to use is determined after adding builds to the queue. At this time, all the VCS roots of the chain are checked for changes and the current revision is fixed in the builds;
- if there is a snapshot dependency and artifact dependency on the Build from the same chain pointing to the same build configuration, TeamCity ensures the download of artifacts from the same-sources build.
- by default, builds that are a part of a build chain are preserved from clean-up, but this can be switched on per-build configuration basis. Refer to the [Clean-Up](#) description for more details.

Depending on the dependencies, topology builds can run sequentially or in parallel.

Behavior on the build chain continuation in case of a build failure is customizable via the snapshot dependency options. For each failed or failed to start dependency you can select one of the four options:

- Run build, but add problem: the dependent build will be run and the problem will be added to it, changing its status to failed (if problem was not muted earlier)
- Run build, but do not add problem: the dependent build will be run and no problems will be added
- Make build failed to start: the dependent build will not run and will be marked as "Failed to start"
- Cancel build: the dependent build will not run and will be marked as "Canceled".

A build of a chain can [reference parameters](#) from the preceding builds via `dep.<configurationId>.<parameterName>` syntax.

There is a [special support](#) for pushing parameters down the chain when a build with snapshot dependencies is triggered. It is done by defining a parameter with `reverse.dep.<configurationId>.<parameterName>` name.

When setting up triggers for the builds in the chain, the recommended approach is: think about the result - the build you want to get at the end of the process, and configure triggers in its corresponding, "top" build configuration. No triggers are necessary in the build configurations this top one depends on, as their builds will be put into the queue automatically when the top one is triggered.

See also the related "Trigger on changes in snapshot dependencies" [setting](#) of a VCS trigger and the "Show changes from snapshot dependencies" [check-box](#) in the "Version Control Settings" configuration section.

Let's consider an example to illustrate how snapshot dependencies work.

Let's assume that we have two build configurations, A and B, and configuration A has a snapshot dependency on configuration B.

1. When a build of configuration A is triggered, it automatically triggers a build of configuration B, and both builds will be placed into the Build Queue. Build B starts first and build A will wait in the queue till build B is finished ([if no other specific options are set](#)).
2. When builds B and A are added to the queue, TeamCity adjusts the sources to include in these builds. All builds will be run with the sources taken at the moment the builds were added to the queue.

**i** If the build configurations connected with a snapshot dependency share the same set of VCS roots, all builds will run on the same sources. Otherwise, if the VCS roots are different, changes in the VCS will correspond to the same moment in time.

3. When the build B has finished and if it finished successfully, TeamCity will start to run build A.

**i** Please note that the changes to be included in build A could have become not the latest ones by the moment the build started to run. In this case, build A becomes a [history build](#).

The above example shows the core basics of snapshot dependencies as a straight forward process without any additional options. For snapshot dependency options, refer to the [Snapshot Dependencies](#) page.

## Artifact Dependency

Artifact Dependencies provide you with a convenient means to use the output ([artifacts](#)) of one build in another build. When an artifact dependency is configured, the necessary artifacts are downloaded to the agent before the build starts. You can then

review what artifacts were used in the build or what build used the artifacts of the current build using the Dependencies tab of build results.

To create and configure an artifact dependency, use the [Dependencies](#) build configuration settings page. If you need to download the artifacts inside a build script or locally, you can use the [REST API](#) or [Ivy Ant tasks](#) for that.

**i** Please note that if both a snapshot dependency and an artifact dependency are configured for the same build configuration, in order for it to take artifacts from the build with the same sources, the Build from the same chain option must be selected in the artifact dependency.

**i** [Notes on Cleaning Up Artifacts](#)

Artifacts may not be [cleaned](#) if they were downloaded by other builds and these builds are not yet cleaned up. For a build configuration with configured artifact dependencies, you can specify whether the artifacts downloaded by this configuration from other builds can be cleaned or not. This setting is available on the [cleanup policies](#) page.

See also:

[Concepts: Build Artifact | Build Dependencies Setup](#)  
[Administrator's Guide: Configuring Dependencies](#)

## Guest User

TeamCity provides the ability to turn on the guest login allowing anonymous access to the TeamCity web UI.

A server administrator can [enable guest login](#) on the Administration | Authentication page.

Roles and groups for the guest user can be configured via Guest user settings link available on the Administration | Users page. By default, guest users have the Project Viewer role for all the projects.

When guest user is enabled, any number of guest users can be logged in to TeamCity simultaneously without affecting each other's sessions. Thus, it can be useful for non-committers who just monitor the projects status on the Projects page.

Guest users do not have any personal settings, such as Changes Page and Profile section (i.e. no way to receive notifications).

If guest login is enabled, you can construct a URL to the TeamCity Web interface, so that no user login is required:

- Add `&guest=1` parameter to a usual page URL. The login will be silently attempted on loading the page.

You can use guest login to download artifacts:

- Use `/guestAuth` before the URL path. For example:

```
http://buildserver:8111/guestAuth/action.html?add2Queue=bt7
```

See also:

[Concepts: Role and Permission | Super User](#)  
[Administrator's Guide: Enabling Guest Login](#)

## History Build

A History Build is a build that starts after a build with more recent changes. That is, a history build is a build that disrupts normal builds flow according to the order of the source revisions.

A build may become a history build in the following situations:

- If you initiate a build on particular changes manually using the [Run Custom Build](#) dialog.
- If you have a [VCS trigger](#) with a quiet period set. During this quiet period a different user can start a build with more recent changes. In this case, your automatically triggered build will have an older source revision when it starts, and will be marked as a history build.
- If there are several builds of the same configuration in the build queue and they have fixed revisions (e.g. they are part of a [Build Chain](#)). If someone manually re-orders these builds, the build with fewer changes can be started first.

As the history build does not reflect the current state of the sources, the following restrictions apply to its processing:

- The status of a history build does not affect the project/build configuration status.
- A user does not get notifications about history builds unless they subscribed to notifications on all builds in the build configuration.
- History builds are not shown on the Projects or Build Configuration > Overview page as the last finished build of a configuration.
- The [Investigation](#) option is not available for history builds.

See also:

[Concepts: Build History | Build Queue](#)  
[Administrator's Guide: Triggering a Custom Build](#)

## Notifier

TeamCity supports the following notifiers:

Notifier	Description
Email Notifier	Notifications regarding specified events are sent via email.
IDE Notifier	Displays the status of the build configurations you want to watch and/or the status of your changes.
Jabber Notifier	Notifications regarding specified events are sent via Jabber.
System Tray Notifier	Displays the status of the build configurations you want to watch in the Windows system tray, and displays pop-up notifications on the specified events.
Atom/RSS Feed Notifier	Notifications regarding specified events are sent via an Atom/RSS feed.

You can configure the notifier settings, create, change and delete notification rules in the Watched Builds and Notifications section of the [My Settings&Tools](#) page.

See also:

[User's Guide: Subscribing to Notifications](#)  
[Administrator's Guide: Customizing Notifications](#)

## Personal Build

A personal build is a build out of the common builds sequence and which typically uses the changes not yet committed into the version control. Personal builds are usually initiated from one of the [supported IDEs](#) via the [Remote Run](#) procedure.

The build uses the current VCS repository sources plus the changed files identified during the remote run initiation. The results of the Personal Build can be seen in the "My Changes" view of the corresponding IDE plugin and on the [Changes](#) page. Finished personal builds are listed in the builds history, but only for the users who initiated them.

See more at [Pre-Tested \(Delayed\) Commit](#).

By default, users only see their own personal builds in the builds lists, but this can be changed on the [user profile page](#).

One can also mark a build as personal using the corresponding option of the [Run...](#) dialog.  
By default, only users with the [Project Developer role](#) can initiate a personal build.

Since TeamCity 9.1, it is possible to [restrict running personal builds](#).

See also:

[Concepts: Pre-tested Commit | Remote Run](#)  
[Installing Tools: IntelliJ Platform Plugin | Eclipse Plugin | Visual Studio Addin](#)  
[Troubleshooting Remote Run Problems](#)

## Pinned Build

A build can be "pinned" to prevent it from being removed when a clean-up procedure is executed, as stipulated by the [clean-up policy](#).

There are several ways to pin/unpin a build :

- using the [Build Results page](#), the Build Action drop-down menu
- on the Overview tab of the Build Configuration Home Page using the pin icon

The Pin/Unpin build dialog also allows you to tag the build.

If the build has snapshot dependencies, the Apply to all snapshot dependencies box appears. Checking the box will apply the actions (pinning/unpinning, adding / removing tags) to all dependency builds of the current one.

See also:

[Concepts: Clean-up policy](#)

## Pre-Tested (Delayed) Commit

An approach which prevents committing defective code into a build, so the entire team's process is not affected. The diagrams on the JetBrains TeamCity web pages provide visualization of the TeamCity approach described below.

The submitted code changes go through testing first . If the code passes all of the tests, TeamCity can automatically submit the changes to the version control. From there, the changes will automatically be integrated into the next build. If any test fails, the code is not committed, and the submitting developer is notified.

Developers test their changes by performing a [Remote Run](#). A pre-tested commit is enabled when the commit changes if successful option is selected.

The pre-tested commit is initiated via a plugin to one of [supported IDEs](#). For remote run a command-line tool is also [available](#).

For Git and Mercurial the recommended way to use [Branch Remote Run Trigger](#) approach to run personal builds off branches.

Matching changes and build configurations

To submit changed files to pre-tested commit or remote run, VCS integration should work in the IDE and TeamCity should be able to check that the files, when committed, will affect the build configurations on the server.

If TeamCity cannot match the changes with the builds, a message "Submitted changes cannot be applied to the build configuration" is displayed. The VCS integration should be correctly configured in the IDE, and TeamCity should be able to match the files on the developer workstation to the build configurations present on the TeamCity server.

In order to be able to do that, VCS should be configured in the same way on the developer's workstation and on the server. This includes:

- for CVS, TFS and Perforce version control systems - use exactly the same URLs to the version control server
- for VSS - use exactly the same path to the VSS database (machine and path)
- for Subversion - use the same server (TeamCity matches [server UUID](#))
- for Git - the current checked out branch should have "remote" set to the server/branch monitored by the TeamCity server and should have common commits in the history with the server-monitored branch.

If upon changed files choosing TeamCity is unable to find build configurations that the files can be sent to, the option to initiate the personal build will not be available.

General Flow of a pre-tested commit

- A developer uses the Remote Run dialog of a [TeamCity IDE plugin](#) to select the files to be sent to TeamCity.
- Based on the selected files, a list of applicable build configurations is displayed. The developer selects the build configurations to test the change against and sets options for a pre-tested commit.
- The TeamCity IDE plugin builds a "patch" - full content of all the files selected and sends it to the TeamCity server. The patch is also preserved locally on the developer's machine. When sent, the change appears on the developer's [changes](#) page. Developer can continue working with the code and can modify the files sent to the pre-tested commit.
- The personal build is queued and processed like other queued builds.
- When the build starts, it checks out the latest sources just like a normal build and then applies the developer's personal changes sent from the IDE over (full file content is used)
- The build runs as usual
- At the end of the build the personal changes are reverted from the build's checkout directory to make sure they do not affect following builds
- The TeamCity IDE plugin pings the TeamCity server to check if all the selected build configurations have personal builds ready. If a build fails, a notification is displayed in the IDE and the process ends.
- If all the personal builds finish successfully, the IDE plugin displays a progress, backs up the current version of the files participating in the personal change (as they might already be modified since the pre-tested commit was initiated), then restores the file contents from the saved "patch", performs the version control commit (reports an error if there was an error like a VCS conflict) and restores the just backed up files to bring the working copy in the last seen state. The pre-tested commit in the TeamCity plugin window gets an error or success mark.

See also:

[Concepts: Remote Run](#)  
[Remote Run on Branch: Remote Run on Branch Build Trigger for Git and Mercurial](#)  
[Installing Tools: IntelliJ Platform Plugin | Eclipse Plugin | Visual Studio Addin](#)

## Remote Run

A remote run is a [Personal Build](#) initiated by a developer from one of the supported IDE plugins to test how the changes will integrate into the project's code base. Unlike [Pre-tested Commit](#), no code is checked into the VCS regardless of the state of the personal build initiated via Remote Run.

For a list of version control systems supported by each IDE please see [supported platforms and environments](#).

See more at [Pre-Tested \(Delayed\) Commit](#).

See also:

[Concepts: Pre-tested Commit | Personal Build](#)  
[Remote Run on Branch: Remote Run on Branch Build Trigger for Git and Mercurial](#)  
[Installing Tools: IntelliJ Platform Plugin | Eclipse Plugin | Visual Studio Addin](#)  
[Troubleshooting Remote Run Problems](#)

## Role and Permission

User access levels are handled by assigning different roles to users.

A role is a set of permissions that can be granted to a user in one or all projects thus controlling access to the projects and various features in the Web UI.

A permission is an authorization granted to a TeamCity user to perform particular operations, e.g. to run a build or modify build configuration settings.

On this page:

- [Authorization Mode](#)
- [Changing Authorization Mode](#)
- [Simple Authorization Mode](#)
- [Per-Project Authorization Mode](#)
- [Agent Management Permissions](#)

### Authorization Mode

TeamCity authorization supports two modes: simple and per-project.

In the simple mode, there are only three types of authorization levels: guest, logged-in user and administrator. In the per-project mode, you can assign users Roles in projects or server-wide. The set of permissions in roles is editable.

Permissions within a role granted at the project level are automatically propagated in all the subprojects of this project. The View project and all parent projects permission allows you to view not only the project (with its subprojects) but its parent projects too.

## Changing Authorization Mode

Unless explicitly configured, the simple authorization mode is used when TeamCity is working in the Professional mode and per-project is used when working in the Enterprise mode.

To change the authorization mode, use the **Enable per-project permissions** check box on the Administration | Authentication page.

### Simple Authorization Mode

Administrator	Users with no restrictions; corresponds to the System Administrator role in the per-project authorization mode
Logged-in user	Corresponds to the default Project Developer role granted for all projects in the per-project authorization mode
Guest user	Corresponds to the default Project Viewer role granted for all projects in the per-project authorization mode

### Per-Project Authorization Mode

Roles are assigned to users by administrators on a per-project basis - a user can have different roles in different projects, and hence, the permissions are project-based. A user can have a role in a specific project or in all available projects, or no roles at all. You can [associate a user account with a set of roles](#). A role can also be granted to a user group. This means that the role is automatically granted to all the users that are included into the group (both directly or through other groups).

By default, TeamCity provides the following roles:

Role	Description
System Administrator	TeamCity System Administrators have no restrictions in their permissions, and have all of the project administrator's permissions. They can create and manage users' accounts, authorize build agents and set up projects and build configurations, edit the TeamCity server settings, manage TeamCity licenses, configure server data clean-up rules, change VCS roots, and etc.
Project Viewer	Project Viewer has read-only access to projects and can only view the project, its parent and subprojects. Project Viewer does not have permissions to <a href="#">view agent details</a> .
Project Administrator	Project Administrator is a person who can customize general settings of a project and settings of build configurations, assign roles to the project users, create subprojects, and who has all the project developer's permissions. Prior to TeamCity 10, this role included in the Agent Manager role, since TeamCity 10 <a href="#">agent management permissions</a> (see below) replace the inherited Agent Manager role.
Project Developer	Project Developer is a person who usually commits changes to a project. He/she can start/stop builds, reorder builds in the build queue, label the build sources, review agent details, start investigation of a failed build.
Agent Manager	The Agent Manager role grants a user permissions for customizing and managing <a href="#">Build Agents</a> ; changing the run configuration policy, <a href="#">enabling/disabling</a> build agents, and <a href="#">pausing/resuming</a> build queue.  Prior to TeamCity 10.0, this role is included in the Project Administrator role. Since TeamCity 10, new agent management permissions have been introduced. See the <a href="#">section below</a> .

When per-project permissions are enabled, server administrators can modify the roles, delete them, or add new roles with any combination of permissions right in the TeamCity Administration web UI, or by modifying the `roles-config.xml` file stored in <[TeamCity Data Directory](#)>/config directory. When assigning roles to users, the View role permissions link in the web UI displays the list of permissions for each role in accordance with their current configuration.

## Agent Management Permissions

TeamCity has 6 permissions to perform a task on an agent: a user must have a specific permission granted in a project. A user can perform a task controlled by one of these permissions on all the agents belonging to some [pool](#) provided this permission is granted to the user in all the projects associated with this pool. For example, a user with 'Enable / disable agents associated with project' permission in some projects can enable or disable agents which belong to the pools of the related projects if the permission is granted in all the projects associated with the pools.

In new installations, these project-related permissions are added to the Project Administrator role and the Agent manager role is no longer included in it.

In the existing installations after [upgrade](#), the new permissions are added to the Agent Manager role (which is included in the Project Administrator role). It is recommended to remove the inherited Agent Manager role manually and add the required permission(s) to the Project Administrator role.

The new permissions are:

- 1) Enable / disable agents associated with project
- 2) Start / Stop cloud agent for project
- 3) Change agent run configuration policy for project
- 4) Administer project agent machines (e.g. reboot, view agent logs, etc.)
- 5) Remove project agent
- 6) Authorize project agent

The last permission and the recently introduced "[maximum number of agents](#)" setting for an agent pool enable you to set up the system in a way which allows project administrators to run new agents and authorize/add them to their pools without involving the global system administrator.

See also:

[Concepts: User Account](#)  
[Administrator's Guide: Enabling Guest Login | Managing Users and User Groups](#)

## Run Configuration Policy

The run configuration policy allows you to select the specific build configurations you want a build agent to run. By default, build agents run all compatible build configurations and this isn't always desirable. The run configuration policy settings are located on the Compatible configurations tab of the [Agent Details page](#).

See also:

[Administrator's Guide: Assigning Build Configurations to Specific Build Agents](#)

## TeamCity Data Directory

TeamCity Data Directory is the directory on the file system used by TeamCity server to store configuration settings, build results and current operation files. The directory is the primary storage for all the configuration settings and holds the data critical to the TeamCity installation.

The build history, users and their data and some other data are stored in the [database](#). See notes on [backup](#) for the description of the data stored in the directory and the database.

Note that in this documentation and other TeamCity materials the directory is often referred to as `.BuildServer`. If you have a different name for it, replace ".BuildServer" with the actual name.

On this page:

- [Location of the TeamCity Data Directory](#)
  - [Configuring the Location](#)
  - [Recommendations as to choosing Data Directory Location](#)
- [Structure of TeamCity Data Directory](#)
- [Direct Modifications of Configuration Files](#)
  - [.dist Template Configuration Files](#)
  - [XML Structure and References](#)

### Location of the TeamCity Data Directory

The currently used data directory location can be seen on the Administration | Global Settings page for a running TeamCity server instance. Clicking the Browse link opens the Administration | Global Settings | Browse Data Directory tab allowing the user to upload new/modify the existing files in the directory.

The current data directory location is also available in the `logs/teamcity-server.log` file (look for "TeamCity data directory:" line on the server startup).

If you are upgrading, please note that prior to TeamCity 7.1 the data directory could be specified [in a different way](#).

### Configuring the Location

There are several ways to configure the location of the TeamCity Data directory:

- on the first server startup screens (only when TeamCity .tar.gz or .exe distribution is used). The specified data directory is then saved into `<TeamCity home directory>/conf/teamcity-startup.properties` file. The screen to specify the data directory location does not appear if the data directory location is configured using one of the options below.
- manually using `TEAMCITY_DATA_PATH` environment variable. The variable can be either system-wide or defined for the user under whom TeamCity server is started. After setting/changing the variable, you might need to restart the computer for the changes to take effect.
- If the `TEAMCITY_DATA_PATH` environment variable is not set and `<TeamCity home directory>/conf/teamcity-startup.properties` file does not define it either, the default TeamCity Data Directory location is in the user's home directory (e.g. it is `$HOME/.BuildServer` under Linux and `%USERPROFILE%\BuildServer` under Windows).



Prior to TeamCity 9.1, the TeamCity Windows installer configured the TeamCity data directory during installation by setting the TEAMCITY\_DATA\_PATH environment variable. The default path suggested for the directory was: %ALLUSERSPROFILE%\JetBrains\TeamCity. In the later TeamCity versions, the installer does not ask for the TeamCity data directory and it can be configured on the first TeamCity start.

## Recommendations as to choosing Data Directory Location

Since the data directory stores all the server and configured projects settings, it is important that it is not available for read and write to the OS users without the corresponding level of access. See the related [security notes](#).

Note that by default the `system` directory stores all the [artifacts](#) and build logs of the builds in the history and can be quite large, so it is recommended to place TeamCity Data Directory on a non-system disk. Refer to [Clean-Up](#) section to configure automatic cleaning of older builds. If a single local disk cannot store all of the artifacts, you can add another disk and configure [multiple artifacts paths](#).

Note that TeamCity assumes reliable and persistent read/write access to TeamCity Data Directory and can malfunction if data directory becomes inaccessible. This malfunctions can affect TeamCity functioning while the directory is unavailable and may also corrupt data of the currently running builds. While TeamCity should tolerate occasional data directory inaccessibility, still under rare circumstances the data stored in the directory can be corrupted and be partially lost.

## Structure of TeamCity Data Directory

The `config` subdirectory of TeamCity Data Directory contains the configuration of your TeamCity projects, and the `system` subdirectory contains build logs, artifacts, and database files (if internal database (HSQLDB) is used which is default). You can also review information on [Manual Backup and Restore](#) to understand better which data is stored in the database, and which is on the file system.

- `.BuildServer/config` — a directory where projects, build configurations and general server settings are stored
  - `_trash` — backup copies of deleted projects, it is OK to delete them manually. or details on restoring the projects check [How To...#Restore Just Deleted Project](#)
  - `_notifications` — notification templates and notification configuration settings, including syndication feeds template
  - `_logging` — [internal server logging](#) configuration files, new files can be added to the directory
  - `projects` — a directory which contains all project-related settings. Each project has its own directory. Project hierarchy is not used and all the projects have a corresponding directory residing directly under "projects"
    - `<projectID>` - a directory containing all the settings of a project with the "`<projectID>`" id (including build configuration settings and excluding subproject settings). New directories can be created provided they have mandatory nested files. The `_Root` directory contains settings of the [root project](#). Whenever `*.xml.N` files occur under the directory, they are backup copies of corresponding files created when a project configuration is changed via the web UI. These backup copies are not used by TeamCity.
      - `buildNumbers` — a directory which contains `<buildConfigurationID>.buildNumbers.properties` files which store the current build number counter for the corresponding build configuration
      - `buildTypes` — a directory with `<buildConfiguration or template ID>.xml` files with corresponding build configuration or template settings
      - `pluginData` — a directory to store optional and plugin-related project-level settings. Bundled plugins settings and auxiliary project settings like custom project tabs are stored in plugin-setting `s.xml` file in the directory. Credentials stored outside of VCS per Versioned settings are stored in `secure/credentials.json` file
      - `vcsRoots` — a directory which contains project's VCS roots settings in the files `_<VcsRootID>.xml`
      - `project-config.xml` — the project configuration file containing the project settings, such as [parameters](#) and [clean-up rules](#).
  - `main-config.xml` — server-wide configuration settings
  - `database.properties` — database connection settings, see more at [Setting up an External Database](#)
  - `license.keys` — a file which stores the license keys entered into TeamCity
  - `change-viewers.properties` — [External Changes Viewer](#) configuration properties, if available
  - `internal.properties` — file for specifying various [internal TeamCity properties](#). It is not present by default and needs to be created if necessary
  - `auth-config.xml` — a file storing server-wide authentication-related settings
  - `ldap-config.properties` — [LDAP authentication](#) configuration properties
  - `ntlm-config.properties` — [Windows domain authentication](#) configuration properties
  - `issue-tracker.xml` — issue tracker integration settings
  - `cloud-profiles.xml` — Cloud (e.g. Amazon EC2) integration settings
  - `backup-config.xml` — web UI backup configuration settings
  - `roles-config.xml` — roles-permissions assignment file
  - `database.*.properties` — default template connection settings files for different external databases
  - `*.dtd` — DTD files for the XML configuration files

- \*.dist — default template configuration files for the corresponding files without .dist. See [below](#).
- .BuildServer/plugins — a directory where TeamCity plugins can be stored to be loaded automatically on the TeamCity start. New plugins can be added to the directory. Existing ones can be removed while the server is not running. The structure of a plugin is described in [Plugins Packaging](#).
  - .unpacked — directory that is created automatically to store unpacked server-side plugins. Should not be modified while the server is running. Can be safely deleted if the server is not running.
  - .tools — create this directory to centralize tools to be installed on all agents. Any folder or .zip file under this folder will be distributed to all agents and appear under <agent root>/tools folder.
- .BuildServer/system — a directory where build results data is stored. The content of the directory is generated by TeamCity and is not meant for manual editing.
  - artifacts — the [default directory](#) where the builds' artifacts, logs and other data are stored. The format of the artifact storage is <project ID>/<build configuration name>/<internal\_build\_id>. If necessary, the files in each build's directory can be added/removed manually - this will be reflected in the corresponding build's artifacts.
    - .teamcity subdirectory stores build's [hidden artifacts](#) and build logs (see below). The files can be deleted manually, if necessary, but it is not recommended as the build will lose the corresponding features backed by the files (like the build log, displaying/using finished build parameters including for the build reuse as snapshot dependency, coverage reports, etc.)
      - logs subdirectory stores [build logs](#) in an internal format. Build logs store the build output, compilation errors, test output and test failure details. The files can be removed manually, if necessary, but corresponding builds will lose build log and failure details (as well as test failure details).
  - messages — a directory where build logs used to be stored before TeamCity 9.0. After automatic build logs migration to the new place under artifacts, the directory stores the files which could not be moved (see server log on the server start about details).
  - changes — a directory where the [remote run](#) changes are stored in internal format. Name of the files inside the directory contains internal personal change id. The files can be deleted manually, if necessary, but corresponding personal builds will lose personal changes in UI and when affected queued builds try to start, they fail or run without personal patch.
  - pluginData — a directory where various plugins can store their data. It is not advised to delete or modify the directory. (e.g. state of build triggers is stored in the directory)
    - audit — directory holding history of the build configuration changes and used to display diff of the changes. Also stores related data in the database.
    - repositoryStates — directory holding current state of the VCS roots. If dropped, some changes might not be detected by TeamCity (between the state last queried by TeamCity and the current state after first server start without this data).
  - caches — a directory with internal caches (of the VCS repository contents, search index, other). It can be [manually deleted](#) to clear caches: they will be restored automatically as needed. It is safer to delete the directory while server is not running.
  - buildserver.\* — a set of files pertaining to the embedded HSQLDB.
- .BuildServer/backup — default directory to store backup archives created via [web UI](#). The files in this directory are not used by TeamCity and can be safely removed if they were already copied for safekeeping.
- .BuildServer/lib/jdbc — directory that TeamCity uses to search for [database drivers](#). Create the directory if necessary. TeamCity does not manage the files in the directory, it only scans it for .jar files that store the necessary driver.

## Direct Modifications of Configuration Files

The files under the config directory can be edited manually (unless explicitly noted). The changes will be taken into account without the server restart. TeamCity monitors these files for changes and rereads them automatically when modifications or new files are detected. Bear in mind that it is easy to break the physical or logical structure of these files, so edit them with extreme caution. Always [back up](#) your data before making any changes.

Please note that the format of the files can change with newer TeamCity versions, so the files updating procedure might need adjustments after an upgrade.

The [REST API](#) has means for most common settings editing and is more stable in terms of functioning after the server upgrade.

### .dist Template Configuration Files

Many configuration files meant for manual editing use the following convention:

- Together with the file (suppose named fileName) there comes a file fileName.dist. .dist files are meant to store default server settings, so that you can use them as a sample for fileName configuration. The .dist files should not be edited manually as they are overwritten on every server start. Also, .dist files are used during the server upgrade to determine whether the fileName files were modified by user, or the latter can be updated.

## XML Structure and References

If you plan to modify the configuration manually, note that there are entries interlinked by ids. Examples of such entries are build configuration -> VCS roots links and Project -> parent project links. All the entries of the same type must have unique ids in the entire server. New entries can be added only if their ids are unique.

See also the related [section](#) on moving projects between TeamCity servers.

See also:

[Installation and Upgrade: TeamCity Data Backup](#)

## TeamCity Specific Directories

Directory	Description
<TeamCity home>	This is TeamCity installation directory chosen in Windows installer, used to unpack TeamCity .zip distribution or Tomcat home directory for .war TeamCity distribution.
<TeamCity data directory>	This is the directory used by TeamCity to store configuration and system files.
<agent work directory>	This is the directory on an agent used as default location for build checkout directories.
<agent home>	Build agent installation directory.
<build checkout directory>	The directory used as "root" one for checking out build sources files.
<build working directory>	This is the directory set as current for the build process. By default, the <Build Working Directory> is the same as the <build checkout directory>.
<TeamCity web application>	If you have installed TeamCity using .exe or tar.gz distribution, the path to the directory is <TeamCity home>/webapps/ROOT, by default. For .war distribution, the path to the directory would depend on where you have deployed TeamCity.

## User Account

User account is a combination of username and password that allows TeamCity users to log in to the server and use its features. User accounts can be created manually, or automatically upon log in depending on used authentication scheme (refer to [Authentication Modules](#) and [LDAP Integration](#) sections for more details).

Each user account:

- Has an associated role that ensures access to all or specific TeamCity features through corresponding permissions. Learn more about [roles and permissions](#).
- Belongs to at least one user group. Learn more about [user groups](#).

In addition to logged in users, there is a special user account for non-registered users called Guest User, that allows monitoring TeamCity projects without authorization. By default, guest login is disabled. Learn more at [Guest User section](#).

See also:

Concepts: [User Group](#) | [Role and Permission](#) | [Authentication Modules](#)

Administrator's Guide: [Enabling Guest Login](#) | [LDAP Integration](#) | [Managing User Accounts, Groups and Permissions](#)

## User Group

To help you manage user accounts more efficiently, TeamCity provides User Groups configured using the Administration | Groups page.

Here you can add, delete, edit groups, view parent groups, roles and all users belonging to the group.

A user group allows you to:

- Assign [roles](#) to all users included in the group at once: users get all the roles of the groups they belong to.
- Set the [notification rules](#) for all users in the group: all the notification rules defined for the group are treated as default notification rules for the users included in this group.

You can create as many user groups as you need, and each user group can include any number of user accounts and even other user groups. A user account (or a whole user group) can be included into several user groups as well.

### "All Users" Group

All Users is a special user group that is always present in the system. The group contains all registered users and no user can be removed from the group. You can modify roles and notification rules of the "All Users" group to make them default for all the users in the system.

All the newly registered users get roles and notification rules inherited from the group.

The [Guest User](#) does not belong to the All Users group so roles and notification rules can be managed for the user separately.

See also:

Concepts: [User Account](#) | [Role and Permission](#)

Administrator's Guide: [Managing User Accounts, Groups and Permissions](#)

## Wildcards

TeamCity supports wildcards in different configuration options.

Ant-like wildcards are:

Wildcard	Description
*	matches any text in the file or directory name excluding directory separator ("/" or "\")
?	matches single symbol in the file or directory name excluding directory separator
**	matches any symbols including the directory separator

You can read more on Ant wildcards in the corresponding [section](#) of Ant documentation.

Examples

For a directory structure:

```
\a
-\b
 -\c
   -file1.txt
   -file2.txt
   -file3.log
-\d
 -file4.log
-file5.log
```

Description	Pattern	Matching files
all the files	**	<pre>\a -\b  -\c    -file1.txt    -file2.txt    -file3.log -\d  -file4.log -file5.log</pre>
all log files	**/*.log	<pre>\a -\b  -\c    -file3.log -\d  -file4.log -file5.log</pre>
all files in a/b directory incl. those in subfolders	a/b/**	<pre>\b  -\c    -file1.txt    -file2.txt    -file3.log</pre>
files directly in a/b directory	a/b/*	<pre>\b  -file2.txt  -file3.log</pre>

Already Fixed In

For some test failures TeamCity can show "Already Fixed In" build.

This is the build where this initially failed test was successful and which was run after the build with initial test failure (for the same [Build Configuration](#)).

"After" means here that

- build with successful test has newer changes than the build with initial failure
- if the changes were the same, the newer build was run after the failed one

So, if you run [History Build](#), TeamCity won't consider it as a candidate for "Already Fixed In" for test failures in later builds (in term of changes).

Tests are considered the same when they have the same name. If there are several tests with the same name within the build, TeamCity counts order number of the test with the same name and considers it as well.

Note that since TeamCity 9.0, the way TeamCity counts tests [has changed](#).

See also:

[Concepts: First Failure](#)

## First Failure

TeamCity displays the "First Failure" build for some tests.

This is the build where TeamCity detected the first failure of this test for the same [Build Configuration](#), which means that starting from the current build, TeamCity goes back throughout the build history to find out when this test failed for the first time. Builds without this test are skipped, and if a successful test run was found, the search stops.

"Back throughout the history" means that builds are analyzed with regard to changes as detected by TeamCity, i.e. [History Builds](#) will be processed correctly.

A test which is run several times within a single build is counted as one test (i.e. all invocations of the same test are counted as 1).

See also:

[Concepts: Already Fixed In](#)

## Super User

The Super user login allows you to access the server UI with System Administrator permissions if you do not remember the credentials or need to fix authentication-related settings. The login is performed using authentication token that can be found in the server logs.

Also, Super user token is used to access the server maintenance pages displayed on the server start when a manual action is required to proceed with the server startup.

The authentication token is automatically generated on every server start. The token is printed in the server console and `teamcity-server.log` (search for the "Super user authentication token" text). The line is printed on the server start and on any login page submit without a username specified.

To log in as a super user, use an empty username and the authentication token as the password on the login page.

A super user is not a usual TeamCity user and does not have any personal settings, such as Changes page and Profile section (i.e. there is no way to receive notifications). The super user has all [System Administrator permissions](#).

Any number of super users can log in to TeamCity simultaneously without affecting each other's sessions.

Instead of using an empty username, you can also go to "<Your TeamCity server URL>/login.html?super=1" page and enter the super user authentication token. On loading the super user login page, the super user token is printed into the server log and console again for your convenience.

The super user login is enabled by default, but it can be disabled by specifying the `teamcity.superUser.disable=true` internal property.

See also:

## Identifier

An ID is an identifier given to TeamCity entities ([projects](#), [build configurations](#), [templates](#), and [VCS roots](#), etc.).

Each entity has two identifiers:

- the so-called **external ID**
- the **Universally Unique ID**

On this page:

- [External IDs](#)
  - [Using IDs](#)
  - [Assigning IDs](#)
- [Universally Unique IDs](#)

### External IDs

The so-called 'external' identifiers are configured in the TeamCity web UI (e.g. Project ID) and must be unique within all the objects of the same type on the entire server; note that build configurations and templates share the same ID space.

IDs can contain only alpha-numeric characters and underscores ("\_") - maximum 80 characters - and should start with a Latin letter.

#### Using IDs

External IDs are used:

- in URLs of the web interface (including [RSS feeds](#), [NuGet feed](#)), e.g. <https://teamcity.jetbrains.com/project.html?projectId=TeamCityPluginsByJetBrains>
- in `dep.` and `vcsRoot` parameter references
- in [REST API](#) and build scripts used to automate actions with TeamCity (e.g. download artifacts via direct URLs or Ivy)
- in the configuration files storing settings of projects and build configurations under `<TeamCity data directory>/config`
- in file and directory names under `<TeamCity data directory>/system` (e.g. build artifacts storage)

#### Assigning IDs

By default, TeamCity automatically suggests an ID for an object by converting its name to match the ID requirements and prefixing that with the ID of the parent project.

The ID can be modified manually.

It is recommended to leave the automatically generated IDs as is unless there are special considerations to modify them.

If you consider moving projects between several TeamCity server installations, it is a good practice to make sure that all the IDs are globally unique.



On changing the ID of a project or build configuration, all the related URLs (including the web UI, artifact download links, [RSS feeds](#) and [REST API](#)) will change.

If any of the URLs containing the old IDs were bookmarked or hard-coded in the scripts, they will stop to function and will need update.

At the moment of the ID change, the correspondingly named directories under TeamCity Data Directory (including directories storing settings and artifacts) are renamed and this can take time.

To reset the IDs to match the default scheme for all projects, VCS roots, build configurations and templates, use the Bulk Edit IDs action on the Administration page of the parent [project](#).

To use the automatically generated ID after it has been modified or after you change an existing object name, you can regenerate ID using the Regenerate ID action.

When you copy a project, TeamCity automatically assigns new IDs to all the child elements. The IDs can be previewed and changed in the Copy dialog.

When you move an object, its ID is preserved and you might want to use Regenerate ID action to make the ID reflect the new placement.

### Universally Unique IDs

TeamCity projects, build configurations, and VCS roots have a UUID, an automatically generated, globally unique ID. UUID is stored in the corresponding entity XML configuration file located in the <TeamCity data directory>/config directory. These UUID should never be edited manually. When a new entity is created by placing a file in the data directory, it should have no "uuid" attribute. In this case TeamCity will generate it automatically and will persist in the file.

See also:

[Concepts: Project | Build Configuration](#)

[Administrator's Guide: Managing Projects and Build Configurations](#)|[Creating and Editing Build Configurations](#)| [Configuring VCS Roots](#)|[Accessing Server by HTTP](#)| [Patterns For Accessing Build Artifacts](#)| [REST API](#)

## VCS root

### VCS Roots in TeamCity

A VCS root defines a [connection to a version control system](#) and consists of a set of settings (paths to sources, username, password, and other settings) that defines how TeamCity communicates with a version control (SCM) system to [monitor changes](#) and get sources for a build. A VCS root can be attached to a build configuration or template. You can add several VCS Roots to a build configuration or a template and specify portions of the repository to checkout and target paths via [VCS Checkout Rules](#).

VCS roots are created in a project and are available to all the Build Configurations defined in that project or its [subprojects](#).

You can view all VCS roots configured within the project and create/edit/delete/detach them using the VCS Roots page under the project settings in the Administration UI.

If someone attempts to modify a VCS root that is used in more than one project or build configuration, TeamCity will issue a warning that the changes to the VCS root could potentially affect other projects or build configurations. The user is then prompted to either save the changes and apply them to all the affected projects and build configurations, or to make a copy of the VCS root to be used by either a specific build configuration or project.

On an attempt to create a new VCS root, TeamCity checks whether there are other VCS roots accessible in this project with similar settings. If such VCS roots exist, TeamCity suggests using them.

Once a VCS root is configured, TeamCity regularly queries the version control system for new changes and displays the changes in the [Build Configurations](#) that have the root attached. You can set up your build configuration to trigger a new build each time TeamCity detects changes in any of the build configuration's VCS roots, which suits most cases. When a build starts, TeamCity gets the changed files from the version control and applies the changes to the [Build Checkout Directory](#).

See also:

[Concepts: Project | Build Configuration| Build Configuration Template](#)

[Administrator's Guide: Configuring VCS Roots | VCS Checkout Rules | VCS Checkout Mode | VCS Labeling](#)

## Remote Debug

The Remote Debug is a feature allowing you to remotely debug your tests on the TeamCity agent machine from the IDE on the local developer machine. This feature is of use when the agent environment is unique in some aspect, which causes a test to fail, and it is difficult to reproduce the problem locally.



With [TeamCity integration for IntelliJ-based IDEs](#) enabled, remote debug sessions can be launched right from IntelliJ IDEA for the builds based on the IntelliJ IDEA Project and Ant build steps.

Currently, the following IntelliJ IDEA run configurations are supported:

- Java application run configuration
- JUnit run configuration
- TestNG run configuration

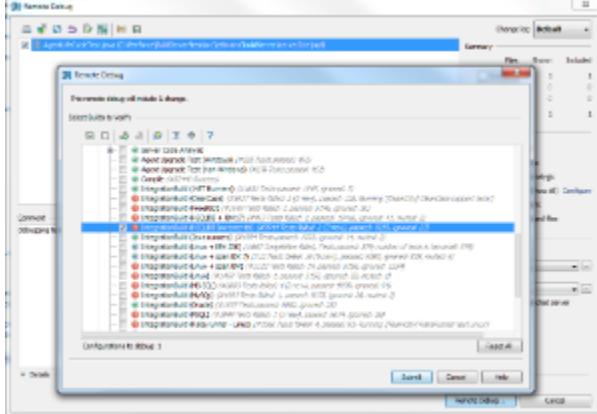
Remote debug for Ant steps requires that the build configuration have the `teamcity.remote-debug.ant.supported=true` parameter.

Prerequisites:

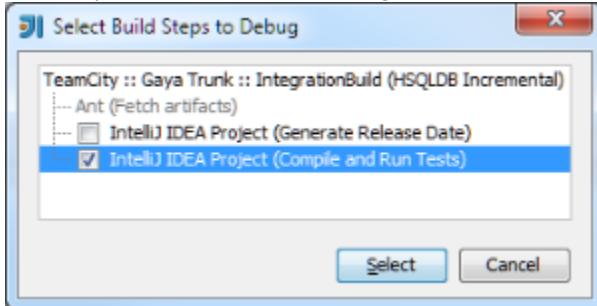
- an IntelliJ IDEA run configuration on the local developer machine with the [TeamCity plugin for IntelliJ IDEA](#) installed,
- a build configuration on the TeamCity Server with the [IntelliJ IDEA Project](#) runner as one of the build steps
- a remote [TeamCity agent](#) to run this build available to the local machine by socket connection

## Debugging Tests Remotely

- To start remote debugging of a test, select the test and choose the Debug <Test Name> Remotely on TeamCity Agent option from the context menu (the Remote Debug action is also available from the TeamCity plugin menu. The action will require you to select an IntelliJ IDEA run configuration).
- Once you do this, the TeamCity plugin will ask you to select a build configuration where you want to start the debug session. The process is similar to starting a personal build. For example, if there are personal changes, a personal patch will be created and sent to an agent. Also, since the process is basically the same, when you select a build configuration, you can specify an agent, customize properties, etc.



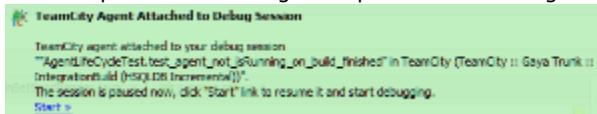
- If the selected configuration contains more than one IntelliJ IDEA Project build step, the plugin will ask you to choose build steps where to start the debug session.



- After that a build is added to the queue and the standard IntelliJ IDEA debug tool window appears:



The debug tool window works in the listening mode, i.e. it waits for the agent to connect to it. Once the agent connects, the Java process on the agent is paused and the Agent Attached notification appears in the IDE:



- Now we can set some breakpoints and actually start the debug session by clicking Start either in the notification popup or in the debug tool window.

Once JVM process exits, another notification popup appears in the IDE:



The debug session is not finished yet, it is possible to either repeat or finish it. Selecting Repeat will rerun the same build step again, which is much faster than starting a new debug session.

## Favorite Build

To easily access builds you want to monitor, you can mark them as favorite.  
On this page:

- [Adding Builds to Favorites](#)
- [Viewing Your Favorites](#)
- [Configuring Notifications on Your Favorite Builds](#)
- [Removing Builds from Favorites](#)

## Adding Builds to Favorites

To manually add a build to your favorites, use the appropriate option from the Actions menu on the [Build results](#) page. Your favorite build will be marked with a star symbol .

Any of your manually triggered builds and [personal builds](#) can be added to your favorites automatically if you enable the corresponding setting in your [user profile settings](#).

You can also add a custom build to favorites by checking [the corresponding box](#).

## Viewing Your Favorites

To view your favorites, in the top right corner of the TeamCity web UI, click the arrow next to your username, and select [My Favorite Builds](#).

You can also configure notifications on your favorite builds using the appropriate link on this page.

## Configuring Notifications on Your Favorite Builds

When [subscribing to notifications](#), you can limit notifications on the selected watched projects or build configurations to your favorite builds only.

## Removing Builds from Favorites

To remove builds from favorites:

- On the Favorite Builds page or [Build results](#) page, click the star symbol .
- Alternatively, on the [Build results](#) page, use the Remove Builds from favorites option from the Actions menu.

## Agent Cloud Profile

On this page:

- [Configuring Cloud Profile](#)
  - [Specifying profile settings](#)
  - [Adding Agent Image](#)
- [Viewing Cloud Agent Information](#)
- [Enabling/disabling Cloud Integration in Project](#)

A cloud profile is a collection of settings for TeamCity to start virtual machines with installed TeamCity agents on-demand while distributing a build queue. Configuring a cloud provider profile is one of the steps required to [enable agent cloud integration](#) between TeamCity and a cloud provider. The settings of profiles slightly vary depending on the cloud type.

 Note that after an upgrade, if you use Kotlin DSL for your TeamCity project settings, you need to update your DSL as described [here](#).

## Configuring Cloud Profile

Prior to TeamCity 2017.1, profiles are configured on the TeamCity | Administration | Agent Cloud page. In the later versions, cloud profiles are configured in the "Cloud Profiles" section of the project settings allowing Project administrators to configure their own cloud profiles without bothering System administrators.

### Specifying profile settings

The following profile settings have to be provided:

Setting	Description
Profile name	Provide a name for the profile
Description	Provide an optional profile description.
Cloud type	Select the cloud provider type from the drop-down
Server URL	This is the URL that the agents cloned from the image will use to connect to the TeamCity server. This URL has to be available from the build agent machine. By default, the <a href="#">TeamCity server URL</a> specified on the Global Settings page of the Administration UI is used. A custom URL can be specified.
Terminate instance idle time	Instruct TeamCity to stop a cloud agent machine using this setting and the options below. Here specify the period (in minutes) for TeamCity to wait before stopping an idle build agent. More than one build may run on the same virtual machine. Leave empty to have no timeout.
Additional terminate conditions	<ul style="list-style-type: none"> <li>After certain work time (minutes): Specify the working time for the agent after which the instance will be terminated. If the time elapses while a build is in progress, the agent will wait until the current build is finished</li> <li>On idle, close to an hour (minutes left to the end of hour): Specify how many minutes before the full hour an idle instance should be stopped: this allows avoiding charges for partial hours if your virtual machines are billed in whole hours, e.g. Amazon EC2 instances.</li> <li>After the first build Select this option if you want TeamCity to stop the virtual machine immediately after the first build finishes. TeamCity will disable the build agents and no more builds will be run on the same machine.</li> </ul>

Next, you need to provide the cloud access information which will differ depending on the provider. After that, you check the connection and add an image to be used as a source for TeamCity cloud agents.

### Adding Agent Image

You need to configure the required options for the image.

Using the maximum instances number settings, you can limit the number of instances across all images (in the cloud profile) and /or set the limit per image (in image settings).

When configuring the image, it is possible to specify which [agent pool](#) the agents should belong to. It is possible if the pool contains this project and/or its subprojects only. Pools containing projects other than the current one and its subprojects will not be available for assignment. If the assigned pool is changed in future so that the criteria are not met or if the agent pool is not specified, the cloud agents will be assigned to a special read-only pool containing only this project and its subprojects. The idea is to restrict the cloud agents pool only to a given project with its subprojects to track cloud costs by project.

After an Agent Cloud profile is created with one or several sources for virtual machines, TeamCity does a test start for all the virtual machines specified in the profile to learn about the agents configured on them. Once agents are connected, TeamCity calculates their build configurations-to-agents compatibility and stores this information.

When a cloud profile is changed, TeamCity detects modifications immediately and forces shutdown of the agents started prior to these changes once the agents finish the current build.

### Viewing Cloud Agent Information

The agents' information is displayed on the Agents | Cloud page under the <Profile name> drop-down.

### Enabling/disabling Cloud Integration in Project

You can enable or disable integration for a project and/or its subprojects via the TeamCity web UI using the Change cloud integration status option on the Cloud profiles page.

## TeamCity Home Directory

The TeamCity Home Directory or the TeamCity Installation Directory is the directory where the TeamCity server application files and libraries have been unpacked when TeamCity was [installed](#). The location of the TeamCity Home directory is defined when you install the TeamCity server. The default directory suggested by the Windows [installation package](#) is C:\Program Files (x86)\JetBrains\TeamCity; however, TeamCity can be installed into any directory.

## Important Files and Directories

`TeamCity-readme.txt` – the directory description

`BUILD <number>` - TeamCity server application build number

`Uninstall.exe` – used to uninstall the currently installed server

- `/bin` - contains executable binary files and scripts (only available in TeamCity `.tar.gz` and `.exe` distributions)
  - `runAll.bat` — batch script to start/stop the server and a build agent from the console under Windows
  - `runAll.sh` — shell script to start/stop the server and a build agent under Linux/Unix
  - `teamcity-server.bat` — batch script to start/stop the server only from the console under Windows
  - `teamcity-server.sh` — shell script to start/stop the server only under Linux/Unix
  - `maintainDB.cmd` ---Windows Command line script to **back up, restore, and migrate** the server data between different databases
  - `maintainDB.sh` — shell script to **back up, restore, and migrate** the server data between different databases
- `/buildAgent` - **Build Agent Home Directory**
- `/conf/` — contains all configuration files for the TeamCity server
  - `server.xml` – the main server configuration file
- `/devPackage/` - the **bundled development package** that can be used to start developing TeamCity plugins.
- `/jre/` - the bundled JRE installation directory
- `/licenses/` - licenses for the **third-party libraries** distributed with TeamCity
- `/logs` - contains the **TeamCity server logs**
- `/temp` — temporary folder
- `/webapps` - TeamCity web application data
- `/work` - standard **Tomcat folder** where Tomcat writes cache files for every page it serves.

## Revision

A revision refers to a specific state of a version control history; basically it is a version of the source code. When changes occur, they are usually identified by a number or letter code, termed "revision".

When displaying **changes** included in a finished or queued build, TeamCity also displays the corresponding revision. This section explains how the VCS revisions are chosen by TeamCity for a build.

TeamCity will use the current repository revision when a new **VCS root** is configured for a **project** or a **build configuration**, or when the **configured VCS root settings** have been modified.

After that TeamCity does not monitor the whole repository but only collects changes for the scope of the repository specified in TeamCity: the **configured VCS root settings with checkout rules**. The revision of the sources corresponding to the latest detected change affecting your build will be displayed as the VCS root revision on the **Changes** page accessible via the link on the Projects page or on the **Changes tab** of the build results.

If the settings of a VCS Root get modified since the last detected change, the revision in TeamCity will be different from the last change in the newly configured VCS root. TeamCity does not have any information on the previous change in this new root, so it starts to monitor changes with the new settings and sets the build revision to the first discovered change. Until the change is discovered, there is no way to get any revision other than the current revision of the repository. Therefore, while TeamCity builds the correct revision of the sources, the revision for the first build after a VCS root change will not be equal to the last change under the specified path.

See also:

Concepts: [Change, Build Configuration](#)

User's Guide: [Investigating and Muting Build Problems](#)

## Mapping TeamCity Concepts to Other CI Terms

This information can be used when migrating to TeamCity from other Continuous Integration tools.

See the [Jenkins to TeamCity Migration Guidelines](#) for mapping of Jenkins concepts to TeamCity.

Bamboo	TeamCity
Project	Project
Plan	Build Configuration
Stage	Build Step

Job	Build
Task - a part of the Job, often making use of an executable	a part of Build Step
Builder task	Build runner step
Agent	Build Agent
(Agent) capability	(Agent) Parameter
(Job) requirement	Agent Requirement

## Supported Platforms and Environments

This page covers software-related environments TeamCity works with. For hardware-related notes, see [this section](#).

In this section:

- [Platforms \(Operating Systems\)](#)
  - [TeamCity Server](#)
  - [Build Agents](#)
    - [Stop Build Functionality](#)
  - [Windows Tray Notifier](#)
- [Web Browsers](#)
- [Build Runners](#)
  - Supported Java build runners
  - Supported .Net platform build runners
  - Other runners
- [Testing Frameworks](#)
- [Version Control Systems](#)
  - [Checkout on Agent](#)
  - [Labeling Build Sources](#)
  - [Remote Run on Branch](#)
  - [Feature Branches](#)
  - [VCS Systems Supported via Third Party Plugins](#)
- [Cloud Agents Integration](#)
- [VCS Hosting Services Integration](#)
- [Issue Tracker Integration](#)
- [IDE Integration](#)
  - [Remote Run and Pre-tested Commit](#)
  - [Code Coverage](#)
- [Supported Databases](#)

## Platforms (Operating Systems)

### TeamCity Server

Core features of TeamCity server are platform-independent. See [considerations](#) on choosing server platform. TeamCity server is a web application that runs within a capable J2EE servlet container.

Requirements:

- Java (JRE), see configuration [notes](#). Supported are:
  - Oracle Java 8 and updates (included in the Windows .exe distribution). 32 or 64 bit (64 bit is recommended for production)
  - OpenJDK 8. 32 or 64 bit

For the .war distribution, note that the .war distribution is going to be discontinued, and it is highly recommended to use the .tar.gz distribution, which has Tomcat bundled. It is not advised to customize Tomcat settings unless absolutely necessary. If you still want to use the .war distribution, note that:

- TeamCity is tested only under Apache Tomcat 8.5, earlier Tomcat versions are not supported.
- TeamCity is meant to support J2EE Servlet 3.0+ and JSP 2.2+ container based on Apache Jasper.
- TeamCity may not work properly if the [Apache Portable Runtime](#) is enabled in Tomcat.

Generally, all the recent versions of Windows, Linux and macOS are supported. If you find any compatibility issues with any of the operating systems please make sure to [let us know](#).

The TeamCity server is tested under the following operating systems:

- Linux (Ubuntu, Debian, RedHat, SUSE, and others)
- macOS
- Windows 7/7x64
- Windows Server 2008, 2012, 2016
- Server Core installation of Windows Server 2016
- Windows 10
  - under the Tomcat 8.5 web application server.

Reportedly works without known issues on:

- Windows 7+
- Windows Server 2008 R2
- Solaris
- FreeBSD
- IBM z/OS
- HP-UX

Note that Windows XP/XP x64 are not supported.

## Build Agents

The TeamCity Agent is a standalone Java application.

Requirements:

- Java (JRE), see configuration [notes](#). Supported are:
  - Oracle Java 6-10, Java 8 is recommended and is included in the Windows .exe distribution. 32 or 64-bit
  - OpenJDK 6-10 is supported, but Oracle JDK 8 is recommended. 32 or 64-bit

TeamCity agent is tested under the following operating systems:

- Linux
- macOS
- Windows 7/7x64
- Windows 10
- Windows Server 2003/2008, 2012, 2016
- Server Core installation of Windows Server 2016

Reportedly works on:

- Windows XP/XP x64
- Windows 2000 (interactive mode only)
- Solaris
- FreeBSD
- IBM z/OS
- HP-UX

## Stop Build Functionality

Build stopping is supported on:

- Windows 7/7x64/10
- Linux on x86, x64, PPC and PPC64 processors
- macOS on Intel and PPC processors
- Solaris 10 on x86, x64 processors

## Windows Tray Notifier

Windows 7/7x64/10 with one of the supported versions of Internet Explorer.

## Web Browsers

The TeamCity Web Interface is mostly W3C-compliant, so just about any modern browser should work well with TeamCity. The following browsers have been specifically tested and reported to work correctly in the recent versions of:

- Google Chrome
- Mozilla Firefox
- Safari under Mac
- Microsoft Edge

- Microsoft Internet Explorer 10+. Note that support for Microsoft Internet Explorer 10 will be discontinued in the next TeamCity version.
- Opera 15+

## Build Runners

TeamCity supports a wide range of build tools, enabling both Java and .Net software teams to build their projects.

### Supported Java build runners

- [Ant](#) 1.6-1.10 TeamCity 2018.1 comes bundled with Ant 1.9.11.
- [Maven](#) versions 2.0.x, 2.x, 3.x (known at the moment of the TeamCity release). Java 1.5 and higher is supported. TeamCity comes bundled with Maven 2.2.1, 3.0.5, 3.1.1, 3.2.5.
- [IntelliJ IDEA Project](#) runner (requires Java 8)
- [Gradle](#) (requires Gradle 0.9-rc-1 or higher)
- [Java Inspections](#) and [Java Duplicates](#) based on IntelliJ IDEA (requires Java 8)

### Supported .Net platform build runners

- [MSBuild](#). Requires .Net Framework or Mono installed on the build agent. Microsoft Build Tools 2013, 2015 and 2017 are also supported.
- [NAnt](#) versions 0.85 - 0.91 alpha 2. Requires .Net Framework or Mono installed on the build agent.
- [Microsoft Visual Studio Solutions](#) (2003 - 2015 and 2017). Requires a corresponding version of Microsoft Visual Studio installed on the build agent.
- [FxCop](#). Requires FxCop installed on the build agent.
- [Duplicates Finder for C# and VB.NET code](#) based on [ReSharper Command Line Tools](#). Supported languages are C# up to version 4.0 and VB.NET version 8.0 - 10.0. Requires .Net Framework 4.0+ installed on the build agent.
- [Inspections for .NET](#) based on [ReSharper Command Line Tools](#). Requires .Net Framework 4.0+ installed on the build agent.
- [.NET Process Runner](#) for running any .NET application (requires .NET installed on the build agent)
- [NuGet](#) runners under Windows, Linux macOS. Require NuGet.exe Command Line tool installed on the agents. Supported NuGet versions under Windows are 1.4+.
  - Windows: NuGet versions prior to 2.8.6 require .Net Framework 4.0+ installed on the build agent
  - Windows: NuGet 2.8.6 and later requires .NET 4.5
  - Linux and macOS: require [Mono](#) 4.4.2+ and NuGet CLI 3.2+ installed on the agent
- [.NET CLI \(dotnet\)](#). Requires the [.NET Core SDK](#) installed on build agents.

### Other runners

- [Rake](#)
- [Command Line](#) Runner for running any build process using a shell script
- [PowerShell](#) versions 1.0 through 5.0 are supported.
- [Xcode](#) versions 3-8 are [supported](#) (requires Xcode installed on the build agent)

## Testing Frameworks

- [JUnit](#) 3.8.1+, 4.x
- [NUnit](#) 2.2.10, 2.4.x, 2.5.x, 2.6.x, 3.0.x are supported (dedicated build runner).
- [TestNG](#) 5.3+
- [MSTest](#) 8.x-12.x, 14.x, 15.x and VSTest are supported by the [Visual Studio Tests](#) runner; requires the appropriate Microsoft Visual Studio edition or Visual Studio Test Agent installed on the build agent.
- [MSpec](#) (requires MSpec installed on the build agent)

## Version Control Systems

- [Git](#) (for automatic "git gc" support requires Git client installed on the server in order to perform maintenance of Git clones, latest version is recommended)
- [Subversion](#) (server versions 1.4-1.9 and higher as long as the protocol is backward compatible).
- [Perforce](#) (requires a Perforce client installed on the TeamCity server). Check [compatibility issues](#).
- [Team Foundation Server](#) 2005, 2008, 2010, 2012, 2013, 2015, 2017 are supported.
- [Mercurial](#) (requires the Mercurial "hg" client v1.5.2+ installed on the server)
- [CVS](#)
- [SourceGear Vault](#) 6 and 7 (requires the Vault command line client libraries installed on the TeamCity server)

- [Borland StarTeam](#) 6 and up (the StarTeam client application must be installed on the TeamCity server)
- [IBM Rational ClearCase](#), Base and UCM modes (requires the ClearCase client installed and configured on the TeamCity server)
- [Microsoft Visual SourceSafe](#) 6 and 2005 (requires a SourceSafe client installed on the TeamCity server, available only on Windows platforms)

For support for other VCS please check [external plugins](#) available.

## Checkout on Agent

The requirements noted are for agent environment and are additional to those for the server listed above.

- Git (git client version 1.6.4+ must be installed on the agent, latest version is recommended)
- Subversion (working copies in the Subversion 1.4-1.8 format are supported)
- Perforce (a Perforce client must be installed on the TeamCity agent machine)
- Team Foundation Server 2005-2015, 2017 are supported.
- Mercurial (the Mercurial "hg" client v1.5.2+ must be installed on the TeamCity agent machine)
- CVS
- IBM Rational ClearCase (the ClearCase client must be installed on the TeamCity agent machine)

## Labeling Build Sources

- Git
- Subversion
- Perforce
- Team Foundation Server
- Mercurial
- CVS
- SourceGear Vault
- Borland StarTeam
- ClearCase

## Remote Run on Branch

- Git
- Mercurial

## Feature Branches

- Git
- Mercurial

## VCS Systems Supported via Third Party Plugins

- AccuRev
- Bazaar
- PlasticSCM (related [details](#))

## Cloud Agents Integration

- Amazon EC2
- VMWare vSphere

Check also other non-bundled and third-party [cloud integration plugins](#).

## VCS Hosting Services Integration

- GitHub / GitHub Enterprise
- Bitbucket Cloud
- Visual Studio Team Services

## Issue Tracker Integration

- JetBrains YouTrack 1.0 and later (tested with the latest version).
- Atlassian Jira 4.4 and later (all major features also reportedly worked for version 4.2).
- Bugzilla 3.0 and later (tested with Bugzilla 5.0.1).
- GitHub
- Bitbucket
- TFS (Microsoft Visual Studio Team Foundation Server 2010-2018, and Visual Studio Team Services are supported)  
Additional requirements are listed in [Integrating TeamCity with Issue Tracker](#).

Links to issues of any issue tracker can also be recognized in change comments using [Mapping External Links in Comments](#).

## IDE Integration

TeamCity provides productivity plugins for the following IDEs:

- [Eclipse](#): Eclipse versions 3.8 and 4.2-4.6 are supported. Eclipse must be run under JDK 1.5+.
- [IntelliJ Platform Plugin](#): compatible with IntelliJ IDEA 15.0.x - 2017.3.x (Ultimate and Community editions); as well as other IDEs based on the same version of the platform, including JetBrains RubyMine 6.3+, JetBrains PyCharm 3.1+, JetBrains PhpStorm/WebStorm 7.1+, AppCode 2.1+. See [more information](#) on compatibility.
- [Microsoft Visual Studio](#) 2010, 2012, 2013, 2015, 2017 is supported by the TeamCity Visual Studio Add-in shipped as a part of ReSharper Ultimate. Installed .NET Framework is required.

## Remote Run and Pre-tested Commit

[Remote Run](#) and [Pre-tested commit](#) functionality is available for the following IDEs and version control systems:

IDE	Supported VCS
Eclipse	<ul style="list-style-type: none"> <li>• Subversion 1.7-1.8 via Subclipse and Subversive Eclipse integration plugins or SvnKit.</li> <li>• Subversion 1.4-1.7 via Subclipse and Subversive Eclipse integration plugins.</li> <li>• Perforce (P4WSAD 2009.2 - 2010.1, P4Eclipse 2010.1 - 2015.1)</li> <li>• Git (the EGit 2.0+ Eclipse integration plugin)</li> <li>• CVS</li> <li>• ClearCase (the client software is required)</li> <li>• see also</li> </ul>
IntelliJ IDEA Platform *)	<ul style="list-style-type: none"> <li>• Subversion</li> <li>• Perforce</li> <li>• Git (remote run only)</li> <li>• Team Foundation Server (since TeamCity 2017.1; the plugin update is required)</li> <li>• ClearCase</li> </ul>
Microsoft Visual Studio	<ul style="list-style-type: none"> <li>• Subversion 1.4-1.9 (the command-line client is required)</li> <li>• Team Foundation Server 2005 and later. Installed Team Explorer is required.</li> <li>• Perforce 2008.2 and later (the command-line client is required)</li> </ul>

\*) Supported only with the VCS integrations bundled with the IDEs by JetBrains

## Code Coverage

IDE	Supported Coverage Tool
Eclipse	<a href="#">IDEA</a> and <a href="#">EMMA</a> code coverage
IntelliJ IDEA Platform	<a href="#">IDEA</a> , <a href="#">EMMA</a> and <a href="#">JaCoCo</a> code coverage
Microsoft Visual Studio	JetBrains dotCover coverage. Requires <a href="#">JetBrains dotCover</a> installed in Microsoft Visual Studio

## Supported Databases

See more at [Setting up an External Database](#)

- HSQLDB 1.8/2.x (internal, used by default) The internal database suits evaluation purposes only; we strongly recommend using an external database in a production environment.
- MySQL 5.0.33+, 5.1.49+, 5.5+, 5.6+, 5.7+ (Please note that due to bugs in MySQL, versions 5.0.20, 5.0.22 and 5.1 up to 5.1.48 are not compatible with TeamCity)
- Microsoft SQL Server 2005, 2008, 2012, 2014 and 2016 (including Express editions), SQL Azure; SSL connections

- support might require these updates.
- PostgreSQL 8.2 and newer
- Oracle 10g and newer (TeamCity is tested with [driver](#) version 12.1.0.1)

## Testing Frameworks

TeamCity provides out-of-the-box support for a number of testing frameworks.

In order to reduce feedback time on the test failures, TeamCity provides support for on-the-fly tests reporting where possible. On-the-fly tests reporting means that the tests are reported in the TeamCity UI as soon as they are run not waiting for the build to complete.

TeamCity directly supports the following testing frameworks:

- JUnit and TestNG for the following runners:
  - Ant (when tests are run by the `junit` and `testng` tasks directly within the script, TeamCity reports tests on the fly)
  - Maven2 (when tests are run by [Maven Surefire plugin](#) or [Maven Failsafe plugin](#), tests reporting occurs after each module test run finish)
  - [IntelliJ IDEA](#) project (when run with appropriate IDEA run configurations. Note that such run configurations should be shared in the IDE and related files should be checked in to the version control)
- NUnit for the following runners:
  - The NAnt (`nunit2` task)
  - The MSBuild ([NUnit community](#) or [NUnitTeamCity](#) tasks)
  - Microsoft Visual Studio Solution runners (2003, 2005, 2008, 2010, 2012, 2013, and Visual Studio 2015)
  - Any runner provided [TeamCity Addin for NUnit](#) is installed
- MSTest 2005, 2008, 2010, 2012, 2013, and 2015 (On-the-fly reporting is not available due to MSTest limitations)
- VSTest 2012, 2013, 2015
- MSpec

Ruby testing frameworks, `Test::Unit`, `Test-Spec`, `Shoulda`, `RSpec`, `Cucumber`, are supported by the TeamCity `Rake` runner. The `minitest` framework requires the `minitest-reporters` gem to be additionally installed.

There are also testing frameworks that have embedded support for TeamCity. e.g. `Gallio` and `xUnit`.

See also [plugins](#).

Also, you can import test run XML reports of supported formats with [XML Report Processing](#).

### Custom Testing Frameworks

If there is no TeamCity support yet for your testing framework, you can report tests progress to TeamCity from the build via [service messages](#) or generate one of the supported [XML reports](#) in the build.

Also, see [notes](#) on integrating with various reporting/metric tools.

See also:

[Concepts: Build State | Build Runner](#)  
[User's Guide: Viewing Tests and Configuration Problems](#)  
[Administrator's Guide: NUnit Support | MSTest Support | NAnt](#)

## Code Quality Tools

TeamCity comes bundled with a number of tools capable of analyzing the quality of your code and reporting the obtained data. If you are using the tools which are currently not supported, TeamCity can be configured to run them and display their report results.

On this page:

- Bundled Tools
  - Java Tools
    - [IntelliJ IDEA-powered Code Analysis Tools](#)
    - Code Coverage tools
  - .Net Tools
    - [ReSharper-powered Tools](#)
    - Code Coverage
- Reporting External Tools Results in TeamCity

- Supported Report Formats
  - Including HTML Reports
  - Importing Code Coverage Results
- Integration with External Tools

## Bundled Tools

Generally, the tools are configured as [build runners](#) and the results are displayed on the [Build Results](#) page as well as in the IDE for some of the tools.

You can also configure builds to fail based on the results and view the trends as statistics charts.

## Java Tools

### IntelliJ IDEA-powered Code Analysis Tools

These are available when you have an IntelliJ IDEA project (.idea directory or .ipr file) or a Maven project file (pom.xml) checked into your version control.

- [Inspections \(IntelliJ IDEA\)](#) runs [IntelliJ IDEA inspections](#) in TeamCity. These include more than 600 Java, HTML, CSS, JavaScript inspections.
- [Duplicates Finder \(Java\)](#) provides a report on the discovered repetitive blocks of code.

## Code Coverage tools

These are configured in the dedicated sections of the build runners.

- [IntelliJ IDEA code coverage](#) is supported for [Ant](#), [IntelliJ IDEA Project](#), [Gradle](#) or [Maven](#) build runners.
- [EMMA coverage](#) supports [Ant](#) build runner.
- [JaCoCo coverage](#) supports [Ant](#), [IntelliJ IDEA Project](#), [Gradle](#) and [Maven](#) build runners.

## .Net Tools

### ReSharper-powered Tools

These are available if you use Visual Studio.

- [Inspections \(ReSharper\)](#) gathers results of [JetBrains ReSharper Code Inspections](#) in your C#, VB.NET, XAML, XML, ASP.NET, JavaScript, CSS and HTML code.
- [Duplicates Finder \(ReSharper\)](#) provides a report on the discovered repetitive blocks of C# and VB.NET code.
- [FxCop](#) uses Microsoft [FxCop](#) pre-installed on a build agent.

## Code Coverage

The following code coverage tools are supported for [.NET Process Runner](#), [MSBuild](#), [MSTest](#), [NAnt](#) and [NUnit](#) build runners:

- [JetBrains dotCover](#)
- [NCover](#)
- [PartCover](#)

For the [.NET CLI \(dotnet\)](#) runner and with [NUnit](#) version 3.x the only supported coverage tool is [JetBrains dotCover](#).

## Reporting External Tools Results in TeamCity

If you need to use non-bundled tools, you can use TeamCity to import their results and display them in the TeamCity UI.

### Supported Report Formats

The external tool reports are supported via the [XML Report Processing](#) build feature. See the list of [supported reports](#).

### Including HTML Reports

If your reporting tool is not supported by TeamCity directly, you can make it produce reports in the HTML format via a build script and add a build results [report tab in TeamCity](#).

### Importing Code Coverage Results

You can also [import code coverage results in TeamCity](#).

## Integration with External Tools

TeamCity can also [be integrated](#) with external build tools or tools generating some report/providing code metrics which are not yet supported by TeamCity. The integration tasks involved are collecting the data in the scope of a build and then reporting the data to TeamCity.

# Installation and Upgrade

In this part you will learn how to install and upgrade TeamCity.

- [Installation](#)
- [Upgrade Notes](#)
- [Upgrade](#)
- [TeamCity Maintenance Mode](#)
- [Setting up an External Database](#)
- [Migrating to an External Database](#)

## Installation

If you are upgrading your existing TeamCity installation, please refer to [Upgrade](#).

### Check the System Requirements

Before you install TeamCity, please familiarize yourself with [Supported Platforms and Environments](#). Additionally, read the [hardware requirements for TeamCity](#). However, note that these requirements differ significantly depending on the server load and the number of builds run.

### Select TeamCity Installation Package

The TeamCity installation package is identical for both Professional and Enterprise Editions and is available on the <http://www.jetbrains.com/teamcity/download/> page.

The following options are available:

Target	Option	Note
--------	--------	------

Windows	TeamCity<version number>.exe	Executable Windows installer bundled with Tomcat and Java 1.8 JRE.
Linux, macOS, Windows	TeamCity<version number>.tar.gz	Archive for manual installation bundled with Tomcat servlet container
J2EE container	TeamCity<version number>.war	Package for installation into an existing J2EE container (not recommended, use .tar.gz instead)
Docker (Linux, Windows)	Docker	Official JetBrains TeamCity server Docker image
AWS	Run on AWS	Official CloudFormation template to launch TeamCity Stack

## Install Additional Build Agents

Although the TeamCity server in `.exe` and `.tar.gz` distributions is installed with a default build agent that runs on the same machine as the server, this setup may result in degraded TeamCity web UI performance, and if your builds are CPU-intensive, it is recommended to install build agents on separate machines or ensure that there is enough CPU/memory/disk throughput on the server machine.

To setup additional build agents, follow the [instructions](#).

See also:

[Installation and Upgrade: Installing and Configuring the TeamCity Server | Setting up and Running Additional Build Agents](#)

## Installing and Configuring the TeamCity Server

This page covers a new TeamCity server installation.  
For upgrade instructions, please refer to [Upgrade](#).

To install a TeamCity server, perform the following:

- Choose the appropriate TeamCity distribution (.exe, .tar.gz or [Docker image](#)) based on the details below.



You can also opt to run TeamCity stack on AWS, in which case proceed to the instructions on [this page](#).

- [Download the distribution](#)
- Review [software requirements](#) and [hardware requirements notes](#) and [platform selection](#)
- Review TeamCity [Licensing Policy](#)
- Install and configure the TeamCity server per instructions below

This page covers:

- [Installing TeamCity Server](#)
  - [Installing TeamCity via Windows installation package](#)
  - [Installing TeamCity bundled with Tomcat servlet container \(Linux, macOS, Windows\)](#)
  - [Installing TeamCity into Existing J2EE Container](#)
  - [Unattended TeamCity server installation](#)
  - [Using another Version of Tomcat](#)
- [Starting TeamCity server](#)
  - [Autostart TeamCity server on macOS](#)
- [Installation Configuration](#)
  - [Troubleshooting TeamCity Installation](#)
  - [Changing Server Port](#)
  - [Changing Server Context](#)
  - [Java Installation](#)
  - [Using 64 bit Java to Run TeamCity Server](#)
  - [Setting Up Memory settings for TeamCity Server](#)
- [Configuring TeamCity Server](#)
  - [Configuring TeamCity Data Directory](#)
  - [Editing Server Configuration](#)
  - [Configuring Server for Production Use](#)

## Installing TeamCity Server

After you obtained the TeamCity installation package, proceed with corresponding installation instructions:

- [Windows .exe distribution](#) - the executable which provides the installation wizard for Windows platforms and allows installing the server as a Windows service;
- [.tar.gz distribution](#) - the archive with a "portable" version suitable for all platforms;
- [Docker image](#) - check the instructions at the [image page](#);
- [.war distribution](#) - for experienced users who want to run TeamCity in a separately installed Web application server. Please consider using .tar.gz distribution instead. .war is not recommended to use unless really required (please [let us know](#) the reasons).

Compared to the .war distribution, the .exe and .tar.gz distributions:

- include a Tomcat version which TeamCity is tested with, so it is known to be a working combination. This might not be the case with an external Tomcat.
- define additional JRE options which are usually recommended for running the server
- have the [teamcity-server startup script](#) which provides several convenience options (e.g. separate environment variable for memory settings) and configures TeamCity correctly (e.g. log4j configuration)
- (at least under Windows) provide better error reporting for some cases (like a missing Java installation)
- under Windows, allow running TeamCity as a service with the ability to use the same configuration as if run from the console
- come bundled with a build agent distribution and single startup script which allows for easy TeamCity server evaluation with one agent
- come bundled with the devPackage for [TeamCity plugin development](#).
- may provide more convenience features in the future

After installation, the TeamCity web UI can be accessed via a web browser. The default addresses are <http://localhost/> for Windows distribution and <http://localhost:8111/> for tar.gz distribution.

If you cannot access the TeamCity web UI after successful installation, please refer to the [#Troubleshooting TeamCity Installation Issues](#) section.



The build server and one build agent will be installed by default for Windows, Linux or macOS. If you need more build agents, refer to the [Installing Additional Build Agents](#) section.

**!** During the server setup you can select either an internal database or an existing external database. By default, TeamCity uses an HSQLDB database that does not require configuring. This database suites the purposes of testing and evaluating the system.  
For production purposes, using a standalone external database is recommended.

## Installing TeamCity via Windows installation package

For the Windows platform, run the executable file and follow the installation instructions. You have options to install the TeamCity web server and one build agent that can be run as a Windows service.

If you opted to install the services, use the standard Windows Services applet to manage the service. Otherwise, use standard [scripts](#).

If you did not change the default port (80) during the installation, the TeamCity web UI can be accessed via "http://localhost/" address in a web browser running on the same machine where the server is installed. Please note that port 80 can be used by other programs (e.g. Skype, or other web servers like IIS). In this case you can specify another port during the installation and use "http://localhost:<port>/" address in the browser.

**i** If you want to edit the TeamCity server's service parameters, memory settings or system properties after the installation, refer to the [Configuring TeamCity Server Startup Properties](#) section.

### **!** Service account

Make sure the user account specified for the service has:

- log on as service right ([related Microsoft page](#))
- write permissions for the [TeamCity Data Directory](#),
- write permissions for the [TeamCity Home](#), i.e. directory where TeamCity was installed,
- all the necessary permissions to work with the source controls used. This includes:
  - access to Microsoft Visual SourceSafe database (if [Visual SourceSafe](#) integration is used).
  - the user, under whose account the TeamCity server service runs, and ClearCase view owner are the same (if the [ClearCase](#) integration is used).

By default, the Windows service is installed under the SYSTEM account. To change it, use the Services applet (Control Panel | Administrative Tools | Services)

## Installing TeamCity bundled with Tomcat servlet container (Linux, macOS, Windows)

Review [software requirements](#) before the installation.

Unpack the TeamCity<version number>.tar.gz archive (for example, using the tar xfz TeamCity<version number>.tar.gz command under Linux, or the WinZip, WinRar or similar utility under Windows).

Please use GNU tar to unpack (for example, Solaris 10 tar is reported to truncate too long file names and may cause a ClassNotFound exception when using the server after such unpacking. Consider getting GNU tar at [Solaris packages](#) or using the gtar xfz command).

Ensure you have JRE or JDK installed and the JAVA\_HOME environment variable is pointing to the Java installation directory. Oracle Java 8 JDK is required.

## Installing TeamCity into Existing J2EE Container

It is not recommended to use .war distribution. Use the [TeamCity .tar.gz](#) distribution (bundled with Tomcat web server) instead. If you have important reasons to deploy TeamCity into existing web server and want to use .war distribution, please let us know the reasons.

1. Make sure your web application server is stopped.
2. Copy the downloaded TeamCity<version number>.war file into the web applications directory of your J2EE container under the TeamCity.war name (the name of the file is generally used as a part of the URL) or deploy the .war following the documentation of the web server. Please make sure there is no other version of TeamCity deployed (e.g. do not preserve the old TeamCity web application directory under the web server applications directory).
3. Ensure the TeamCity web application gets sufficient amount of [memory](#). Please increase the memory accordingly if you have other web applications running in the same JVM.
4. If you are deploying TeamCity to the Tomcat container, please add useBodyEncodingForURI="true" attribute to the main Connector tag for the server in the Tomcat/conf/server.xml file.
5. If you are deploying TeamCity to Jetty container version >7.5.5 (including 8.x.x), please make sure the system

- `property org.apache.jasper.compiler.disablejsr199 is set to true`
6. Ensure that the servlet container is configured to unpack the deployed war files. Though for most servlet containers it is the default behavior, for some it is not (e.g. Jetty version >7.0.2) and should be explicitly configured. TeamCity is not able to work from a packed .war: if started this way, there will be a note on this the logs and UI .
  7. Configure the appropriate [TeamCity Data Directory](#) to be used by TeamCity.



Note that it is recommended to start with an empty TeamCity Data Directory. After completing the installation and performing the first TeamCity server start, the required data (e.g. [database settings file](#)) can be moved to the directory.

8. Check/configure the TeamCity [logging properties](#) by specifying the `log4j.configuration` and `teamcity_logs` internal properties.
9. Restart the server or deploy the application via the servlet container administration interface and access `http://server:port/TeamCity/`, where "TeamCity" is the name of the war file.

TeamCity J2EE container distribution is tested to work with Tomcat 7 servlet container. (See also [Supported Platforms and Environments#The TeamCity Server](#))



If you're using Tomcat J2EE container, make sure the [Apache Portable Runtime](#) feature of this container is disabled (actually it is disabled by default). Otherwise due to issues in the [Apache Portable Runtime](#), TeamCity may not work properly.

## Unattended TeamCity server installation

For automated server installation, use .tar.gz distribution.

Typically, you will need to unpack it and make the script perform the steps noted in [Configuring Server for Production Use](#) section.

If you want to get a pre-configured server right away, put files from a previously configured server into the Data Directory. For each new server you will need to ensure it points to a new database (configured in `<Data Directory>\config\database.properties`) and change `<Data Directory>\config\main-config.xml` file not to have "uuid" attribute in the root XML element (so new one can be generated) and setting appropriate value for "rootURL" attribute.

## Using another Version of Tomcat

If you want to use another version of Tomcat web server instead of the one bundled in .tar.gz and .exe distributions), you have the choices of whether to use the [.war TeamCity distribution](#) (not recommended) or perform the Tomcat upgrade/patch for TeamCity installed from the .exe or .tar.gz distributions.

For the latter, you might want to:

- backup the current [TeamCity home](#)
- delete/move out the directories from the [TeamCity home](#) which are also present in the Tomcat distribution
- unpack the Tomcat distribution into the [TeamCity home directory](#)
- copy TeamCity-specific files from the previously backed-up/moved directories to the [TeamCity home](#). Namely:
  - files under `bin` which are not present in the Tomcat distribution
  - review differences between the default Tomcat `conf` directory and one from TeamCity, update Tomcat files with TeamCity-specific settings (`teamcity-*` files, and portions of `server.xml`)
  - delete the default Tomcat `webapps/ROOT` directory and replace it with the one provided by TeamCity

## Starting TeamCity server

If TeamCity server is installed as a Windows service, follow the usual procedure of starting and stopping services.

If TeamCity is installed using the .exe or .tar.gz distributions, the TeamCity server can be started and stopped by the `teamcity-server` scripts provided in the `<TeamCity home>/bin` directory.

- (evaluation only) To start/stop the TeamCity server and one default agent at the same time, use the `runAll` script, e.g.:
  - Use `runAll.bat start` to start the server and the default agent
  - Use `runAll.bat stop` to stop the server and the default agent
- To start/stop the TeamCity server only, use the `teamcity-server` scripts and pass the required parameters. Start the script without parameters to see the usage instructions.  
The `teamcity-server` scripts support the following options for the `stop` command:
  - `stop n` - Sends the stop command to the TeamCity server and waits up to n seconds for the process to end.
  - `stop n -force` - Sends the stop command to the TeamCity server, waits up to n seconds for the process to end, and terminates the server process if it did not stop.

 The TeamCity server will restart automatically if the Java process of the server crashes since TeamCity 2017.2.

By default, TeamCity runs on <http://localhost:8111> and has one registered build agent that runs on the same computer.

See the information [below](#) for changing the server port.

If you need to pass special properties to the server, refer to [Configuring TeamCity Server Startup Properties](#).

If TeamCity is installed into an existing web server (.war distribution), start the server according to its documentation. Make sure you configure TeamCity-specific logging-related properties and pass suitable [memory options](#).

## Autostart TeamCity server on macOS

Starting up TeamCity server on macOS is quite similar to starting Tomcat on macOS.

1. Install TeamCity and make sure it works if started from the command line with `bin/teamcity-server.sh start`. We'll assume that TeamCity is installed in the `/Library/TeamCity` folder
2. Create the `/Library/LaunchDaemons/jetbrains.teamcity.server.plist` file with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>WorkingDirectory</key>
<string>/Library/TeamCity</string>
<key>Debug</key>
<false/>
<key>Label</key>
<string>jetbrains.teamcity.server</string>
<key>OnDemand</key>
<false/>
<key>KeepAlive</key>
<true/>
<key>ProgramArguments</key>
<array>
<string>/bin/bash</string>
<string>--login</string>
<string>-c</string>
<string>bin/teamcity-server.sh run</string>
</array>
<key>RunAtLoad</key>
<true/>
<key>StandardErrorPath</key>
<string>logs/launchd.err.log</string>
<key>StandardOutPath</key>
<string>logs/launchd.out.log</string>
</dict>
</plist>
```

3. Test your file by running `launchctl load /Library/LaunchDaemons/jetbrains.teamcity.server.plist`. This command should start the TeamCity server (you can see this from `logs/teamcity-server.log` and in your browser).
4. If you don't want TeamCity to start under the root permissions, specify the `UserName` key in the plist file, e.g.:

```
<key>UserName</key>
<string>teamcity_user</string>
```

The TeamCity server will now start automatically when the machine starts. To configure automatic start of a TeamCity Build Agent, see [the dedicated section](#).

## Installation Configuration

### Troubleshooting TeamCity Installation

Upon successful installation, the TeamCity server web UI can be accessed via a web browser. The default address that can be used to access TeamCity from the same machine depends on the installation package and installation options. (Port 80 is used for Windows installation, unless another port is specified, port 8111 for .tar.gz installation unless not changed in the server configuration).

If the TeamCity web UI cannot be accessed, please check:

- the "TeamCity Server" service is running (if you installed TeamCity as a Windows service);
- the TeamCity server process (Tomcat) is running (it is a java process run in the `<TeamCity home>/bin` directory);
- the console output if you run the server from a console,
- the `teamcity-server.log` and other files in the `<TeamCity home>\logs` directory for error messages.

One of the most common issues with the server installation is using a port that is already used by another program. See [below](#) on changing the default port.

## Changing Server Port

If you use the TeamCity server Windows installer, you can set the port to be used during installation. If you use the .war distribution, refer to the manual of the application server used.

Use the following instructions to change the port if you use the .tar.gz distribution.

If another application uses the same port as the TeamCity server, the TeamCity server (Tomcat server) won't start and this will be identified by "Address already in use" errors in the server logs or server console.

To change the server port, in the `<TeamCity Home>/conf/server.xml` file, change the port number in the not commented "`<Connector>`" XML node (here the port number is 8111):

```
<Connector port="8111" ...
```

To apply the changes, restart the server. If the server was working with the old port previously, you would need to change the port in all the stored URLs of the server (browser bookmarks, agents' `serverUrl` property, URL in user's IDEs, "Server URL" setting on the "Administration | Global Settings" page).

If you run another Tomcat server on the same machine, you might need to also change other Tomcat server service ports (search for "port=" in the `server.xml` file).

If you want to use the `https://` protocol, it should be enabled separately and the process is not specific to TeamCity, but rather for the web server used (Tomcat by default). See also [Using HTTPS to access TeamCity server](#)

## Changing Server Context

By default, the TeamCity server is accessible under the root context of the server address (e.g. `http://localhost:8111/`). To make it available under a nested path instead (e.g. `http://localhost:8111/teamcity/`), you need to:

- stop the TeamCity server;
- rename the `<TeamCity home>\webapps\ROOT` directory to the `<TeamCity home>\webapps\teamcity`;
- start the TeamCity server.



Note that after this change [automatic update](#) will be disabled for your installation and you will have to upgrade TeamCity [manually](#).

## Java Installation

The TeamCity server is a web application that runs in an J2EE application server (a JVM application). TeamCity server requires a Java SE JRE installation to run.

The TeamCity server requires JRE 8 to operate ([download page](#)), the agent can run with Java 6-10, but Java 8 is recommended. The recommended Oracle download is Java SE JDK.

It is recommended to use the 32-bit installation unless you need to [dedicate more memory](#) to TeamCity server. Please check the [64-bit Java notes](#) before upgrade.

If you configured any native libraries for use with TeamCity (like a .dll for using Microsoft SQL database Integrated Security option), you need to update the libraries to match the JVM x86/x64 platform.

For TeamCity agent Java requirements, check [Setting up and Running Additional Build Agents](#).

The necessary steps to update the Java installation depend on the distribution used.

- if your TeamCity installation has a bundled JRE (there is the `<TeamCity Home>\jre` directory), update it by installing a newer JRE per installation instructions and copying the content of the resulting directory to replace the content of the existing `<TeamCity home>\jre` directory.



Note that on upgrade, TeamCity will overwrite the existing JRE with the bundled 32-bit version, so you'll have to update to the 64-bit JRE again after upgrade.

If you also run a TeamCity agent from the `<TeamCity home>\buildAgent` directory, install JVM (Java SDK) installation instead of JRE and copy content of JVM installation directory into `<TeamCity Home>\jre`.

- if there is no `<TeamCity Home>\jre` directory present, set `JRE_HOME` or `JAVA_HOME` environment variables to be available for the process launching the TeamCity server (setting global OS environment variables and system restart is recommended). The variables should point to the home directory of the installed JRE or JVM (Java SDK) respectively and if both are present, the installed `JRE` will be used.
- if you use the `.war` distribution, Java update depends on the application server used. Please refer to the manual of your application server.

## Using 64 bit Java to Run TeamCity Server

TeamCity server can run under both the 32- and 64-bit JVM.

It is recommended to use the 32-bit JVM unless you need to dedicate more than 1.2Gb of memory (via `-Xmx` JVM option) to the TeamCity process (see [details](#)) or your [database requirements](#) are different.

If you choose to use the 64-bit JVM, note that the memory usage is almost doubled when switching from the 32- to 64-bit JVM, so please make sure you specify at least twice as much memory as for 32-bit JVM, see [#Setting Up Memory settings for TeamCity Server](#).

To update to the 64-bit Java:

- update Java to be used by the server
- set [JVM memory options](#). It is recommended to set the following options for the 64-bit JVM: `-Xmx4g -XX:ReservedCodeCacheSize=350m`

## Setting Up Memory settings for TeamCity Server

As a JVM application, TeamCity only utilizes memory devoted to the JVM. The memory used by JVM usually consists of: heap (configured via `-Xmx`) and metaspace (limited by the amount of available native memory), internal JVM (usually tens of Mb), and OS-dependent memory features like memory-mapped files. TeamCity mostly depends on the heap memory and this settings can be configured for the TeamCity application manually by [passing `-Xmx`](#) (heap space) option to the JVM running the TeamCity server.

Once you start using TeamCity for [production](#) purposes or you want to load the server during evaluation, you should manually set the appropriate memory settings for the TeamCity server.

To change the memory settings, refer to [Configuring TeamCity Server Startup Properties](#), or to the documentation of your application server, if you run TeamCity using the `.war` distribution.

Generally this means setting `TEAMCITY_SERVER_MEM_OPTS` environment variable to the value like `-Xmx750m`



The Permanent Generation (PermGen) space [has been replaced with metaspace](#) memory allocation. It is recommended to remove the `-XX:MaxPermSize` JVM option from `TEAMCITY_SERVER_MEM_OPTS` environment variable, if previously configured.

If slowness, `OutOfMemory` errors occur, or you consistently see a memory-related warning in the TeamCity UI, increase the

setting to the next level.

- minimum setting (the 32-bit Java should be used (bundled in .exe distribution)): `-Xmx750m`
- recommended setting for medium server use (the 32-bit Java should be used): `-Xmx1024m`. Greater settings with the 32-bit Java can cause an OutOfMemoryError with "Native memory allocation (malloc) failed" JVM crashes or "unable to create new native thread" messages
- recommended setting for large server use (64-bit Java should be used): `-Xmx4g -XX:ReservedCodeCacheSize=350m`. These settings should be suitable for an installation with up to two hundreds of agents and thousands of build configurations. Custom plugins installed might require increasing the values defined via the Xmx parameter.
- maximum settings for large-scale server use (64-bit Java should be used): `-Xmx10g -XX:ReservedCodeCacheSize=512m`. Greater values can be used for larger TeamCity installations. However, generally it is not recommended to use values greater than 10g without consulting TeamCity support.

Tips:

- the 32-bit JVM can reliably work with up to 1Gb heap memory (`-Xmx1024m`). (This can be increased to `-Xmx1200m`, but JVM under Windows might crash occasionally with this setting.) If more memory is necessary, the 64-bit JVM should be used with not less than 2.5Gb assigned (`-Xmx2500m`). It's highly unlikely that you will need to dedicate more than 4Gb of memory to the TeamCity process.
- A rule of thumb is that the 64-bit JVM should be assigned twice as much memory as the 32-bit for the same application. If you switch to the 64-bit JVM, make sure you adjust the memory setting (`-Xmx`) accordingly. It does not make sense to switch to 64 bit if you dedicate less than the double amount of memory to the application.

The recommended approach is to start with initial settings and monitor for the percentage of used memory using the Administration | Diagnostics page. If the server uses more than 80% of memory consistently without drops for tens of minutes, that is probably a sign to increase the `-Xmx` memory value by another 20%.

## Configuring TeamCity Server



- If you have a lot of projects or build configurations, we recommend you avoid using the Default agent in order to free up the TeamCity server resources. The TeamCity Administrator can [disable](#) the default agent on the Agents page of the web UI.
- When changing the TeamCity data directory or database, make sure they do not get out of sync.

## Configuring TeamCity Data Directory

The default placement of the TeamCity data directory can be changed. See corresponding section: [TeamCity Data Directory](#) for details.

## Editing Server Configuration

After successful server start, any TeamCity page request will redirect to prompt for the server administrator username and password. Please make sure that no one can access the server pages until the administrator account is setup.

After administration account setup you may begin to create Project and Build Configurations in the TeamCity server. You may also want to configure the following settings in the Server Administration section:

- Server URL
- Email server address and settings
- Jabber server address and settings

## Configuring Server for Production Use

Out-of-the-box TeamCity server installation is suitable for evaluation purposes. For production use you will need to perform additional configuration which typically includes:

- Configuring correct [server port](#) (and access via https)
- Make sure server URL, email and (optionally) Jabber server settings are specified and are correct
- Configuring the server process for OS-dependent autostart on machine reboot
- Using reliable storage for [TeamCity Data Directory](#)
- [Using external database](#)
- [Configuring recommended memory settings](#), use "maximum settings" for active or growing servers
- Planning for regular [backups](#)
- Planning for regular [upgrades](#) to the latest TeamCity releases
- (since TeamCity 10.0.3) Consider adding the `"teamcity.installation.completed=true"` line into the `<TeamCity Home Directory>\conf\teamcity-startup.properties` file - this will prevent the server from creating an administrator user if no such user is found

Please also review the notes on configuring the server for performance and security notes.

See also:

[Installation and Upgrade: Setting up and Running Additional Build Agents](#)

## CSRF Protection

- General information
- Implications for reverse proxy configuration
- Implications for non-browser HTTP clients
- CSRF checks for HTTP request
  - Trusted domain
- Troubleshooting

### General information

CSRF protection in TeamCity has been implemented since TeamCity 2017.1 ([issue](#)). This protection implies a number of requirements on HTTP requests.

### Implications for reverse proxy configuration

When a TeamCity server has a reverse proxy in front of it (Nginx, IIS, Apache), this proxy should be configured to pass headers from the original request to the TeamCity server.

Origin and Referer headers must be passed unmodified, when present.

The Host header should be passed as well, and if it is not possible due to some reason, X-Forwarded-Host must be set to the value of the original Host header.

Here are our [recommendations for reverse proxy configuration for TeamCity](#).

## Implications for non-browser HTTP clients

Non-browser HTTP clients which reuse authentication for REST scripting by supplying the `TCSESSIONID` cookie with the request need to be updated to supply the `Origin` HTTP header with the same value as the host the request is being sent to.

## CSRF checks for HTTP request

When considering HTTP request safety from the TeamCity perspective, the following checks are sequentially made:

1. If an HTTP request is a non-modifying one (such as GET), it is considered safe
2. If an HTTP request has a secure CSRF token either in the parameter or in the HTTP header and this token matches the one stored in user session, it is considered safe.
3. If an HTTP request has the `Origin` header, it must match a host from a trusted domain (see below), otherwise, the request is rejected without further processing
4. If an HTTP request has the `Referer` header which matches the `Host` header or `X-Forwarded-Host` header, it is considered safe
5. If an HTTP request has the `X-Requested-With=XMLHttpRequest` header, it is considered safe
6. If an HTTP request uses basic authentication and there is no user in TeamCity session/cookies, the request is considered safe

## Trusted domain

TeamCity considers the following domains/hosts as trusted:

- Host header value
- X-Forwarded-Host header value (can be set separately in case of proxy configuration)
- One of the CORS origins, [configured](#) for REST access.

## Troubleshooting

When you face problems regarding CSRF protection in TeamCity, you can follow these steps:

- If you use a reverse proxy, make sure you correctly configure Host/Origin headers, as described above. In the meantime, you may want to add the public URL of your server to [CORS-enabled origins](#).
- You can temporarily disable CSRF protection at all by setting the `teamcity.csrf.origin.check.enabled=logOnly internal property`.
- Information about failed CSRF attempts are logged into TeamCity/logs/teamcity-auth.log files. For more detailed diagnostics of the requests, [enable debug-auth logging preset](#).

## Running TeamCity Stack in AWS

The 'Run on AWS' option for TeamCity available on the [JetBrains](#) site lets you run the TeamCity stack in AWS using the official CloudFormation template.

See also related [blog post](#)

- [Stack Overview](#)
- [Prerequisites](#)
- [Using Template](#)
- [Connecting to server and viewing logs](#)
- [Next Steps](#)
- [Upgrading TeamCity in AWS](#)

### Stack Overview

The stack includes:

- a CoreOS EC2 instance with the official TeamCity server of the specified version from Docker Hub
- one TeamCity Build Agent running as a separate container on the same instance.
- an RDS MySQL database

The official Docker images with the TeamCity server and build agent are used.

The server and the database are placed in their own VPC which is completely secure. The database allows only internal connections within the VPC and it's possible to connect to the Server via HTTP(s) or SSH only.

### Prerequisites

To create a TeamCity stack and connect to it, you will need

- an [EC2 KeyPair](#) in the same region as the TeamCity stack
- an installed SSH client to connect to the TeamCity server and view the logs

### Using Template

1. Specify the parameters provided by the template:

Setting	Description
Name	The name for your TeamCity server, set to test by default

EC2 instance Type	Specify <a href="#">the type of instance</a> for the TeamCity server EC2 instance
EC2 KeyPair	The existing EC2 KeyPair for SSH access to the TeamCity Server EC2 instance
TeamCity Version	By default, the template will create a TeamCity installation of the latest version. You can also specify the exact version number here, e.g. 2017.1.5, 2017.2
RDS Database Instance Type	Specify the type for the RDS MySQL instance used as the external database for TeamCity

2. Click Next. (Optional) In the dialog that appears, provide additional options if required.

3. Click Next, review your settings and click Create. No other actions are required. It takes about 15 minutes for the template to deploy the whole stack. Once the deployment is ready, you will see the TeamCity server endpoint in the Output section which points you to your TeamCity installation.

4. Access the TeamCity instance from your browser, create the administrator's account and start using your TeamCity.

### Connecting to server and viewing logs

To connect to the server's console, you need to use your instance private key:

```
ssh -i <path to private key\privatekey.pem> core@<server_IP_address>
```

To see the teamcity-agent.log or teamcity-server.log, just run the `docker logs` command for the desired container, e.g. for the server logs, run

```
docker logs teamcity-server
```

### Next Steps

Once you have TeamCity up and running, consider the following steps:

- Use the [EC2 integration](#) to run and connect more build agents to your server
- Configure TeamCity to use the S3 bucket as [external artifact storage](#).

### Upgrading TeamCity in AWS

To update TeamCity started from the CloudFormation template:

1. In the [AWS CloudFormation console](#), from the list of stacks, select the running TeamCity stack and use the [Update Stack option](#).
2. You will be redirected to the Select Template page: use the Current Template option and click Next.
3. On the template settings page, enter the TeamCity version you want to update to. Note that if you previously used the TeamCity version tagged 'latest', you will now need to provide the actual version number as the "latest" tag can be applied to the server only once.
4. Click Next, provide additional options if required, review the new settings and click Update. Once the Update is complete, access the TeamCity Web UI from the browser.
5. If required, provide the [Super User token](#): to obtain it, you need to connect to your server instance, get the TeamCity server log as described [above](#), and retrieve the maintenance token.
6. Wait for the server to upgrade, log in to the TeamCity server and wait for the agent to upgrade and connect to the server.

# Setting up and Running Additional Build Agents

This page covers:

- Prerequisites
  - Necessary OS and environment permissions
    - Network
    - Common
    - Windows
    - Linux
    - Build-related Permissions
  - Agent-Server Data Transfers
    - Unidirectional Agent-to-Server Communication
    - Bidirectional Communication
- Installing Additional Build Agents
  - Installing via Windows installer
  - Installing via Docker Agent Image
  - Installing via ZIP File
  - Installing via Agent Push
    - Remote Host Requirements
    - Installation
- Starting the Build Agent
  - Manual Start
  - Automatic Start
    - Automatic Agent Start under Windows
      - Build Agent as a Windows Service
    - Automatic Agent Start under Linux
    - Automatic Agent Start under macOS
      - Install and start build agent
      - Configure automatic build agent start
      - Configure a second build agent on macOS
- Stopping the Build Agent
- Configuring Java
  - Upgrading Java on Agents
- Installing Several Build Agents on the Same Machine

Before you can start customizing projects and creating build configurations, you need to configure [build agents](#). Please review the [agent-server communication](#) and [Prerequisites](#) section before proceeding with agent installation.



- If you install TeamCity bundled with a Tomcat servlet container, or use the TeamCity installer for Windows, both the server and one build agent are installed on the same machine. This is not a recommended setup for production purposes because of [security concerns](#) and since the build procedure can slow down the responsiveness of the web UI and overall TeamCity server functioning. If you need more build agents, perform the procedure described below.
- If you need the agent to run on a different operating system than the TeamCity server, perform the procedure described below.
- For production installations, it is recommended to adjust the [Agent's JVM parameters](#) to include the `-server` option.

## Prerequisites

### Necessary OS and environment permissions

Before the installation, please review the [Conflicting Software section](#). In case of any issues, make sure no conflicting software is used.

Please note that in order to run a TeamCity build agent, the user account used to run the Agent requires the following privileges:

#### Network

- An agent must be able to open connections to the server using the server address specified in the `serverUrl` property (usually the same URL as the server web UI). Specific URLs which can be used for requests to the server should not be limited. See also the recommended [reverse proxy settings](#).

#### Legacy bidirectional communication notes

By default [unidirectional](#) agent-to-server connection via polling protocol is used by TeamCity.

If you need to use legacy [bidirectional communication](#) (not recommended), in addition for the agent to server connections, the server must be able to open HTTP connections to the agent. The agent port is determined using the `ownPort` property of the `buildAgent.properties` file as the starting port (9090 by default, next port is used if the specified port is busy), and the following IP addresses are tried:

- the address specified in the `ownAddress` property of the `buildAgent.properties` file (if any)
- the source IP of the HTTP request received by the server when the agent establishes a connection to the server. If a proxying server is used, it must be [correctly configured](#).
- the addresses of the network interfaces on the agent machine

If the agent is behind NAT and cannot be accessed by any of addresses of the agent machine network interfaces, please specify the `ownAddress` property in the `buildAgent.properties` file.

Please ensure that any firewalls installed on the agent, server machine, or in the network and network configuration comply with these requirements.

#### Common

The agent process (java) should:

- be able to open outbound HTTP connections to the server address (the same address you use in the browser to view the TeamCity UI)
- have full permissions (read/write/delete) to the following directories recursively: `<agent home>` (necessary for automatic agent upgrade and agent tools support), `<agent work>`, `<agent temp>`, and agent system directory (set by `workDir`, `tempDir` and `systemDir` parameters in `buildAgent.properties` file)
- be able to launch processes (to run builds).

#### Windows

- Log on as a service (to run as Windows service)
- Start/Stop service (to run as Windows service, necessary for the agent upgrade to work, see also [Microsoft KB article](#))
- Debug programs (for take process dump functionality to work)
- Reboot the machine (for agent reboot functionality to work)
- To be able to [monitor performance](#) of a build agent run as a Windows [service](#), the user starting the agent must be a member of the Performance Monitor Users group



For granting necessary permissions for unprivileged users, see [Microsoft documentation](#).

You can assign rights to manage services with Microsoft [SubInACL](#) utility, a command-line tool enabling administrators to directly edit security information. The tool uses the following syntax:

```
SUBINACL /SERVICE \\MachineName\ServiceName  
/GRANT=[DomainName]UserName[=Access]
```

For example, to grant Start/Stop rights, you might need to execute the command:

```
subinac.exe /service TCBuildAgent /grant=<user login name>=PTO
```

## Linux

- the user must be able to run the `shutdown` command (for the agent machine reboot functionality and the machine shutdown functionality when running in a cloud environment)

## Build-related Permissions

The build process is launched by a TeamCity agent and thus shares the environment and is executed under the OS user used by the TeamCity agent. Please ensure that the TeamCity agent is configured accordingly.

See [Known Issues](#) for related Windows Service Limitations.

## Agent-Server Data Transfers

A TeamCity agent connects to the TeamCity server via the URL configured as the "serverUrl" agent property. This is called [unidirectional](#) agent-to-server connection. If specifically configured, TeamCity agent can use legacy [bidirectional communication](#) which also requires establishing a connection from the server to the agents.

Unless security in transfer between the agent and the server is important, it is recommended to deploy agents and the server into a secure environment and configure agents to use plain HTTP URL for the server as this reduces transfer overhead.

To view whether the agent-server communication is unidirectional or bidirectional for a particular agent, navigate to Agents | <Agent Name> | Agent Summary tab, the Details section, Communication Protocol.

### Unidirectional Agent-to-Server Communication

Agents use unidirectional agent-to-server connection via the polling protocol: the agent establishes an HTTP(S) connection to the TeamCity Server, and polls the server periodically for server commands.

#### HTTPS agent-server connection

It is possible and recommended to set up a server to be available via the HTTPS protocol, so all the data travelling through the connections established from an agent to the server (including the download of build sources, build artifacts, build progress messages and build log) can be secured. See [Using HTTPS to access TeamCity server](#) for configuration details.

### Bidirectional Communication

The bidirectional communication is a legacy connection between the agent and the server and it needs to be specifically enabled (see [below](#)). When enabled, it requires the agent to connect to the server via HTTP (or HTTPS) and the server to connect to the agent via HTTP.

The data that is transferred via the connections established by the server to agents is passed via an unsecured HTTP channel and thus is potentially exposed to any third party that may listen to the traffic between the server and the agents. Moreover, since the agent and server can send "commands" to each other, an attacker that can send HTTP requests and capture responses may in theory trick the agent into executing an arbitrary command and perform other actions with a security impact.

#### Changing Communication Protocol

The communication protocol used by TeamCity agents is determined by the value of the `teamcity.agent.communicationProtocols` server [internal property](#). The property accepts a comma-separated ordered list of protocols ("polling" and "xml-rpc" are supported protocol names) and is set to `teamcity.agent.communicationProtocols=polling` by default. If several protocols are specified, the agent tries to connect using the first protocol from this list and if it fails, it will try to connect via the second protocol in the list. "polling" means unidirectional protocol, "xml-rpc" - older, bidirectional communication.

- To change the communication protocol for all agents, set the TeamCity server `teamcity.agent.communicationProtocols` server [internal property](#). The new setting will be used by all agents which will connect to the server after the change. To change the protocol for the existing connections, restart the TeamCity server.
- By default, the agent's property is not configured; when the agent first connects to the server, it receives it from the TeamCity server. To change the protocol for an individual agent after the initial agent configuration, change the value of the `teamcity.agent.communicationProtocols` property in the [agent's properties](#). The agent's property overrides the server property. After the change the agent will restart automatically upon finishing a running build, if any.

- When connecting TeamCity agents of version 2018.1+ to TeamCity server of version before 9.1 (e.g. after roll back following an upgrade), the agents will need to be updated to use "xml-rpc" protocol (or they can be reinstalled) to be able to connect to an older server to perform the self-update procedure.

## Installing Additional Build Agents

1. Install a build agent using any of the following options:
  - [Using Windows installer](#) (Windows only)
  - [By downloading a zip file and installing manually](#) (any platform)
  - Via the [Docker Agent Image](#) option to prepare a Docker container based on the official [TeamCity Agent image](#)
2. After installation, configure the agent specifying its name and the address of the TeamCity server in the `conf/buildAgent.properties` file.
3. [Start](#) the agent. If the agent does not seem to run correctly, please check the [agent logs](#).

When the newly installed agent connects to the server for the first time, it appears on the [Agents page](#), Unauthorized agents tab visible to administrators/users with the permissions to authorize it. Agents will not run builds until they are authorized in the TeamCity web UI. The agent running on the same computer as the server is authorized by default.

The number of authorized agents is limited by the number of agents licenses on the server. See more under [Licensing Policy](#).

## Installing via Windows installer

1. In the TeamCity Web UI, navigate to the Agents tab.
2. Click the Install Build Agents link and select MS Windows Installer to download the installer.
3. Run the `agentInstaller.exe` Windows Installer and follow the installation instructions.



Please ensure that the user account used to run the agent service has appropriate [permissions](#)

## Installing via Docker Agent Image

1. In the TeamCity Web UI, navigate to the Agents tab.
2. Click the Install Build Agents link and select Docker Agent Image.
3. Follow the instructions on the [page](#) that opens.

## Installing via ZIP File

1. Make sure a JDK (JRE) 8 (Java 6-10 are supported, but 8 is recommended) is properly installed on the agent computer.
2. On the agent computer, make sure the `JRE_HOME` or `JAVA_HOME` environment variables are set (pointing to the installed JRE or JDK directory respectively).
3. In the TeamCity Web UI, navigate to the Agents tab.
4. Click the Install Build Agents link and select Zip file distribution to download the archive.
5. Unzip the downloaded file into the desired directory.
6. Navigate to the `<installation path>\conf` directory, locate the file called `buildAgent.dist.properties` and rename it to `buildAgent.properties`.
7. Edit the `buildAgent.properties` file to specify the TeamCity server URL and the name of the agent. Please refer to [Build Agent Configuration](#) section for details on agent configuration.
8. Under Linux, you may need to give execution permissions to the `bin/agent.sh` shell script.



On Windows you may also want to install the [build agent windows service](#) instead of the manual agent startup.

## Installing via Agent Push

TeamCity provides functionality that allows installing a build agent to a remote host. Currently supported combinations of the server host platform and targets for build agents are:

- from the Unix-based TeamCity server, build agents can be installed to Unix hosts only (via SSH).
- from the Windows-based TeamCity server, build agents can be installed to Unix (via SSH) or Windows (via psexec) hosts.



### SSH note

Make sure the "Password" or "Public key" authentication is enabled on the target host according to preferred authentication method.

## Remote Host Requirements

There are several requirements for the remote host:

Platform	Prerequisites
Linux	<ol style="list-style-type: none"><li>1. Installed JDK(JRE) 6-10 (8 is recommended). The JVM should be reachable with the JAVA_HOME(JRE_HOME) global environment variable or be in the global path (i.e. not in user's .bashrc file, etc.)</li><li>2. The <code>unzip</code> utility.</li><li>3. Either <code>wget</code> or <code>curl</code>.</li></ol>
Windows	<ol style="list-style-type: none"><li>1. Installed JDK/JRE 6-10 (8 is recommended).</li><li>2. <code>Sysinternals psexec.exe</code> on the TeamCity server. It has to be accessible in paths. You can install it using A dministration   Tools page. Note that PsExec applies additional requirements to a remote Windows host (for example, administrative share on remote host must be accessible). Read more about <a href="#">PsExec</a>.</li></ol>

## Installation

1. In the TeamCity Server web UI navigate to the Agents | Agent Push tab, and click Install Agent.... If you are going to use same settings for several target hosts, you can create a preset with these settings, and use it each time when installing an agent to another remote host.
2. In the Install agent dialog, either select a saved preset or choose "Use custom settings", specify the target host platform and configure corresponding settings.  
 Agent push to a Linux system via SSH supports custom ports (the default is 22) specified as the SSH port parameter . The port specified in a preset can be overridden in the host name, e.g. `hostname.domain:2222`, during the actual agent installation.
3. You may need to download `Sysinternals psexec.exe`, in which case you will see the corresponding warning and a link to the Administration | Tools page where you can download it.

You can use Agent Push presets in [Agent Cloud profile](#) settings to automatically install a build agent to a started cloud instance.

## Starting the Build Agent

TeamCity build agents can be started manually or configured to start automatically.

### Manual Start

To start the agent manually, run the following script:

- **for Windows:** <installation path>\bin\agent.bat start
- **for Linux and macOS:** <installation path>\bin\agent.sh start

### Automatic Start

To configure the agent to be started automatically, see the corresponding sections:

#### Windows

Linux : configure daemon process with `agent.sh start` command to start it and `agent.sh stop` command to stop it.  
macOS

### Automatic Agent Start under Windows

To run agent automatically on the machine boot under Windows, you can either set up the agent to be run as a Windows service or use another way of the automatic process start.

Using the Windows service approach is the easiest way, but Windows applies some constraints to the processes run this way. A TeamCity agent works reliably under Windows service provided all the requirements are met, but is often not the case for the build processes configured to be run on the agent.

That is why it is recommended to run a TeamCity agent as a Windows service only if all the build scripts support this. Otherwise, it is advised to use alternative OS-specific ways to start a TeamCity agent automatically.

One of the ways is to configure an [automatic user logon](#) on Windows start and then configure the TeamCity agent start (via `agent.bat start`) on the user logon (e.g. via Windows Task Scheduler).

### Build Agent as a Windows Service

In Windows, you may want to run TeamCity agent as a Windows service to allow it running without any user logged on. If you use the Windows agent installer, you have an option to install the service in the installation wizard.

**!** Service system account

To run builds, the build agent must be started under a user with sufficient permissions for performing a build and managing the service. By default, a Windows service is started under the SYSTEM account which is not recommended for production use due to extended permisisons the account uses. To change it, use the standard Windows Services applet (Control Panel|Administrative Tools|Services) and change the user for the TeamCity Build Agent service.



If you start an Amazon EC2 cloud agent as a Windows service, check [related notes](#).

The following instructions can be used to install the Windows service manually (e.g. after .zip agent installation). This procedure should also be performed to create Windows services for the second and following agents on the same machine.

To install the service:

1. Check if the service with the required name and id (see #4 below, service name is "TeamCity Build Agent" by default) is not present; if installed, remove it.
2. Check that the wrapper.java.command property in the <agent home>\launcher\conf\wrapper.conf file contains a valid path to the Java executable in the JDK installation directory. You can use wrapper.java.command=..../jre/bin/java for the agent installed with the Windows distribution. Make sure to specify the path of the java.exe file without any quotes.
3. If you want to run the agent under user account (recommended) and not "System", add "wrapper.ntservice.account" and "wrapper.ntservice.password" properties to the <agent home>\launcher\conf\wrapper.conf file with appropriate credentials
4. (for second and following installations) modify the <agent>\launcher\conf\wrapper.conf file so that the wrapper.console.title, wrapper.ntservice.name, wrapper.ntservice.displayname and wrapper.ntservice.description properties have unique values within the computer.
5. Run the <agent home>\bin\service.install.bat script under a user with sufficient privileges to register the new agent service. Make sure to start the agent for the first time only after it is configured as described.

To start the service:

- Run <agent home>/bin/service.start.bat  
(or use Windows standard Services applet)

To stop the service:

- Run <agent home>/bin/service.stop.bat  
(or use Windows standard Services applet)

You can also use the standard Windows net.exe utility to manage the service once it is installed.

For example (assuming the default service name):

```
net start TCBuildAgent
```

The <agent home>\launcher\conf\wrapper.conf file can also be used to alter the agent JVM parameters.

The user account used to run the build agent service must have enough rights to start/stop the agent service, as described above.

Automatic Agent Start under Linux

To run agent automatically on the machine boot under Linux, configure daemon process with the agent.sh start command to start it and agent.sh stop command to stop it. Refer to an example procedure below:

1. Navigate to the services start/stop services scripts directory:

```
cd /etc/init.d/
```

2. Open the build agent service script:

```
sudo vim buildAgent
```

3. Paste the following into the file :

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:      TeamCity Build Agent
# Required-Start: $remote_fs $syslog
# Required-Stop:  $remote_fs $syslog
# Default-Start: 2 3 4 5
# Default-Stop:   0 1 6
# Short-Description: Start build agent daemon at boot time
# Description:    Enable service provided by daemon.
### END INIT INFO
#Provide the correct user name:
USER="agentuser"

case "$1" in
start)
su - $USER -c "cd BuildAgent/bin ; ./agent.sh start"
;;
stop)
su - $USER -c "cd BuildAgent/bin ; ./agent.sh stop"
;;
*)
echo "usage start/stop"
exit 1
;;
esac

exit 0
```

4. Set the permissions to execute the file:

```
sudo chmod 755 buildAgent
```

5. Make links to start the agent service on the machine boot and on restarts using the appropriate tool:

- For Debian/Ubuntu:

```
sudo update-rc.d buildAgent defaults
```

- For Red Hat/CentOS:

```
sudo chkconfig buildAgent on
```

#### Automatic Agent Start under macOS

For macOS/Mac OS X, TeamCity provides the ability to load a build agent automatically when a build user logs in. For that, TeamCity uses standard macOS way to start daemon processes - LaunchDaemon plist files.

To configure an automatic build agent startup, follow these steps:

Install and start build agent

- Install a build agent on a Mac via `buildAgent.zip`
- Prepare the `conf/buildAgent.properties` file (set agent name there, at least)
- Make sure that all files under the `buildAgent` directory are owned by `your_build_user` to ensure a proper agent upgrade process.
- Load the build agent via command:

Run these commands under `your_build_user` account

```
mkdir buildAgent/logs # Directory should be created under your_build_user user
sh buildAgent/bin/mac.launchd.sh load
```

You have to wait several minutes for the build agent to auto-upgrade from the TeamCity server. You can watch the process in the logs:

```
tail -f buildAgent/logs/teamcity-agent.log
```

- When the build agent is upgraded and successfully connects to TeamCity server, stop it:

```
sh buildAgent/bin/mac.launchd.sh unload
```

#### Configure automatic build agent start

- After `buildAgent` upgrade from the TeamCity server, copy the `buildAgent/bin/jetbrains.teamcity.BuildAgent.plist` file to `$HOME/Library/LaunchAgents/` directory.

 You can configure to start build agent on system boot, and place `plist` file to `/Library/LaunchDaemons` directory. But in this case there could be some troubles with running GUI tests, and with build agent auto-upgrade. So we do not recommend this approach.

See this [external posting](#) for some more details on LaunchDaemons.

 The old `jetbrains.teamcity.BuildAgent.plist` files bundled with TeamCity before 9.0.4 had a bug which prevent iOS simulator from starting on OS X Yosemite ([TW-38954](#)). The fix is to remove `SessionCreate` property from this file.

- Configure your Mac system to automatically login as a build user, as described [here](#)
- Reboot

On the system startup, the build user should automatically log in, and the build agent should start.

#### Configure a second build agent on macOS

If you want to start several build agents, repeat the procedure of installing and starting build agent with the following changes:

- Install the second build agent in a different directory
- In `conf/buildAgent.properties`, you should specify an unique name for the build agent
- Start and upgrade second agent via `bin/agent.sh` script
- In `bin/jetbrains.teamcity.BuildAgent.plist` file, you should specify unique "`Label`" parameter (which is `jetbrains.teamcity.BuildAgent` by default)
- When copying `jetbrains.teamcity.BuildAgent.plist` file to `LaunchAgents`, you should give it a different name

#### Stopping the Build Agent

To stop the agent manually, run the `<Agent home>\agent` script with a `stop` parameter.

Use `stop` to request stopping after the current build finished.

Use `stop force` to request an immediate stop (if a build is running on the agent, it will be stopped abruptly (canceled))

Under Linux, you have one more option top use: `stop kill` to kill the agent process.

If the agent runs with a console attached, you may also press **Ctrl+C** in the console to stop the agent (if a build is running, it will be canceled).

## Configuring Java

A TeamCity build agent is a Java application and it requires JDK version 6 or later to work. Oracle Java SE JDK 8, 32-bit is recommended. [Java download page](#)

A build agent contains two processes:

- Agent Launcher — a Java process that launches the agent process
- Agent — the main process for a Build Agent; runs as a child process for the agent launcher

The (Windows) .exe TeamCity distribution comes bundled with Java 8.

If you run a previous version of the TeamCity agent, you will need to repeat the agent installation to update the JVM.

Using x32 bit JDK (not JRE) is recommended. JDK is required for some build runners like [IntelliJ IDEA Project](#), [Java Inspections](#) and [Duplicates](#). If you do not have Java builds, you can install JRE instead of JDK.

Using of x64 bit Java is possible, but you might need to double the `-Xmx` memory value for the main agent process (see [Configuring Build Agent Startup Properties](#) and alike [section](#) for the server).

For the .zip agent installation you need to install the appropriate Java version. Make it available via PATH or available in one of the following places:

- the `<Agent home>/jre` directory
- in the directory pointed to by the `TEAMCITY_JRE`, `JAVA_HOME` or `JRE_HOME` environment variables (check that you only have one of the variables defined).
- if you plan to run the agent via Windows service, make sure to set `wrapper.java.command` property in the `<agent home>\launcher\conf\wrapper.conf` file to a valid path to the Java executable

## Upgrading Java on Agents

If a build agent uses a Java version older than the one required by agent (Java 6 currently), the agent will not be able to start and will be shown as disconnected.

If a build agent uses a Java version older than the recommended Java 8 (e.g. Java 6 or 7), you will see the corresponding warning on the agent's page and a [health item](#) in the web UI.



Support for Java prior to version 8 will be dropped in future TeamCity versions. Consider upgrading Java on the agent if you see the warning.

To update Java on agents, do one of the following:

- Upgrade the Java automatically: if the appropriate Java version of the same bitness as the current one is detected on the agent, the agent page provides an action to upgrade the Java automatically. Upon the action invocation, the agent process is restarted (once the agent becomes idle, i.e. finishes the current build if there is one) using the new java.
- (Windows) Since the build agent .exe installation comes bundled with the required Java, you can just reinstall the agent using the .exe installer obtained from the TeamCity server | Agents page.
- Install a required Java on the agent into one of the standard locations, and restart the agent - the agent should then detect it and provide an action to use a newer Java in the web UI (see above).
- Install a required Java on the agent and [configure the agent](#) to use it.



In a rare case of updating the Java for the process that launches the TeamCity agent, use one of the options for the agent Java upgrade.

Another way for Build Agent started as a Windows service, is to stop the service, change the `wrapper.java.command` variable in `buildAgent\launcher\conf\wrapper.conf` to point to the new `java.exe` binary, and restart the service.

## Installing Several Build Agents on the Same Machine

You can install several TeamCity agents on the same machine if the machine is capable of running several builds at the same

time. However, we recommend running a single agent per (virtual) machine to minimize builds cross-influence and making builds more predictable. When installing several agents, it is recommended to install them under different OS users so that user-level resources (like Maven/Gradle/NuGet local artifact caches) do not conflict.

TeamCity treats all agents equally regardless of whether they are installed on the same or on different machines. When installing several TeamCity build agents on the same machine, please consider the following:

- The builds running on such agents should not conflict by any resource (common disk directories, OS processes, OS temp directories).
- Depending on the hardware and the builds, you may experience degraded builds' performance. Ensure there are no disk, memory, or CPU bottlenecks when several builds are run at the same time.
- It is recommended to set up the agents to be run under different OS users

After having one agent installed, you can install additional agents by following the regular installation procedure (see an exception for the Windows service below), but make sure that:

- The agents are installed in separate directories.
- The agents have the distinctive `workDir` and `tempDir` directories in the `buildAgent.properties` file.
- Values for the `name` and `ownPort` properties of `buildAgent.properties` are unique.
- No builds running on the agents have the absolute checkout directory specified.

Make sure there are no build configurations with the absolute `checkout directory` specified (alternatively, make sure such build configurations have the "clean checkout" option enabled and they cannot be run in parallel).

Usually, for a new agent installation you can just copy the directory of the existing agent to a new place with the exception of its "temp", "work", "logs" and "system" directories. Then, modify `conf/buildAgent.properties` with the new `name`, `ownPort` values. Please also clear (delete or remove the value) for the `authorizationToken` property and make sure the `workDir` and `tempDir` are relative/do not clash with another agent.

If you use Windows installer to install additional agents and want to run the agent as a service, you will need to perform manual steps as installing second agent as a service on the same machine is not supported by the installer: the existing service is overwritten (see also a [feature request](#)).

In order to install the second agent, it is recommended to install the second agent [manually](#) (using .zip agent distribution). You can use Windows agent installer and do not opt for service installation, but you will lose uninstall option for the initially installed agent this way.

After the second agent is installed, register a new service for it as mentioned in the [section above](#).

 For step-by-step instructions on installing a second Windows agent as a service, see a related [external blog post](#).

See also:

[Concepts: Build Agent](#)

## Build Agent Configuration

Configuration settings of the build agent are stored in the `<TeamCity Agent Home>/conf/buildagent.properties` file. The file can also store properties that will be published on the server as Agent properties and can participate in the [Agent Requirements](#) expressions.

All the `system` and `environment properties` defined in the file will be passed to every build run on the agent.

The file is a [Java properties](#) file.

A quick guide is:

- use `property_name=value<newline>` syntax
- use "#" in the first position of the line for a comment
- use "/" instead of "\" as the path separator. If you need to include "\" escape it with another "\".
- whitespaces within a line matter

This is an example of the file:

```

## The address of the TeamCity server. The same as is used to open the TeamCity web interface in
## the browser.
serverUrl=http://localhost:8111/

## The unique name of the agent used to identify this agent on the TeamCity server
## Use blank name to let server generate it.
## By default, this name would be created from the build agent's host name
name=Default agent

## Container directory to create default checkout directories for the build configurations.
workDir=../work

## Container directory for the temporary directories.
## Please note that the directory may be cleaned between the builds.
tempDir=../temp

## Container directory for agent state files and caches.
## TeamCity agent assumes ownership of the directory and can delete the content inside.
systemDir=../system

#####
# Optional Agent Properties #
#####
## A token which is used to identify this agent on the TeamCity server for agent authorization
purposes.
## It is automatically generated and saved back on the first agent connection to the server.
authorizationToken=1234567890abcdefghijklml

```



Please make sure that the file is writable for the build agent process itself. For example, the file is updated to store its authorization token that is generated on the server-side.

If the "name" property is not specified, the server will generate a build agent name automatically. By default, this name will be created from the build agent's host name.

The file can be edited while the agent is running: the agent detects the change and (upon finishing a running build, if any) restarts automatically loading the new settings.

### Optional Properties

If the default polling protocol is changed in favor of legacy [bidirectional communication](#) between the server and the agent, the server must be able to open HTTP connections to the agent.

The port where the TeamCity build agent starts and where it listens for the incoming data from the server is determined via the `ownPort` property (9090 by default). If the firewall is configured, make sure that the incoming connections for this port are allowed on the agent computer.

```
ownPort=9090
```



If more than one build agent is hosted on the same machine, different ports must be assigned to them via the `ownPort` property in `buildAgent.properties` file of every agent.

The IP address used by TeamCity server to connect to the build agent is automatically detected by the server when the agent first connects to TeamCity, unless the `ownAddress` property is defined. If the machine has several network interfaces, automatic detection may fail and it is recommended to specify the `ownAddress` property:

```
ownAddress=<own IP address or server-accessible domain name>
```

## Set up Agent behind Proxy

Since TeamCity 2017.1 it is possible to configure a forward proxy server for agent-to-server connections.

On the TeamCity agent side, specify the proxy to connect to TeamCity server using the following properties in the `buildAgent.properties` file:

```
## The domain name or the IP address of the proxy host and the port
teamcity.http.proxyHost=123.45.678.9
teamcity.http.proxyPort=8080

## If the proxy requires authentication, specify the login and password
teamcity.http.proxyLogin=login
teamcity.http.proxyPassword=password
```

Note that the proxy has to be configured not to cache any TeamCity server responses; e.g. if you use Squid, add "cache deny all" line to the `squid.conf` file.

See also:

[Concepts: Build Agent](#)  
[Administrator's Guide: Predefined Build Parameters](#) | [Configuring Agent Requirements](#) | [Configuring Build Parameters](#)

## TeamCity Integration with Cloud Solutions

TeamCity integration with cloud (IAAS) solutions allows TeamCity to provision virtual machines running TeamCity agents on-demand based on the build queue state.

This page covers general information about the configuration of integration. For the list of currently supported solutions, refer to [Available Integrations](#).

On this page:

- [General Description](#)
- [Available Integrations](#)
- [TeamCity Setup for Cloud Integration](#)
  - [Preparing a virtual machine with an installed TeamCity agent](#)
  - [Preparing a virtual machine](#)
  - [Capturing an image from a virtual machine](#)
  - [Configuring a cloud profile in TeamCity](#)
- [Estimating Costs](#)
  - [Traffic Estimate](#)
  - [Running Costs](#)

### General Description

In a large TeamCity setup with many projects, it can be difficult to predict the load on build agents, and the number of agents we need to be running. With the cloud agent integration configured, TeamCity will leverage clouds elasticity to provision additional build agents on-demand.

For each queued build TeamCity first tries to start it on one of the regular, non-cloud agents. If there are no usual agents available, TeamCity finds a matching cloud image with a compatible agent and starts a new instance for the image. TeamCity ensures that number of running cloud instances limit is not exceeded.

The integration requires:

- a configured virtual machine with an installed TeamCity agent in your cloud pre-configured to start the TeamCity agent

- on boot,
- a configured cloud [profile in TeamCity](#).

Once a cloud profile is configured in TeamCity with one or several images, TeamCity does a test start of one instance for all the newly added images to learn about the agents configured on them. When the agents are connected, TeamCity stores their parameters to be able to correctly process build configurations-to-agents compatibility. An agent connected from a cloud instance started by TeamCity is automatically authorized, provided there are available agent licenses: the number of cloud agents is limited by the total number of agent licenses you have in TeamCity. After that the agent is processed as a regular agent.

Depending on the profile settings, when TeamCity realizes it needs more agents, it can either

- start an existing virtual machine and stop it (after the build is finished or an idle timeout elapses). The machines that are stopped will be deallocated so the virtual machine fee does not apply when the agent is not active. The storage cost for this type of TeamCity agent will still apply.
- create a new virtual machine from an image. Such machines will be destroyed (after the build is finished or an idle timeout elapses). This ensures that the machines will incur no further running costs.

The disconnected agent will be removed from the authorized agents list and deleted from the system to free up TeamCity build agent licenses.

## Available Integrations

Integration with cloud solutions is implemented as plugins. The platform-specific details are covered on the following pages:

- [Amazon EC2](#)
- [VMWare vSphere](#)

Also available as separate plugins are [Windows Azure](#), [Google Cloud Agents](#) and [others](#). New integrations can be implemented using custom TeamCity plugins, see [Implementing Cloud support](#).

## TeamCity Setup for Cloud Integration

This section describes general steps required for cloud integration.

### Preparing a virtual machine with an installed TeamCity agent

The requirements for a virtual machine/image to be used for TeamCity cloud integration:

- The TeamCity agent must be correctly [installed](#) and configured to start [automatically](#) on the machine startup.
- the `buildAgent.properties` file can be left "as is". The `serverUrl`, `name`, and `authorizationToken` properties can be left empty or set to any value, they are ignored when TeamCity starts the instance unless otherwise specifically stated in the [platform-specific documentation](#).

Provided these requirements are met, the usual TeamCity agent installation and cloud-provider image bundling procedures are applicable.

If you need the [connection](#) between the server and the agent machine to be secure, you will need to set up the agent machine to establish a secure tunnel (e.g. VPN) to the server on boot so that the TeamCity agent receives data via the secure channel. Please keep in mind that communication between TeamCity agent and server is bi-directional and requires an open port on the agent as well as on the server.

### Preparing a virtual machine

- Create and start a virtual machine with desired OS installed.
- Connect and log in to the virtual machine.
- Configure the running instance:
  - [Install](#) and configure build agent.
    - Configure the server name and agent name in the `buildAgent.properties` file — this is optional if TeamCity will be configured to launch the image, but it is useful to test the agent is configured correctly.
    - It usually makes sense to specify `tempDir` and `workDir` in `conf/buildAgent.properties` to use a non-system drive (e.g. `D` drive under Windows)
  - Install any additional software necessary for the builds on the machine (e.g. Java, the .Net framework)
  - Start the agent and wait until it connects to server, ensure it is working OK and is compatible with all necessary build configurations (in the TeamCity Web UI, go to the Agents page, select the build agent and view the Compatible Configurations tab), etc.
  - Configure the system so that the agent is [started on the machine boot](#) (and make sure TeamCity server is accessible on the machine boot).
  - Check the port on which the build agent will listen for incoming data from TeamCity and open the required firewall ports (usually 9090).
- Test the setup by rebooting machine and checking that the agent connects normally to the server. Once the agent connects, it will automatically update all the plugins. Please wait until the agent is connected completely so that all

plugins are downloaded to the agent machine.

If you want TeamCity to start an existing virtual machine and stop it after the build is finished or an idle timeout elapses, the setup above is all you need. If you want TeamCity to create and start virtual machines from an image and terminate the machine after use, the image should be captured from the virtual machine that was created.

## Capturing an image from a virtual machine

Do the following:

1. Complete the steps for [creating a virtual machine](#).
  - Remove any temporary/history information in the system.
  - Stop the agent (under Windows, stop the service but leave it in the Automatic startup type)
  - (optional) Delete the content of the `logs` and `temp` directories in the [agent home](#)
  - (optional) Clean up the `<Agent Home>/conf/` directory from platform-specific files
  - (optional) Change the `buildAgent.properties` file to remove the `name`, `serverUrl`, and `authorizationToken` properties unless otherwise specifically stated in the [platform-specific documentation](#).
2. Make a new image from the running instance. Refer the cloud documentation on how to do this.



TeamCity agent auto-upgrades whenever distribution of agent (e.g. after TeamCity upgrade) or agent plugins on the server changes. If you want to reduce the agent startup time, you might want to capture a new virtual machine image after the agent distribution or plugins have been updated.

## Configuring a cloud profile in TeamCity

A cloud profile is a collection of settings for TeamCity to start virtual machines with installed TeamCity agents on-demand. Prior to TeamCity 2017.1, profiles are configured on the TeamCity | Administration | Agent Cloud page. In the later versions, cloud profiles are configured at the project level, on the dedicated page of project settings.

## Estimating Costs

The cloud provider pricing applies. Please note that the charges can depend on the specific configuration implemented to deploy TeamCity. We advise you to check your configuration and the cloud account data regularly in order to discover and prevent unexpected charges as early as possible.

Please note that traffic volumes and necessary server and agent machines characteristics depend a big deal on the TeamCity setup and nature of the builds run.

## Traffic Estimate

Here are some points to help you estimate TeamCity-related traffic:

If the TeamCity server is not located within the same region or affinity group as the agent, the traffic between the server and agent is subject to usual external traffic charges imposed by your provider.

When estimating traffic, please remember that there are many types of traffic related to TeamCity (see the non-complete list below).

External connections originated by server:

- VCS servers
- Email and Jabber servers
- Maven repositories
- NuGet repositories

Internal connections originated by server:

- TeamCity agents (checking status, sending commands, retrieving information like thread dumps, etc.)

External connections originated by agent:

- VCS servers (in case of agent-side checkout)
- Maven repositories
- NuGet repositories
- any connections performed from the build process itself

Internal connections originated by agent:

- TeamCity server (retrieving build sources in case of server-side checkout or personal builds, downloading artifacts, etc.)

Usual connections used by the server

- Web browsers
- IDE plugins

## Running Costs

Cloud providers calculate costs based on the virtual machine uptime, so it is recommended to adjust the timeout setting according to your usual builds length. This reduces the amount of time a virtual machine is running.

It is also highly recommended to set an execution timeout for all your builds so that a build hanging does not cause prolonged instance running with no payload.

## Setting Up TeamCity for Amazon EC2

TeamCity Amazon EC2 integration allows you to configure TeamCity with your Amazon account and then start and stop images with TeamCity agents on-demand based on the queued builds.

For integrations with other cloud solutions, see the following pages:

- VMWare vSphere (bundled)
- Windows Azure
- Google Cloud Agents
- [Implementing Cloud support](#) enables to you create your own integration.

On this page:

- [General Description](#)
- [Configuration](#)
  - [Required IAM permissions](#)
    - [Optional permissions](#)
  - [Preparing Image with Installed TeamCity Agent](#)
  - [Agent auto-upgrade Note](#)
  - [Configuring a cloud profile in TeamCity](#)
    - [Amazon EC2 Spot Instances support](#)
    - [Amazon EBS-Optimized Instances](#)
    - [Tagging for TeamCity-launched instances](#)
      - [Requirements](#)
      - [Automatic tags](#)
      - [Custom tags](#)
    - [Tagging instance-dependent resources](#)
    - [Sharing single EBS instance between several TeamCity servers](#)
  - [New instance types](#)
  - [Proxy settings](#)
  - [Custom script](#)
  - [Estimating EC2 Costs](#)
  - [Traffic Estimate](#)
    - [Uptime Costs](#)

### General Description

It is assumed that the machine images are pre-configured to start TeamCity agent on boot (see details [below](#)). The exception is usage of [agent push](#).

Once a cloud profile is configured in TeamCity with one or several images, TeamCity does a test start for all the new images to learn about the agents configured on them.

Once the agents are connected, TeamCity stores their parameters to be able to correctly process build configurations-to-agents compatibility.

For each queued build, TeamCity first tries to start it on one of the regular, non-cloud agents. If there are no usual agents available, TeamCity finds a matching cloud image with a compatible agent and starts a new instance for the image. TeamCity ensures that the running instances limit configured in the cloud profile is not exceeded.

Once an agent is connected from a cloud instance started by TeamCity, it is automatically authorized (provided there are available agent licenses). After that the agent is processed as a regular agent.

If running timeout is configured on the cloud profile and it is reached, the instance is terminated.

If an EBS-based instance id is specified in the images list, the instance is stopped instead.

On instance terminating/stopping, its disconnected agent is removed from authorized agents list and is deleted from the system.

[Amazon EC2 Spot Instances](#) are supported.

### Configuration

Understanding Amazon EC2 and ability to perform EC2 tasks is a prerequisite for configuring and using TeamCity Amazon EC2

integration.

Basic TeamCity EC2 setup involves:

- preparing an Amazon EC2 image (AMI) with an installed TeamCity agent
- configuring EC2 integration on TeamCity server



Please note that the number of EC2 agents is limited by the total number of agent licenses you have in TeamCity.

Please ensure that the server URL specified on Global Settings page in Administration area is correct since agents will use it to connect to the server.

If you need TeamCity to use proxy to access EC2 services, please read on a current workaround in the dedicated issue.

Required IAM permissions

TeamCity requires the following permissions for Amazon EC2 Resources:

- ec2:Describe\*
- ec2:StartInstances
- ec2:StopInstances
- ec2:TerminateInstances
- ec2:RebootInstances
- ec2:RunInstances
- ec2:ModifyInstanceAttribute
- ec2:Tags

To use **spot instances**, the following additional permissions are required:

- ec2:RequestSpotInstances
- ec2:CancelSpotInstanceRequests

To launch an **instance with Iam Role** (applicable to instances cloned from AMI-s only) the following additional permissions are required:

- iam>ListInstanceProfiles
- iam>PassRole

An example of custom IAM policy definition (allows all ec2 operations from a specified IP address):

AWS IAM Policy Definition Example » Expand source

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "1",  
            "Effect": "Allow",  
            "Action": "ec2:*",  
            "Condition": {  
                "IpAddress": {  
                    "aws:SourceIp": "<TeamCity server IP address>"  
                }  
            },  
            "Resource": "*"  
        }  
    ]  
}
```

Optional permissions

See the [section below](#) for permissions to set IAM roles on an agent instance.

View information on example policies for [Linux](#) and [Windows](#) on the Amazon website.

#### Preparing Image with Installed TeamCity Agent

Here are the requirements for an image that can be used for TeamCity cloud integration:

- TeamCity agent should be correctly [installed](#).
- TeamCity agent should start on machine startup
- `buildAgent.properties` can be left "as is". `"serverUrl"`, `"name"` and `"authorizationToken"` properties can be empty or set to any value, they are ignored when TeamCity starts the instance.

Provided these requirements are met, usual TeamCity agent installation and cloud-provider image bundling procedures are applicable.

If you need the [connection](#) between the server and the agent machine to be secure, you will need to set up the agent machine to establish a secure tunnel (e.g. VPN) to the server on boot so that TeamCity agent receives data via the secure channel.

Recommended image (e.g. Amazon AMI) preparation steps:

1. Choose one of existing generic images.
2. Start the image.
3. Configure the running instance:
  - Install and configure build agent:
    - Configure server name and agent name in `conf/buildAgent.properties` — this is optional, if the image will be started by TeamCity, but it is useful to test the agent is configured correctly.
    - It usually makes sense to specify `tempDir` and `workDir` in `conf/buildAgent.properties` to use non-system drive (d: under Windows)
  - Install any additional software necessary for the builds on the machine.
  - Run the agent and check it is working OK and is compatible with all necessary build configurations, etc.
  - Configure system so that agent it is started on machine boot (and make sure TeamCity server is accessible on machine boot).
  - To ensure proper TeamCity agent communication with EC2 API under Windows, add a dependency from the TeamCity Build Agent service on the [AmazonSSMAgent](#) or [EC2Launch / EC2Config](#) service (the service which makes sure the machine is fully initialized in regard to AWS infrastructure use). This can be done, for example, via the [Registry](#) or using `sc config`, e.g. `sc config TCBuildAgent depend=EC2Config`. Alternatively, you can use the "Automatic (delayed start)" service starting mode.
4. Test the setup by rebooting machine and checking that the agent connects normally to the server.
5. Prepare the Image for bundling:
  - Remove any temporary/history information in the system.
  - Stop the agent (under Windows stop the service but leave it in Automatic startup type)
  - Delete content logs and temp directories in agent home (optional)
  - Delete "`<Agent Home>/conf/amazon-*`" file (optional)
  - Change `config/buildAgent.properties` to remove properties: `name`, `serverUrl`, `authorizationToken` (optional). `serverUrl` can be removed for EC2 integration plugins. Other plugins might require that it is present and set to correct value.
6. Make a new image from the running instance (or just stop it for Amazon EBS images).

#### Agent auto-upgrade Note

TeamCity agent auto-upgrades whenever distribution of agent plugins on the server changes (e.g. after TeamCity upgrade). If you want to cut agent startup time, you might want to re-bundle the agent AMI after agent plugins have been auto-updated.

#### Configuring a cloud profile in TeamCity

Next configure Amazon EC2 [Agent Cloud Profile](#) in the Server Administration UI, on the Administration | Agent Cloud.

#### IAM profiles

It is possible to use IAM profiles with build agents launched as Amazon EC2 instances, which requires the supplied AWS account to have the following permissions:

- `iam>ListInstanceProfiles`
- `iam:PassRole`

IAM profiles must be [preconfigured](#) in Amazon EC2. In the TeamCity Web UI, the IAM profile dropdown enables you to select a role. Every new launched EC2 instance will assume the [selected IAM role](#).

#### Amazon EC2 Spot Instances support

TeamCity supports [Amazon EC2 Spot Instances](#) and now you can place your bid on unused EC2 capacity and use it, as long as your suggested price exceeds the current "Spot price".

Enable spot instances on the TeamCity [cloud profile page](#) and enter your current bid level. After the profile is saved, TeamCity creates a spot instance request (sir) and the availability of Amazon spot instances is checked. If the sir is not fulfilled after 10 min, it is killed by Amazon. If there are still queued builds which can run on such an agent, the request is automatically recreated by TeamCity. NOTE: It is not recommended to use spot instances for production-critical builds due to the possibility of [an unexpected spot instance termination by Amazon](#).

## Amazon EBS-Optimized Instances

The behavior of [EBS-optimization](#) in TeamCity 10.0 is similar to that offered by EC2 console. When configuring an image of the [Amazon cloud profile](#), the optimization can be set using the corresponding box of the Instance Type. Note that

- EBS-optimization is turned on by default for `c4.*`, `d2.*`, and `m4.*` (non-configurable)
- EBS-optimization is turned off by default for any other instance types and can be turned on for instances that support it (such as `c3.xlarge`, etc.)

## Tagging for TeamCity-launched instances

### Requirements

The following requirements must be met for tagging instances launched by TeamCity:

- you have the `ec2:Tags` permissions
- the [maximum number of tags \(50\)](#) for your Amazon EC2 resource is not reached.

In the absence of tagging permissions, TeamCity will still launch Amazon AMI and EBS images with no tags applied; Amazon EC2 Spot Instances will not be launched.

### Automatic tags

TeamCity enables users to get instance launch information by marking the created instances with `"teamcity:TeamcityData"` tag containing `<server UUID>:-<cloud profile ID>:-<image reference>`.

### Custom tags

Custom tags can be applied to EC2 cloud agent instances: when configuring Cloud profile settings, in the Add Image/ Edit Image dialog use the Instance tags: field to specify tags in the format of `<key1>=<value1>,<key2>=<value2>`. [Amazon tag restrictions](#) need to be considered.

If you'd like to use `equal(=)` sign in the tag value, no escaping is needed. For instance, the string `extraParam=name=John` will be parsed into `<key=extraParam>` and value `<name=John>`.

## Tagging instance-dependent resources

When launching Amazon EC2 instances, TeamCity tags all the resources (e.g. volumes and network adapters) associated with the created instances, which is important when evaluating the overall cost of an instance (taking into account the storage drive type and size, I/O operations (for standard drives), network (transfers out), etc.

## Sharing single EBS instance between several TeamCity servers

As mentioned [above](#), TeamCity tags every instance it launches with `"teamcity:TeamcityData"` tag that represents server, cloud profile and source (AMI or EBS-instance). So, in case when several TeamCity servers tries to use the same EBS instance, the second one will see message "Instance is used by another TeamCity server. Unable to start/stop it". If you are sure that no other TeamCity servers are working with this instance, you can delete the `"teamcity:TeamcityData"` tag and the instance will become available for all TeamCity servers again.

## New instance types

Since Amazon doesn't provide a robust API method to retrieve all instance types, Amazon integration relies on periodical update of AWS SDK to make new instance types available.

However, there is a workaround if you are not willing to wait. To register new Instance Types, use the following [internal property](#):

`teamcity.ec2.instance.types` property with new instance types separated by `,`

## Proxy settings

If your TeamCity server needs to use a proxy to connect to AWS API endpoint, configure the following server [internal properties](#) to connect to Amazon AWS addresses.

teamcity.http.proxy.host.ec2 - proxy server host name  
teamcity.http.proxy.port.ec2 - proxy server port

For proxy server authentication:

teamcity.http.proxy.user.ec2 - proxy access user name  
teamcity.http.proxy.password.ec2 - proxy access user password

For NTLM authentication:

teamcity.http.proxy.domain.ec2 - proxy user domain for NTLM authentication  
teamcity.http.proxy.workstation.ec2 - proxy access workstation for NTLM authentication

#### Custom script

It is possible to run a custom script on the instance start (applicable to instances cloned from AMI's only). The Amazon website details the script format for [Linux](#) and [Windows](#).

#### Estimating EC2 Costs

Usual Amazon EC2 pricing applies. Please note that Amazon charges can depend on the specific configuration implemented to deploy TeamCity. We advise you to check your configuration and Amazon account data regularly in order to discover and prevent unexpected charges as early as possible.

Please note that traffic volumes and necessary server and agent machines characteristics depend a big deal on the TeamCity setup and nature of the builds run. See also [How To...#Estimate Hardware Requirements for TeamCity](#).

#### Traffic Estimate

Here are some points to help you estimate TeamCity-related traffic:

- If TeamCity server is not located within the same EC2 region or availability zone that is configured in TeamCity EC2 settings for agents, traffic between the server and agent is subject to usual Amazon EC2 external traffic charges.
- When estimating traffic please bear in mind that there are lots types of traffic related to TeamCity (see a non-complete list below).

External connections originated by server:

- VCS servers
- Email and Jabber servers
- Maven repositories

Internal connections originated by server:

- TeamCity agents (checking status, sending commands, retrieving information like thread dumps, etc.)

External connections originated by agent:

- VCS servers (in case of agent-side checkout)
- Maven repositories
- any connections performed from the build process itself

Internal connections originated by agent:

- TeamCity server (retrieving build sources in case of server-side checkout or personal builds, downloading artifacts, etc.)

Usual connections served by the server:

- web browsers
- IDE plugins

#### Uptime Costs

As Amazon rounds machine uptime to the nearest full hour, please adjust timeout setting on the EC2 image setting on TeamCity cloud integration settings according to your usual builds length.

It is also highly recommended to set execution timeout for all your builds so that a build hanging does not cause prolonged instance running with no payload.

#### Setting Up TeamCity for VMware vSphere and vCenter

TeamCity vSphere integration allows using TeamCity [agent cloud features](#) with VMware vSphere and vCenter installation. It requires configuring TeamCity with your VMware vSphere/vCenter account and then handles automatic creation/starting/stopping/deleting of the virtual machines with TeamCity agents on-demand, based on the queued builds.

The functionality is implemented as a plugin bundled with TeamCity.

On this page:

- Requirements
- Features
- Usage
  - Notes on configuring Agent cloud profile

## Requirements

 The integration requires a write access for vSphere/vCenter via API, which is available in the licensed versions; the API support of free versions is limited to read-only access.

## Features

The TeamCity vSphere integration allows you to:

- select the type of behavior:
  - either to start/stop an existing virtual machine
  - or clone a virtual machine or a template, deleting the clone once it becomes idle (when the build is finished or an idle timeout elapses, depending on the profile settings)
- select a virtual machine snapshot to start
- specify the folder for clones and the resource pool where your machine will be allocated
- configure the maximum number of started instances.

## Usage

The following steps are required to set up TeamCity-VMware vSphere agent cloud integration:

1. Create a virtual machine where your builds will run. Refer to the VMware vSphere web site for details on creating [virtual machines](#).

Prepare a [machine with an installed TeamCity agent](#).

 When configuring the TeamCity build agent, be sure to specify the valid TeamCity server URL in the build agent properties.

1. If you want TeamCity to start/stop this machine on demand or to clone it, proceed to configuring the VMware [cloud profile](#) on the TeamCity server. When the profile is modified, TeamCity detects the changes immediately, and forces shutdown of the agents started prior to these changes once the agents finish the current build.
2. If you want to create a template of this machine and clone it, refer to the VMware vSphere web site for details on creating [templates](#) and proceed to configuring the VMware [cloud profile](#) on the TeamCity server. Make sure to specify the valid [vCenter SDK URL](#).

## Notes on configuring Agent cloud profile

- You can limit a number of instances across all images and /or set the limit per image.
- It is possible to specify unique hostnames for cloud agents: when adding an image, choose a customization spec from the corresponding field. The option is available for Windows and Linux virtual machines.
- When using the same image in different cloud profiles, to avoid possible conflicts, it is possible to use a custom agent image name when configuring a cloud profile in TeamCity. This can be also useful with naming patterns for agents. When a custom agent image name is specified, the names of cloud agent instances cloned from the image will be based on this name.

## Disabling TeamCity Plugins

TeamCity plugins are parts of TeamCity used to provide integration with specific build tools / perform specific tasks.

 If you disable some plugins, TeamCity will not be able to run their builds/ tasks, so be sure you do not need the features you are going to disable.

To disable plugins via config files:



This is applicable to versions prior to TeamCity 2017.2. Starting from this version, it is recommended to disable plugins via the web UI.

1. Create the `disabled_plugins.txt` file in `<TeamCity data directory>/config`.
2. Provide a new-line separated list of the plugin names (as specified in the `teamcity-plugin.xml` file of every plugin, the file is usually located in the default directory for bundled TeamCity plugins `<TeamCity web application>/WEB-INF/plugins/<plugin>/teamCity-plugin.xml`).
3. Restart the server. The plugin names are displayed as the strikethrough text on the Administration | Plugins List page noting the file where the plugins were disabled.

To disable plugins via Web UI:

Any plugin can be disabled using the TeamCity UI: on the Administration | Plugins List page every external plugin has a button, clicking which opens a popup with the corresponding option; the bundled plugins have a link which can be used to disable them. On disabling a plugin, a warning is displayed that this may impact other TeamCity components (e.g. other plugins). During the next server restart, the disabled plugin is not loaded. If there are other plugins depending on the disabled one, they will not be loaded either. Disabled plugins are greyed out in the list.

## Installing Additional Plugins

To install a plugin manually:

1. Copy the zip plugin package into the `<TeamCity Data Directory>/plugins` directory. If you have an earlier version of the plugin in the directory, remove it. Alternatively, use the Administration | Plugins List page to upload the zip plugin package into the directory (do not change the file name of the plugin).
2. Restart the TeamCity server. Check the installed plugin version is listed on the Administration | Plugins List page.

If the plugin has an agent part, all the agents will be updated automatically. For plugins with the agent part only, the server restart can be skipped: the agents will be updated automatically.

To install a plugin via Web UI:

1. Go to the Administration | Plugins List page and upload a plugin zip from your local machine using the corresponding link.
2. Once the plugin is added, the option to restart the server appears on the page. Click the link to restart the server and check that the installed plugin version is listed on the Administration | Plugins List page.

The other option on this page, the Available plugins link, opens the [list of TeamCity plugins](#) on the official JetBrains site.

You can uninstall a plugin from the Web UI:

1. Go to the Administration | Plugins List page, locate an external plugin in the list, click the arrow icon next to it and use the Delete option.
2. Once the plugin is deleted, the option to restart the server appears on the page. Click the link to restart the server and check that the plugin version is no longer listed on the Administration | Plugins List page.



To uninstall a plugin manually, remove the plugin package from the `<TeamCity Data Directory>/plugins` directory and restart the TeamCity server.

## Installing Agent Tools

In TeamCity an agent tool (i.e. a set of files/a binary distribution) is a type of plugin without any classes loaded into the runtime; agent tools are used to only distribute binary files to agents.

TeamCity allows you to install / remove additional tools on all the agents, which is especially useful in the environments with a large number of build agents as you can distribute tools to or remove them from all build agents at once, centralize configuration files distribution (e.g. you want to distribute a custom configuration file/library to all agents), etc.

The Administration | Tools page provides a unified interface to set up tools to be used by appropriate plugins. You can install different versions of any of the tools and/or change the default one. The tools will be automatically distributed to all build agents to be used in the related runners.

The following types of tools can be managed up via the Administration | Tools page:

- IntelliJ Inspections and Duplicates Engine with the bundled version of IntelliJ IDEA set as default.
- JetBrains dotCover Command Line Tools with the bundled version set as default. Used to collect code coverage for your .Net project
- JetBrains ReSharper Command Line Tools: by default the tools are bundled with TeamCity and are used by Inspections (.NET), Duplicates Finder (.NET) build runners to run code analysis.
- Maven: several bundled versions are displayed, with 3.0.5 set as default.
- NuGet.exe used in NuGet specific build steps and NuGet Dependency Trigger. NuGet packages (.nupkg files) with the tools/NuGet.exe file inside are supported.
- NUnit 3: different versions can be installed and the default version set/changed
- Sysinternals handle.exe used to determine processes which hold files in the checkout directory on Windows agents.
- Sysinternals psexec.exe required for installing a TeamCity agent from a Windows server to a Windows host using Agent push
- You can also upload your own tool as a .zip archive: the structure of the tool plugin is described [on the Plugins Packaging page](#). TeamCity will use the name of the zip file as the tool name on all agents. The zip file will be automatically unpacked on the agents to the directory with the same name.  
When a tool is installed, a separate section appears on the page displaying the installed version(s), the tool usages, the default version or the ability to change the default. You can also remove an installed tool/version.

You can see that the tool appears on the agent in the TeamCity Web UI by checking [configuration parameters reported by the agent](#) in the form `teamcity.tool.<the installed tool id>`. You can use this parameter in your build: reference this parameter in the TeamCity Web UI (anywhere where the `%parameter%` format is supported) or refer to [this parameters in your build as an environment or a system parameter](#).

To distribute a tool to all agents, TeamCity places them into the `<TeamCity Data Directory>/plugins/.tools` and monitors the content of this folder. Agents will restart in the process of obtaining the tool.

## Upgrade Notes

### Changes from 2018.1 to 2018.1.1

No noteworthy changes.

### Changes from 2017.2.x to 2018.1

#### Known issues

While publishing NuGet packages into the TeamCity NuGet feed in multiple build steps, only the packages published by the first build step will be visible. See [TW-55703](#) for details. If you experience problems with download of NuGet packages published within archives see [TW-55833](#).

#### Stricter rules for parameter names used in parameter references

Names of the Build Configuration parameters are now validated in more strict manner. While already existing parameters should continue to work, it is highly recommended to review the names and use Latin letters and no special symbols. [Details](#)

## User self-registration

If you have Built-in authentication enabled with the "Allow user registration from the login page" setting on, the setting will be disabled on upgrade. If you need the registration, make sure the server is not open to unauthorized users access (e.g. not accessible from Internet) and enable the setting via the health item displayed at the top of the administration pages or in the "Administration | Authentication" under the "Built-in" module settings.

## Bundled Tools Update

- The IntelliJ IDEA Project Runner uses JPS 2017.3.4 requiring Java 1.8 as the minimal version.
- The bundled ReSharper CLT and dotCover have been updated to version 2018.1.2

## NuGet feed

- Configuration of the NuGet feed was moved from the server level to the project level: now each project can have its own feed. The "NuGet packages indexer" build feature can be added to build configurations whose artifacts should be indexed.
- The following NuGet feed-related build parameters are deprecated:
  - teamcity.nuget.feed.auth.server
  - teamcity.nuget.feed.server
  - system.teamcity.nuget.feed.auth.serverRootUrlBased.serverYou now need to explicitly specify the URL from the [NuGet Feed](#) page in the project settings.
- The enabled default NuGet feed with all published packages accessed by URL /app/nuget/v1/FeedService.svc/ is now moved to the Root project feed /app/nuget/feed/\_Root/default/v2/. It is recommended to switch to new URL in your projects.
- .nupkg files are now indexed on the agent side instead of the server which could slightly increase the time of builds for projects with the NuGet Feed feature and the automatic package indexing enabled or for builds with NuGet Packages Indexer build feature.

## REST API

REST API uses version 2018.1. The previous versions of the API are still available under /app/rest/2017.2, /app/rest/2017.1 (/app/rest/10.0), /app/rest/9.1, /app/rest/9.0, /app/rest/8.1, /app/rest/7.0, /app/rest/6.0 URLs. It is recommended to stop using previous APIs URLs as we are going to remove them in the following releases.

### Filtering builds by agent names

When agent name contains the parentheses symbols, instead of using `agentName:<name>`, use "agentName:(value:<value>)"

### Locators with "value:<text>"

Requests which used "value:<text>" locators (e.g. for matching properties) and no "matchType" dimension specification will start to use "equals" matching by default. Add "matchType:contains" to preserve the old behavior. [Details](#)

## VSS plugin is unbundled

The Visual SourceSafe plugin is no longer bundled with TeamCity but is available as a [separate download](#). Please contact our [support](#), if you still use this VSS for your builds.

## Other

- Commit Status Publisher supports Gerrit 2.6+ versions. For support for older Gerrit versions, please turn to our [support](#).
- When upgrading from TeamCity versions before 9.1, if TeamCity 2018.1 starts and agents are upgraded, but then you decide to roll back the server to the previous TeamCity version, the agents will not be able to connect back to the old server and will need to be reinstalled manually.
- Make sure that no HTTP requests from the agents to the server are blocked (e.g. requests to .../app/agents/... URLs)

## Changes from 2017.2.3 to 2017.2.4

The Inspections (.NET) and Duplicates Finder (.NET) build steps were renamed to Inspections (ReSharper) and Duplicates finder (ReSharper)

## Changes from 2017.2.2 to 2017.2.3

It is recommended to add "`teamcity.artifacts.restrictRequestsWithArtifactReferer=true`" [internal property](#) to enhance security of the server.

## Changes from 2017.2.1 to 2017.2.2

### Known issues

(Fixed 2017.2.3) If you use the Artifactory plugin and get the "Invalid RSA public key" browser message on opening build step settings, please apply the [workaround](#).

Under Windows, when TeamCity server is started as a service, "logs\teamcity-winservice.log" file is not created and server startup errors are nowhere to be seen. [Details](#)

### IDE Plugins

It is highly recommended to update the IDE plugins for all users to the latest version and then add the " teamcity.uploadPersonalPatch.requireAuthorization =true " [internal property](#) to enhance security of the server.

### Perforce VCS Root executable paths

Since TeamCity 2017.2.2, the field which specifies the path to p4 works only on the agent side, for agent-side checkout.

For the server, the p4 binary should be present in the PATH of the TeamCity server (or can be specified via the `teamcity.perforce.customP4Path` [internal property](#)).

The `teamcity.perforce.p4PathOnServerWhitelist` [internal property](#) can be used to specify a semi-colon-separated list of allowed p4 paths. The paths from this list can be set in VCS Root p4 path parameter for the server side (to restore old behavior).

### Mercurial VCS Root properties

Since TeamCity 2017.2.2, a number of Mercurial VCS root properties change their behavior for security reasons.

- the "HG command path" is used on the TeamCity server only if included into the [whitelist](#)
- the "Clone repository to" property is hidden if the VCS root doesn't have it already and is ignored by default. To make TeamCity display the property in all VCS roots, add the `teamcity.hg.showCustomClonePath=true` [internal property](#). The value of the VCS root property is respected only if it is included into the whitelist specified by the `teamcity.hg.customClonePathWhitelist` [internal property](#), which is a semi-colon-separated list of directories where a clone is allowed. Use `/path/to/dir/*` to allow clones to the child directories of the `/path/to/dir`.
- the "Mercurial config" is ignored on the server. If you need to enable some Mercurial plugins, please do that in the global `.hgrc` on the TeamCity server machine.

## Changes from 2017.2 to 2017.2.1

### Kotlin DSL changes

The versions in pom.xml were updated: `kotlin.version` is updated to 1.2.0, `teamcity.dsl.version` is updated to 2017.2.1. The dependency on `kotlin-stdlib` is replaced with the dependency on `kotlin-stdlib-jdk8` in order to provide access to additional functionality available in jdk8 (e.g. named groups in regexps). The dependency on the deprecated `kotlin-runtime` and the redundant dependency on `kotlin-compiler-embeddable` were dropped.

Now TeamCity provides a parent maven project for Kotlin DSL which defines the `teamcity.dsl.version` and `kotlin.version` properties. With such a parent project, you will not have to update your pom.xml after each TeamCity upgrade.

The easiest way to apply these changes is to run the 'Download settings in Kotlin format' action in project admin area and use the pom.xml from the zip produced by the TeamCity server.

### Bundled Java used in Docker images

The bundled Java used in Docker images has been updated to 8u151.

## Changes from 2017.1.x to 2017.2

### Known issues

(Fixed 2017.2.1) TFS in Java working mode (when Team Explorer is not installed on the machine) report "TFS subsystem was destroyed" errors. See [TW-52685](#) for details.

Upgrading using Windows installer can take significant time if your TeamCity installation directory contains lots of nested directories (e.g. TeamCity Data Directory is under it). The long stage can occur after "Extract: Uninstall.exe..." progress message. In you encounter this long step, please wait for the completion of the operation (the installer runs `icacls.exe` utility as

a nested process). To prevent the issue it is recommended to [move the data directory](#) out of TeamCity server installation home.

## Perforce branch specification change

There is a breaking change which requires your action if you use Perforce streams with enabled feature branches and you're using a non-default branch filter.

Starting from TeamCity 2017.2, Perforce VCS Roots use the same format for Perforce streams and TeamCity feature branches specification.

In the VCS Root branch specification for Perforce, `+:stream_name` must now be replaced with `+://stream_depot/stream_name`. Also, for better presentation of stream names in the UI, you may want to replace the default branch specifications like `+:*` with `+://your_stream_depot/*`.

This change was made in the scope of fixing [TW-48038](#).

## Server process restart

Now if the server process is stopped unexpectedly or killed, the process will automatically restart. The server should be stopped using the `teamcity-server.bat/sh stop` command which performs a graceful stop.

## Bundled plugins

### Docker integration plugin

The Docker integration plugin is bundled since TeamCity 2017.2.x. If you installed the plugin for the previous version manually, please [remove it](#).

### .NET CLI plugin

The .NET CLI (.NET Core) plugin is bundled since TeamCity 2017.2.x. If you installed the plugin for the previous version manually, please [remove it](#).

During upgrade all existing .NET Core build steps will be converted into .NET CLI steps and existing .NET Core plugin will be disabled.

Note: The `DotNetCore` and `DotNetCore_Path` agent configuration parameters will be changed to `DotNetCLI` and `DotNetCLI_Path`; please consider updating your agent requirements which depend on these parameters.

## REST API

REST API uses version 2017.2. The previous versions of the API are still available under `/app/rest/2017.1` (`/app/rest/10.0`), `/app/rest/9.1`, `/app/rest/9.0`, `/app/rest/8.1`, `/app/rest/7.0`, `/app/rest/6.0` URLs.

### buildType entity

has "templates" sub-element now instead of "template" to support multiple templates.

### build entity

No longer expose boolean "running" attribute, textual "state" attribute with value "running" is used instead.

## Windows Versions Support

Windows XP and Vista are no longer the supported versions of Windows for the TeamCity Server and Agent. While the server and agent can still work on these old versions, we do not target the versions during our development. Let us know if the support for the versions is important for your TeamCity usage.

## J2EE Servlet 2.5 container is no longer supported

J2EE Servlet container version 2.5 is not supported since TeamCity 2017.2. TeamCity does not guarantee support for Tomcat 6.x and Jetty 7.x implementing Servlet 2.5. For .war distribution (not recommended, .tar.gz distribution is recommended), TeamCity supports Apache Tomcat 7+, J2EE Servlet 3.0+ and JSP 2.2+.

## Other

- The bundled Tomcat 8.5. restricted usage of special characters in the URL including curly bracket symbols (`{ }`). [Details](#)
- TeamCity integration with IntelliJ-based IDEs no longer supports StarTeam and Visual Source Safe version controls.

## Changes from 2017.1.4 to 2017.1.5

The bundled JetBrains dotCover has been updated to version 2017.2

The SSH Agent build feature started to report a build problem if it fails to start an SSH agent with the specified SSH key (in the scope of [TW-42707](#)). Previously errors were only logged, but not reported as build problems. As a result, builds with invalid SSH Agent settings would start to fail after the upgrade.

## Changes from 2017.1.3 to 2017.1.4

### Known issues

TFS Personal support lists all build configurations for TFVC VCS root. See [TW-51497](#) for details.

## Changes from 2017.1.2 to 2017.1.3

[TW-50148](#) was fixed and the DSL API documentation was improved. If you need these changes for local development, please update the [maven dependency version](#) to 2017.1.3.

Now TeamCity server runs 'git gc' automatically to improve performance of git operations. This requires a git client to be installed on the server and be available to the server via the PATH environment variable. If a native git client cannot be found, then the corresponding health report is shown. For TeamCity to find the git client, the client needs to be installed on the server machine and added to \$PATH (the server restart is required afterwards). Instead of modifying PATH, the path to the git client can be specified via the `teamcity.server.git.executable.path` [internal property](#).

## Changes from 2017.1.1 to 2017.1.2

No noteworthy changes.

## Changes from 2017.1 to 2017.1.1

The bundled IntelliJ IDEA has been updated to 2017.1.2

## Changes from 10.0.x to 2017.1

### Known issues

Editing cloud profile cancels all builds on profile agents. See [TW-49616](#) for details.

### TeamCity Versioning Changes

Since 2017, TeamCity adopts the common JetBrains versioning scheme that identifies versions by year following the pattern: "<year>.<number of the feature release within the year>.<bugfix update number>". The current version is TeamCity 2017.1 formerly known as TeamCity 10.1.

### Update settings in Kotlin DSL

In this version TeamCity settings format has been changed and if your settings are stored in Kotlin DSL, you might need to update [Kotlin DSL scripts](#) before continuing to use them. Check any related server health reports after the server upgrade.

### CSRF Protection: Modifying GET requests and Proper Proxy configurations

TeamCity now implements [CSRF protection](#) to improve web UI security and this introduces several changes in behavior which might affect your installation. In particular:

- If you use a reverse proxy before TeamCity, the proxy should not change the original "Host" request header (this typically requires configuring the proxy to set the Host header to the original request value). Also, it should not modify the "Origin" and "Referer" headers present in the original request. While this has been the recommended setup for a long time, now it becomes critical for the TeamCity web UI functioning;
- if you use non-bundled clients which perform HTTP GET requests to TeamCity, some of the GET requests (those which change the state of the server, like `http://server/action.html?add2Queue=XXX`) stop working in 2017.1, please change the requests to use POST instead of GET;
- the non-browser clients which reuse authentication by supplying the TCSESSIONID cookie with the request, need to be updated to supply the "Origin" HTTP header with the value the same as the host the request is being sent to.

If the check fails, you get the HTTP 403 response with the details of the failed check. The details are also logged into `teamcity-auth.log`.

## Old IPR Runner

The old (TeamCity 6.0) IPR runner has been removed from the TeamCity. It was deprecated and has not been offered as an option since TeamCity 6.0, and now it is gone completely (the corresponding build configurations will not run anymore).

## Log4j configuration

It is recommended to overwrite the server's `conf\teamcity-server-log4j.xml` file with the content of the `conf\teamcity-server-log4j.dist` file which represents the default logging configuration which has changed in this release. If you do need a custom logging configuration, consider using [logging presets](#) instead of modifying `conf\teamcity-server-log4j.xml`.

The same overwriting from the sibling `.dist` file is recommended for the TeamCity agents.

The `conf\teamcity-*-log4j.xml.dist` file is created after the first start of the upgraded TeamCity version.

## Builds metadata storage (NuGet feed)

Builds metadata storage will be re-created and builds will be reindexed right after the upgrade. As a result, immediately after the upgrade, the TeamCity internal NuGet feed will not contain all of the packages.

During the builds reindexing, the following messages may appear in the `teamcity-server.log`:

```
INFO - .index.BuildIndexer (metadata) - Enqueued next 100 builds for indexing, builds left: 2000, last build id: 3813
INFO - .index.BuildIndexer (metadata) - Enqueued next 100 builds for indexing, builds left: 1900, last build id: 3713
```

"builds left:" indicates the number of builds left to process. Note that TeamCity starts reindexing from the most recent builds, so all fresh builds should appear in the TeamCity NuGet feed in a relatively short time.

To increase the metadata indexing speed you could use the [following tip](#).

## REST API

REST API has only minor changes, so the same API is exposed under the `app/rest/10.0` and `/app/rest/2017.1` URLs. API version has been updated to 2017.1 though to reflect the changes.

The build's node "triggeredBy" now has more correct values of "type" attribute for the builds started after 2017.1 upgrade. In particular, the "buildType" value is not used anymore, the "finishBuild", "snapshot", etc. values are used instead.

## Visual Studio Add-in fails to install from TeamCity UI

As a workaround could be used [ReSharper web installer](#). See [TW-51680](#) for details.

## Changes from 10.0.4 to 10.0.5

If you are using TFS with agent-side checkout, note that due to the fix of [TW-48555](#) TeamCity will have to re-create TFS workspaces, which may result in a clean checkout on the agents after the upgrade.

## Changes from 10.0.3 to 10.0.4

### Precedence of %dep.ID.NAME% parameter references

When `%dep.ID.NAME%` is used and there several different builds accessible via (direct or indirect) artifact dependencies, the result of the reference resolution could have used any of the builds without any guaranteed precedence.

Since 10.0.4 `dep.` parameter resolution works as follows:

1. if there is a snapshot dependency, the build from the same chain wins.
2. if there is no snapshot dependency and several builds are accessible via an artifact dependency, the build with a greater `buildId` wins. If there are several artifact dependencies from a single build configuration, only the first one is considered.

## Updates

AWS SDK has been updated to 1.11.66 to support new instance types (r4.4xlarge, f1.16xlarge, t2.2xlarge, t2.xlarge,

r4.2xlarge, r4.xlarge, r4.large, r4.16xlarge, r4.8xlarge, f1.2xlarge).

## Changes from 10.0.2 to 10.0.3

### Amazon EBS-Optimized Instances

The behavior of [EBS-optimization](#), enabled by default since TeamCity 10.0, is changed similarly to what EC2 console offers:

- 1) EBS-optimization is turned on by default for c4.\* , m4.\* and d2.\* (non-configurable)
- 2) EBS-optimization is turned off by default for any other instance types.
- 3) EBS-optimization can be turned on for instances that support it (such as c3.xlarge, etc.) by checking the appropriate box when configuring the image of the Amazon cloud profile

### Bundled tools updates

The bundled dotCover has been updated to version 2016.2.2

## Changes from 10.0.1 to 10.0.2

- The known issue mentioned for 10.0.1 is fixed.
- The bundled JetBrains dotCover has been updated to version 2016.2.
- Jabber integration is now [more restrictive](#) with regard to SSL connection checking. If you are using jabber.org for sending notifications, and face a problem regarding SSL certificate, you should either enable "Use legacy SSL" option or (better) change JVM version for running TeamCity to at least 1.8.0\_101.
- Precedence of %dep.ID.NAME% parameters resolution was changed unintentionally in case of several different builds used as dependency. See [TW-47518](#) for details.

## Changes from 10.0 to 10.0.1

All known issues mentioned for 10.0 are fixed.

### Known Issues

(fixed in 10.0.2) TeamCity server temp folder can fill up if an agent tool is installed as a directory in <TeamCity Data Directory>/plugins/.tools. [Details and workaround](#).

## Changes from 9.1.x to 10.0

### Known Issues

(These known issues are fixed in 10.0.1)

Failed to collect TFS changes - From version <x> is greater then current version <y>

If you use TFS version control and get "Error collecting changes for VCS repository ... Failed to collect TFS changes - From version x is greater then current version y" error, either commit a new change so that it appears in the affected build configuration, or install an [updated plugin](#) ([related issue](#)).

Project administrator may not be able to re-define parameters inherited from the parent project

See request [TW-46372](#) for details and possible workaround.

Upgrade form TeamCity version before 8.1 fails with "Can't take exclusive lock when db lock is not held"

See request [TW-46385](#) for details and possible workaround.

Subversion VCS roots with svn+ssh:// protocol can report "Host key (xxx) can not be verified."

See request [TW-46489](#) for details and for the plugin with the fix

### Changes in agent properties reporting .NET 4.x runtime

TeamCity agents before version 10 used to report DotNetFramework4.0\_\* properties whenever any of 4.x versions of .NET framework were installed on the agent. Since TeamCity 10, DotNetFramework4.0\_\* properties are reported only when 4.0 runtime (without updates) is installed. For 4.5.\* , 4.6.\* updates corresponding DotNetFramework4.N\_\* properties are reported. This change in behavior allows for more precise requirements definition.

If after the upgrade you get incompatible agents with Unmet requirement: Exists=>DotNetFramework4.0\_x86(/x64) exists message, review the explicit requirements in your build configuration. If your build is compatible with any .NET 4.x runtime (it is a most common case) please use Exists=>DotNetFramework4.\*\_x86(/x64) requirement. If you would like to run on .NET 4.5+ agents - use Exists=>DotNetFramework4.(5|6).\* requirement.

Some third-party plugins are known to be affected. On the upgrade, make sure to upgrade xUnit plugin to version 1.1.2+.

#### Ignored tests table optimization

During upgrade TeamCity will optimize data in the ignored\_tests table (we do that in order to speedup the TeamCity built-in backup / restore process). In some rare cases, when this table contains millions of rows, the process of table optimization may take significant time - possibly a few hours. Among other data, the table contains the reasons why a particular test in a build was marked as ignored. If this information for old builds is not very important for you, you can start TeamCity server with the additional **JVM option**:

```
-Dteamcity.truncateIgnoreReasonConverter.copyReasons=false
```

In this case TeamCity will not copy the ignore reasons into the new, optimized table, and this particular step of the upgrade process will run much faster.

#### Java 8

Starting from TeamCity 10, the TeamCity server requires Java 8 JRE/JDK (included in the Windows .exe distribution).

TeamCity agents currently require Java 1.6+, but starting from the next TeamCity version, the minimum requirement for [the Java on the agent](#) will be Java 8 (included in agent's Windows .exe distribution). It is recommended that you now consider upgrading the agents Java.

#### Java memory options change

It is recommended to remove the " -XX:MaxPermSize=..." JVM option from TEAMCITY\_SERVER\_MEM\_OPTS environment variable, if previously configured. (This is due to the fact that Java 8 **does not use** permanent generation (PermGen) anymore)

#### Agent requirements and artifact dependencies disabling

Agent requirements and artifact dependencies can be disabled now. TeamCity plugins and REST API- based code using a version of API prior to TeamCity 10 is likely to ignore the disabled status of these settings.

#### TFS

TeamCity comes with cross-platform TFS integration: to work with TFS, you no longer need to install a TeamCity server on a Windows machine.

#### Visual Studio Online Work Items plugin

Visual Studio Online Work Items plugin is obsolete since TeamCity 10.0 and can be safely removed. TeamCity 10.0 has a built-in integration with [Team Foundation Work Items](#) which supports TFS 2010+ and Visual Studio Team Services. After upgrade, TeamCity will detect the existing issue tracker connections of this plugin and convert them into TFS Work Items.

#### Default Checkout mode for newly created build configurations

The default setting for the VCS checkout mode on creating new build configurations has changed: now TeamCity will check out the sources on the agent before the build. If the agent-side checkout is not possible, TeamCity will use the server-side checkout. Explicit server-side or agent-side checkout is still in place. The new default applies only to newly created build configurations; all the existing ones will work as configured before.

#### Project-based Agent Management Permissions

New TeamCity installations now have different agent management permissions assignments: Project Administrator role does not include (global) Agent Manager role. Instead, Project administrator role has [agent-project permissions](#) which allow to manage agents from the agent pools with only projects where user has Project Administrator role.

Existing installations are not affected by this change in order not to change the user permissions. However, it is recommended to review the Project Administrator role and consider excluding "Agent Manager" role and adding the following permissions:

- Enable / disable agents associated with project
- Start / Stop cloud agent for project
- Change agent run configuration policy for project
- Administer project agent machines (e.g. reboot, view agent logs, etc.)

- Remove project agent
- Authorize project agent

## UI Changes

### Server Administration UI

The new Administration | Tools page allows setting up tools to be used by appropriate plugins. Tools are automatically distributed to all build agents and can be used in related runners.

#### New Create project / Create build configuration buttons

The new Create subproject and Create build configuration buttons have a drop-down now allowing you to select whether you want to create a project from scratch (manually), from URL, or using the popular version control systems GitHub.com and Bitbucket.

#### NuGet-related UI

NuGet settings page is removed. NuGet.exe can be installed using the new Tools page; to set up TeamCity as a NuGet Server, go to the Administration | NuGet Feed page.

#### Tests-related UI

The Problematic Tests tab is no longer available and the View all tests failed within the last 120 hours link is removed from the Current Problems tab.

TeamCity now detects [Flaky tests](#) displayed on the dedicated tab for a given project.

#### Visual Studio Add-in

The legacy version of the TeamCity Visual Studio Add-in is no longer supported. Visual Studio 2005 and 2008 are not supported.

TeamCity Visual Studio Add-in is shipped [as a part of ReSharper Ultimate](#). After installation, the TeamCity Add-in will be available under the RESHARPER menu in Visual studio.

Note that the installer will remove the pre-bundle products versions: TeamCity and ReSharper versions prior to 9.0, dotCover prior to 3.0, dotTrace prior to 6.0.

ReSharper Ultimate does not support the Visual Studio versions 2005 and 2008.

#### IntelliJ IDEA Compatibility

IntelliJ IDEA 12.1 and older as long as other IntelliJ-based products released prior to year 2013 no longer supported by [IntelliJ Platform Plugin](#).

#### Snapshot dependencies builds rebuilding

After the server upgrade, the builds used as snapshot dependencies may be rebuilt once even if the snapshot dependency has the "Do not run new build if there is a suitable one" option set to ON. This is done to fix the [issue](#).

#### Perforce

Clean checkout will be enforced in builds with Stream-based and Client-based Perforce VCS Roots.

#### Subversion

Starting from TeamCity 10, TeamCity does not accept by default connections to SVN servers accessed by https:// protocol with non-trusted server SSL certificate. To enable access with such certificates, you should either import the certificate to server JVM keychain, or enable VCS Root option "Accept non-trusted SSL certificate" (Enable non-trusted SSL certificate in 10.0) ([issue](#)).

#### Bundled tools updates

Ant runner: the bundled Ant distribution has been upgraded from 1.9.6 to 1.9.7

.NET dotCover coverage: the bundled dotCover is updated to 2016.1

ReSharper command line tools: the bundled R# CLT is updated to 2016.1

Java inspections and duplicates: the bundled IntelliJ IDEA is updated to 2016.2

## GitHub Issue Tracker

If you were using the TeamCity-GitHub [third-party plugin](#) prior to TeamCity 10.0, you can safely [remove](#) it: the built-in TeamCity integration will detect the existing connection to GitHub issue tracker and pick up your settings automatically.

## NuGet Support

Configuration parameters `teamcity.tool.NuGet.CommandLine.%NUGET_VERSION%.nupkg` are not reported anymore. `teamcity.tool.NuGet.CommandLine.%NUGET_VERSION%` parameters should be referenced instead.

e.g instead of using `%teamcity.tool.NuGet.CommandLine.DEFAULT.nupkg%` parameter reference `%teamcity.tool.NuGet.CommandLine.DEFAULT%` should be used.

## Build Statistics

Several statistic values (metrics) has been reworked and renamed:  
BuildCheckoutTime into buildStageDuration:sourcesUpdate  
BuildArtifactsPublishingTime into buildStageDuration:artifactsPublishing  
ArtifactsResolvingTime into buildStageDuration:dependenciesResolving

Old keys are still supported in charts definitions.

## REST API

REST API uses version 10.0. The previous versions of the API are still available under `/app/rest/9.1`, `/app/rest/9.0`, `/app/rest/8.1`, `/app/rest/7.0`, `/app/rest/6.0` URLs.

Requests for a set of items with the locator addressing a single item which resulted in 404 responses previously will now return an empty set as a more consistent approach. For example, "`.../app/rest/builds?locator=id:<non-existent build id>`". REST debug logging might have diagnostics message with more details as to the case.

Requests for set of items can return not all/incomplete results (with zero or more items included) and provide "`nextHref`" sub-element with the link to retrieve the next "page" of items. The search result is complete when no "`nextHref`" sub-element is provided.

Some requests for set of items (e.g. `.../app/rest/vcs-roots` and `.../app/rest/vcs-root-instances`) will use paged results by default when queried without a locator (they used to list all the items). Add the "`count:NNN`" locator dimension to set page size.

### Finding builds (`.../app/rest/builds/...` URL)

When performing builds scan to find those matched by the locator specified, by default for performance reasons TeamCity will return partial result limited by scanning only 5000 most recent builds. To process a larger portion of the history, check the "`nextHref`" attribute returned or set the "`lookupLimit`" locator dimension to a larger value.  
Previously, until specifically requested the builds from non-default branch as well as canceled, personal and failed to start builds were not returned. Now these filtered out builds are returned by default for running and queued builds queries as well as when filtering by agent or user. Use "`defaultFilter:true/false`" locator dimension to manage the default filtering explicitly. Also, "`number:NNN`" locator now adheres to the same default logic: only "usual" finished builds from default branch are searched by number and several builds can be returned if found.

### Finding VCS roots (`.../app/rest/vcsRoots/...` URL)

(minor)A VCS root locator with `project` and `buildType` specified used "project" as the context for finding "buildType". This is no longer the case, the `buildType` locator should be full one to find the build configuration.

### Build Configuration's Artifact Dependencies (entities returned by `.../app/rest/buildTypes/...` URL)

The "`artifact-dependencies`" sub-element of the `buildType` element now uses textual generated ids instead of numeric ones which depended on the order previously. This also affects requests for artifact dependencies modification.  
The "`agent-requirements`" sub-element of the `buildType` element now uses generated ids instead of the parameter name as id. This also affects requests for agent requirements modification.

### Editing agent requirements (`.../app/rest/buildTypes/.../artifact-requirements/...` URL)

Previously, on adding a new agent requirement for the same parameter, the existing one was overridden by the new one; now a new one is added.

Previously, on adding a new agent requirement, the parameter name was derived from the "`id`" attribute of the "agent-requirement" node. Since TeamCity 10, the parameter name is derived from the "`property-name`" property.

### Test and problem occurrences (`.../app/rest/testOccurrences`, `.../app/rest/problemOccurrences` URL)

The sorting of the returned results is changed for some of the queries compared to the previous versions. For example, the ".../app/rest/testOccurrences?locator=build:(xxx)" request now returns the tests in the order they were run in the build. Previously, test occurrences were sorted by the new status and then by name. Problem occurrences were sorted by problem id. Also, the "build" dimension in the test/problem-related locators now supports multiple builds so for the requests which matched several builds via the "build" dimension, all the builds will be processed; previously only the first matching build was processed.

#### Entities

The "`property`" entity used to have its "own" Boolean attribute indicating whether the parameter is redefined in the build configuration as opposed to inherited from a template/project. Now the attribute is renamed to '`inherited`' and its value is inverted.

The "`vcs-root`" and "`vcs-root-instance`" used to have "status" and "lastChecked" attributes. Now VCS root instance has "status" element (with `current` node which has `status` and `timestamp` attributes) and VCS root does not have the data as it produced undefined results in case of several VCS root instances.

The "`test`" entity "id" became String instead of Long (due to [inability](#) to represent some Java Long value in JavaScript)

#### Build type and template

The "`settings`" node used to contain all the supported settings. Now only those defined in the build configuration or template are present. The same is true for .../app/rest/buildTypes/XXX/settings/\* requests: only the values changed from defaults are present.

#### Compatible agents

When querying for compatible agents, only the agents which can actually run the builds are now returned. By default, unauthorized, disconnected and disabled agents are not listed. This behavior differs from that in previous versions which had a number of discrepancies. Affected requests and entities: .../app/rest/agents?locator=compatible:(...); .../app/rest/agents/.../compatibleBuildTypes and incompatibleBuildTypes; nested nodes Agent.compatibleBuildTypes, QueuedBuild.compatibleAgents, BuildType.compatibleAgents

## Changes from 9.1.6 to 9.1.7

No noteworthy changes.

## Changes from 9.1.5 to 9.1.6

### Known Issues

There is a [known issue](#) in the bundled dotCover run on Windows XP and Vista. You can use the [hotfix](#) or the [workaround](#) provided. The issue will be fixed in the next dotCover release.

There is a [known issue](#) with [NuGet Credentials](#) not working for the Authenticated Feed URL of the internal TeamCity NuGet server on the local agents. As a workaround, instead of using `%teamcity.nuget.feed.auth.server%`, please specify the external server URL in the build step and the build feature.

### NuGet

NuGet versions prior to 2.8.6 require .Net Framework 4.0+ installed on the build agent, NuGet 2.8.6 and later requires .NET 4.5.

### NUnit

Since version 9.1.6, TeamCity does not support NUnit 3 beta versions (released before NUnit 3.0.0).

The "Run a process per assembly" option of the [NUnit runner](#) has been removed from NUnit 3 settings. Configure the desired behavior using the required [command line options](#) in the [corresponding field](#).

### Bundled tools updates

Bundled IntelliJ IDEA updated to version # 143.1945 (roughly equivalent to 15.0.3 with a few additional fixes).

Bundled version of Maven 3.2.x updated to 3.2.5.

### Performance Monitor

Note on permissions: to [monitor performance](#) of a build agent run as a service as a Windows service, make sure the user starting the agent is member of the Performance Monitor Users group.

## Changes from 9.1.4 to 9.1.5

### Known Issues

There is a [known issue](#) in the bundled dotCover run on Windows XP and Vista. You can use the [hotfix](#) or the [workaround](#) provided. The issue will be fixed in the next dotCover release.

### Product icons

JetBrains product icons are updated in accordance with the [new JetBrains branding](#).

### Git

Since TeamCity 9.1.5, `git sparse-checkout` is disabled by default. To enable it in a TeamCity project, add the `teamcity.git.useSparseCheckout=true` parameter to this project.

### Gradle: Breaking change compared to 9.1.2

Gradle runner `system.* properties`, introduced in TeamCity 9.1.2, defined for the build as JVM's system properties of the Gradle process, will not work since 9.1.5. Now the TeamCity system properties can be accessed in Gradle scripts as Gradle properties (similarly to the ones defined in the `gradle.properties` file) and are to be referenced as follows:

- a) name allowed as a Groovy identifier (the property name does not contain dots): `customUserProperty`
- b) name not allowed as a Groovy identifier (the property name contains dots, e.g. `build.vcs.number.1`): `project.ext["build.vcs.number.1"]`

### Bundled tools updates

The bundled JetBrains IntelliJ IDEA (IDEA inspections and duplicates) has been updated to version 15.0.2

### .Net tools updates

JetBrains ReSharper command line tools (.NET inspection and duplicates) have been updated to match ReSharper 10.0.2 release

TeamCity Visual Studio Addin Web installer updated to ReSharper 10.0.2 release

Bundled JetBrains dotCover updated to version 10.0.2

## Changes from 9.1.3 to 9.1.4

### Known Issues

Certain roles/permissions configurations can result in error loading roles and no ability for regular users to view projects. In such cases the "Circular reference is detected between roles" critical server error is displayed on the server administration pages for those logged in as server administrator. Please check the [workaround](#) for the issue.

Git agent-side checkout may malfunction (details at [TW-43202](#)) when the `teamcity.git.use.native.ssh=true` parameter is specified in a build configuration or in the agent config. To fix that, install the `#snapshot-34` build of the Git-plugin.

Git agent-side checkout works incorrectly with git client versions 1.7.0-1.7.4: the checkout directory contains files only, all directories are missing (details at [TW-43330](#)). To workaround the problem, add the `teamcity.git.useSparseCheckout=false` parameter in the Root TeamCity project.

### TeamCity Windows binaries signatures

Since 9.1.4 the TeamCity Windows binaries are signed with SHA-2 code-signing certificate following the [Microsoft SHA-2 policies](#). This means that on systems prior to Windows XP SP3, the executables will not pass code signing verification; newer Windows systems require the corresponding security update from Microsoft.

### Bundled tools updates

Bundled Oracle JRE (in both Server and Agent.exe installers) has been updated to version 1.8.0\_66 (32-bit)

### .Net tools updates

JetBrains ReSharper command line tools (.NET inspection and duplicates) have been updated to match ReSharper 10.0 release

TeamCity Visual Studio Addin Web installer updated to ReSharper 10.0 release

Bundled JetBrains dotCover updated to version 10.0

## Changes from 9.1.2 to 9.1.3

### Known Issues

There is a [known issue](#) in the bundled dotCover 3.2 which could cause a build's failure with the following exception: "System.Security.VerificationException: System.Security.VerificationException: Operation could destabilize the runtime". The issue is fixed in dotCover 10.0 bundled with TeamCity 9.1.4 release.

Bundled JVM (Server Windows installer and Agent Windows installer) is updated which results in disabled SSL RC4 cipher suite for outgoing HTTPS connections. For example this makes connections to CloudForge SVN servers non-functional with "SSLHandshakeException: Received fatal alert: handshake\_failure" error. ([details](#))

## Changes from 9.1.1 to 9.1.2

### Known issues

The Command line runner can fail to execute a custom script if it has a non-default hashbang specified at the beginning of the script: [TW-42498](#)

When using Amazon EC2 cloud integration, an AMI-image, containing build agent versions 9.1.2-9.1.5 will appear twice with name EC2-i-abcdefg and EC2-i-abcdefg-1 consuming 2 licenses. To fix the issue please use an AMI-image with agent 9.1.6+. Just updating server to 9.1.6 won't help, if AMI remains the same. ([details](#))

### Build status icons

Build status icons updated to a more "standard" look and are of a bit larger now.

### Bundled tools updates

JetBrains ReSharper command line tools (.NET inspection and duplicates) have been updated to match ReSharper 9.2 release

TeamCity Visual Studio Addin Web installer updated to ReSharper 9.2 release

Bundled JetBrains dotCover updated to version 3.2

Bundled Oracle JRE (in both Server and Agent .exe installers) updated to version 1.8.0\_60 (32-bit)

## Changes from 9.1 to 9.1.1

Bundled Jacoco coverage library updated to version 0.7.5

Perforce VCS Roots with disabled ticket authentication won't run 'p4 login' operation anymore if password authentication is disabled on the Perforce server.

I.e. if password authentication is disabled, "Use ticked-based authentication" option must be enabled on the VCS Root. [TW-42818](#)

## Changes from 9.0.x to 9.1

### Bundled Ant

Bundled Ant distribution has been upgraded form 1.8.4 to 1.9.6. Note that Ant build steps using bundled Ant will use another version of Ant after the server upgrade. Ant 1.9.6 requires Java 1.5 at least, so builds using Ant and running under Java 1.4 will stop working.

MSTest runner converted into Visual Studio Tests runner

[MSTest runner](#) is merged with [VSTest console runner](#) (previously provided as a separate plugin) into the [Visual Studio Tests runner](#).

Note that after upgrade to TeamCity 9.1, MSTest build steps are automatically converted to the Visual Studio Tests runner steps, while VSTest steps remain unchanged.



If you have used [VSTest.Console runner plugin](#), make sure that you have latest version (build 32407) installed. The plugin version can be viewed on [Administration->Plugins List](#) page. Earlier versions of this plugin are not

[compatible with TeamCity 9.1](#) and may cause malfunction of .Net related build runners which can manifest with "java.lang.NoSuchMethodError: jetbrains.buildServer.runner.NUnit.NUnitVersion.parse(Ljava/lang/String;)" build errors. The plugin can be downloaded from [its page](#).

Consider migrating your vstest.console execution steps to the bundled Visual Studio Tests runner.

## MSTest installation agent properties

TeamCity agent automatically detects the installed MSTest and used to expose the locations in `system.MSTest.N.N` system properties.

Since TeamCity 9.1, the locations are exposed via `teamcity.dotnet.mstest.N.N` configuration parameters. Check [TW-41845](#) for a workaround if you cannot easily change the properties usage.

## Nested test reporting

Previously TeamCity supported a case when one test could have been reported from within another test using [service messages](#). Now, after the fix of [TW-40319](#), starting another test finishes the currently started test in the same "flow". To still report tests from within other tests, you will need to specify another `flowId` in the nested test service messages.

## REST API

REST API uses version 9.1. Previous versions of API are still available under `/app/rest/9.0`, `/app/rest/8.1`, `/app/rest/7.0`, `/app/rest/6.0` URLs.

### Finding builds

**Summary (tl;dr):** Some build filtering rules has subtle changes. Most importantly, a queued build can now be returned instead of 404 when searching by build id and meaning of the "project" locator dimension has changed to be not recursive. Also, failed to start builds are now not included until "failedToStart:any" locator dimension is specified. Details:

Affected requests: `/app/builds/<locator>...`, `/app/builds?locator=<locator>`, `/app/buildTypes/<btLocator>/builds` and others with build locator

**locator: id:<number> or taskId:<number>**

- previously, if the matching build was a queued one, 404 (Not Found) was returned
- now the queued build is returned

**locator: project:<id>...**

- previously, all the builds belonging to build configurations of the project and all it's subprojects (recursively) were found
- now only the builds belonging to build configurations of the project specified are found. For finding the builds recursively, use "affectedProject:<id>." dimension. This makes the usage consistent with build type locators.

**locator: tag:<text>**

- previously, when "<text>" used ":" character, that used to treat the entire "<text>" as tag name
- now the "<text>" is parsed as a nested locator. For searching tags with ":" character, locator "tag:(name:(<tag>))" should be used

**locator: <text>**

- previously, if <text> is not a number, response was 400 (Bad Request) with "LocatorProcessException: Invalid single value: '<text>'. Should be a number." message.
- now search by build number across all builds on the server is performed (this is not recommended to be used on production servers). For not found builds 404 (Not found) response is returned

**locator: id:<number>,xxx:yyyy**

- previously, build was found by id "<number>", other dimensions were ignored
- now if the build found by id does not match other dimensions, response is 404 (Not Found)

**locator: agent:<agentLocator>**

- previously <agentLocator> was used as is, without applying any defaults (unauthorized agents were included until specifically excluded)
- now <agentLocator> has the same behavior as in `/app/rest/agents` request: unauthorized agents are excluded by default

### Finding projects

Searching project by name used to return 404 error if several projects were matched. Now will return the first project found.

### Build's artifacts

There are several bugs fixed in listing build artifacts via `/app/rest/builds/<locator>/artifacts/*` requests which can cause subtle

changes in the results for the request. Check the new behavior if you relied on the response.  
The most important changes are:

- the initial path specified via URL part is searched for without current locator value, it will not generate 404 responses until there is no such artifact on disk.
- archives are not treated as directories (do not have children elements) by default. Specify "browseArchives:true" to treat archives as directories (in "recursive:true" mode only one level of archives is treated as directories)

#### Agents

Agents which are not known to the system (which were deleted) used to have id "-1". Now "id", "pool" and some other entries are no longer included for such agents.

#### Xcode 7 Support

[Experimental support for Xcode 7](#) has been added.

#### Issue trackers integration

Due to API changes, third party issue trackers integration plugins might not be compatible with TeamCity version 9.1. Old plugins will not work and will report "java.lang.NoSuchMethodError:  
`jetbrains.buildServer.issueTracker.AbstractIssueProviderFactory.<init>(Ljetbrains/buildServer/issueTracker/IssueFetcher;Ljava/lang/String;)V`" error in `teamcity-server.log` log (more details in the [issue](#)). If you observe such errors, please contact the [plugin authors](#). If you are the author of affected plugin, please refer to the related notes in [Open API Changes](#).

### Changes from 9.0.4 to 9.0.5

No noteworthy changes.

### Changes from 9.0.3 to 9.0.4

No noteworthy changes.

### Changes from 9.0.2 to 9.0.3

No noteworthy changes.

### Changes from 9.0.1 to 9.0.2

No noteworthy changes.

### Changes from 9.0 to 9.0.1

#### Known Issues

If you have enabled versioned settings for projects which use meta-runners in TeamCity 9.0, on upgrade and following commit into the settings VCS root, the meta runners will be deleted from the server. Workaround is to commit the meta-runners definitions into the settings repository manually. Related issue: [TW-39519](#).

Oracle 10.x JDBC driver is not supported anymore

Due to missing support for national character sets (nvarchar) in Oracle 10.x JDBC drivers, TeamCity 9.0.1 will ask to upgrade Oracle JDBC drivers to the latest version. The minimal supported version of Oracle JDBC driver is 11.1.

### Changes from 8.1.x to 9.0

#### Known Issues

- If you have custom artifact cleanup rules configured which mention ".teamcity" directory, build logs can be deleted by the cleanup procedure. Make sure you have build logs backup before upgrade and remove all the custom artifacts cleanup rules with ".teamcity". Related issue: [TW-40042](#). This issue is fixed in 9.0.3 release.
- If you use Microsoft SQL Server database with TeamCity, after the scheduled cleanup background run, TeamCity UI pages can lock until the server restart. See [TW-39549](#) for details. This issue is fixed in 9.0.2 release.
- If you use LDAP authentication on the server and there are lots of login attempts on the server (e.g. there is an active REST-using script), OutOfMemory errors can occur and require server restart. Consider installing an LDAP plugin with a fix from the [issue](#). This issue is fixed in 9.0.1 release.
- If you have large Maven projects, you can see builds failing with OutOfMemoryError. This is caused by update of back-end embedded Maven to 3.2.3 which has bigger memory footprint. Consider increasing Build Agent [memory limits](#)

Related issue: [TW-41052](#)

#### UUID in XML settings files

Since TeamCity 9.0, disk-stored XML settings definitions of projects, build configurations and VCS roots have unique non-human-readable id (uuid) stored. These ids are automatically generated and are assumed to globally unique. On settings files copying, you need to change/make unique not only id (file name) and name (across siblings) of the entity , but also remove it's uuid from the file. TeamCity will generate new uuid automatically.

#### Build logs storage

The location of the build logs in the internal format stored under [TeamCity Data Directory](#) has changed. The build log files in internal format are now stored under hidden build artifacts.

Namely, the location has changed from `system/messages/CHyy/xxyy.*` to `system/artifacts/<PROJECT EXTERNAL ID>/<BUILD CONFIGURATION NAME>/xxyy/.teamcity/logs/buildLog.*`.

Old build logs are migrated to the new location on TeamCity server startup ([TW-37362](#)). To avoid this migration, `teamcity.skip.logs.migration` internal property should be set before server startup.

#### Builds re-indexing after upgrade

On the first server start after upgrade from a version prior to 9.0, the server will reindex all builds for the purpose of builds search functionality and NuGet feeds. During the indexing time, some builds will not be available in the search results and in NuGet feeds. The server can also behave in less performant manner way during the indexing. `teamcity-server.log` has corresponding logging. On indexing finishing, there are "BuildIndexer (search) - Finished re-indexing builds" and "BuildIndexer (metadata) - Finished re-indexing builds" lines in the log.

#### Integration with issue trackers

Since TeamCity 9.0, the issue trackers are configured on the project level instead of the global server-wide configuration. On the server upgrade, all existing issue tracker integrations are moved to the Root project, which makes them still accessible to all the projects on the server.

#### WebSocket connections and proxy servers

Since 9.0, TeamCity tries to establish WebSocket connections between the browser and the server for the UI updates. If you have a proxy server (like nginx) in front of the TeamCity web UI, make sure that the proxy is [configured](#) properly to support WebSocket connections.

If the proxy is misconfigured or does not support the WebSocket protocol, a server health item will be shown for TeamCity system administrators. In this case TeamCity will use plain old polling for the UI updates as before.

#### REST API

REST API uses version 9.0. Previous versions of API are still available under `/app/rest/8.1`, `/app/rest/7.0`, `/app/rest/6.0` URLs.

- Change bean: the `webLink` attribute is renamed to `webUrl` to match other beans ([TW-34398](#)).
- Sub-elements representing empty collections in some of the beans are no longer included into responses (used to be included as an empty tag in XML).
- the `builds` `changes` element does not include the "count" attribute by default (for performance reasons), count can still be included by providing `fields=$long,changes(count,href)`
- The `/app/rest/agents` request now returns all the authorized agents by default (used to include unauthorized connected agents as well)
- queued builds now have the `id` attribute instead of the `taskId` attribute (they are the same for new builds since TeamCity 9.0)

#### Build tags-related changes

The `/app/rest/builds/<buildLocator>/tags` build request now returns a different XML: `<tags count="1"><tag name="TAG"/></tags>` instead of `<tags><tag>TAG</tag></tags>`.

The same applies to the `/app/rest/buildTypes/<buildTypeLocator>/<buildTypeLocator>/buildTags` request.  
The same change in the structure also applies to the build's entity nested "tags" element.

To create a tag, there is an old way to post a plain-text tag name to the `app/rest/builds/<buildLocator>/tags` URL. When sending POST or PUT XML or JSON requests to the URL, the new XML format is to be used (`<tag name="TAG"/></tags>` instead of `<tag>TAG</tag>`).

#### Handling tests with the same name within a build

In TeamCity 9.0, multiple tests with the same name within the same build are considered a single test with an invocation count. If any of these test runs fail, the whole test is considered failed in the build. The related issue is [TW-24212](#).

This change results in drop of test number counters in builds which have multiple runs for the same test. If you have a build failure condition which relies on test number in the build, this change may affect you.

If you need the tests to be treated as separate ones, consider running them in the test suites with different names or otherwise changing the test/running logic to change the full test name displayed in TeamCity.

#### Database-related changes

The national character sets (nchar, nvarchar, nclob types) for text fields are now supported in MS SQL databases used by TeamCity. It is recommended to use the Microsoft native JDBC driver, as jTDS JDBC driver does not support the nchar and nvarchar characters. If you still use jTDS, please [migrate](#).

Upon upgrade and entering the normal working status, TeamCity starts a background process to move the entries from the vcs\_changes database table to vcs\_change table. This process is transparent and you can continue working with the server as usual. It has negligible impact on the server performance and the only affected logic is the projects import feature (it is recommended to be used only with backups taken after the process is completed). The progress of the process can be seen on the Backup section in the server administration, along with "TeamCity is currently optimizing VCS-related data in the database for better backup/restore performance" message.

The other important thing is that the data copying increases the size of the raw database storage.

If this is an issue for your case (e.g. it might be with Microsoft SQL Server database with set database size limit), it is recommended to ensure the database size limit is twice the current size before the upgrade. It is possible to perform database-specific procedures to shrink the storage to match the actually stored data after the VCS changes migration process finishes.

#### VCS Root-related changes

The Git and Mercurial VCS roots no longer provide the ability to specify a custom clone path on the server for new VCS roots. If you need this ability, set the following internal properties to true for git and mercurial respectively: `teamcity.git.showCustomClonePath`, `teamcity.hg.showCustomClonePath`.

#### Visual Studio Addin

The TeamCity Add-in installed as a part of [ReSharper Ultimate](#) will remove the pre-bundle products versions: TeamCity and ReSharper versions prior to 9.0, dotCover prior to 3.0, dotTrace prior to 6.0.

Besides, it will not use the settings provided by the 8.1 version. The traditional add-in downloaded from TeamCity server can still use settings from previous version.

#### Other

TeamCity agent installation via the Java web start installation package is no longer available.

### Changes from 8.1.1 to 8.1.4

No noteworthy changes.

### Changes from 8.1 to 8.1.1

#### Command Line Runner

The change in behavior introduced in 8.1 (see [below](#)) has been fixed. Command line runners using "Executable with parameters" option which were created/changed with TeamCity 8.1 can expose a change in behavior with the upgrade. The recommended approach is to switch to "Custom script" option instead of "Executable with parameters" in command line runner.

#### Separate download for VSTest.Console runner

VSTest console runner is no longer bundled with TeamCity and is available as a separate plugin. For download details, see the [plugin page](#)

### Changes from 8.0.6 to 8.1

#### Known issue with creating MS SQL database with integrated security

When installing TeamCity anew and creating an MS SQL database with integrated security using the database setup UI, you

may receive an error. We are planning to resolve it in the next bugfix, meanwhile use the following workaround:

1. After receiving the error, stop the TeamCity server.
2. Start the TeamCity server.
3. On the database configuration screen, fill out the required information. Do not use the Refresh button. Make sure the information specified is correct.
4. Continue with the configuration steps.

If for some reason the workaround above does not resolve the problem, do the following:

1. Start the TeamCity server, approve a new database creation, configure the MS SQL access with a login and password, without [integrated security](#).
2. Ensure that TeamCity works properly.
3. Stop the TeamCity server.
4. Modify the `database.properties` file: configure MS SQL connection string to use integrated security, and remove the login and password.
5. Start the TeamCity server again.

#### Known issue with VSTest.Console runner

A new "VSTest.Console" runner which first appeared in TeamCity 8.1 is in experimental state and is not recommended for production use at this time. It will not be present in TeamCity 8.1.x by default (will be available as a separate download).

#### Known issue with PowerShell runner

PowerShell runner plugin is broken in 8.1. Fix is available, please follow instructions in [issue comment](#).

#### Known issue with Command Line Runner

Command line runner using "Executable with parameters" option can process quotes ("") and percentage signs (%) in a bit different way then in previous TeamCity versions (see details in the [issue](#)). To switch back to the previous (8.0) behavior you may specify `command.line.run.as.script=false` configuration parameter in a build configuration or in a project. The issue is fixed in 8.1.1.

The recommended approach is to switch to "Custom script" option instead of "Executable with parameters" in command line runner.

#### Memory Settings

If you have not [switched to 64 bit JVM](#) yet and use `-Xmx1300` memory setting for the server and the server is running on Windows, consider decreasing the setting to `-Xmx1200` as otherwise you might encounter "Native memory allocation (malloc) failed" JVM crash. See the [recommended memory settings](#) for details.

#### Actions menu

Some actions has moved under the "Actions" button available at the top-right of the page, near Run button.

These include:

"Label this build sources" on Changes tab of a build,  
"Pause", "Copy", "Move", "Delete", "Associate with Template", "Extract Template", "Extract Meta-Runner" on build configuration settings administration page,  
"Copy", "Move", "Delete", "Archive", "Bulk edit IDs" on project settings administration page.

#### Create Maven build configuration is not available by default

Action "Create Maven build configuration" is no longer available. Most of its functionality is covered by create project from URL and create VCS root from URL pages.

#### triggeredBy parameter from GroovyPlug plugin

The `build.triggeredBy` and `build.triggeredBy.username` configuration parameters provided by the [plugin](#) added by the plugin are now [available](#) without the plugin under `teamcity.build.triggeredBy` and `teamcity.build.triggeredBy.username` names respectively. Consider migrating to the latter set of parameters if you used the plugin's ones.

#### Shared Resources build feature

If the build takes lock on all values of a resource with custom values, these values are provided as lock values in build parameters. Corresponding issue: [TW-29779](#)

#### TeamCity Disk Space Watcher

The following [internal properties](#) define free disk space thresholds on the TeamCity server machine:

- `teamcity.diskSpaceWatcher.threshold` set to 500 Mb by default displays a warning on all the pages of the TeamCity Web UI.
- `teamcity.pauseBuildQueue.diskSpace.threshold` set to 50 Mb by default pauses the build queue.

The `teamcity.diskSpaceWatcher.softThreshold` property is removed.

#### PowerShell

The PowerShell plugin now uses the version that was specified in the UI as the `-version` command line argument when executing scripts. Corresponding issue: [TW-33472](#)

#### REST API

The latest version of the API has not changed, it is still "8.0" while there are changes in the API detailed below. If you find this inconvenient for your REST API usages, please comment in the corresponding [issue](#).

Entities returned in the response of REST API requests might now exclude attributes/elements with empty/default values. This is relevant for boolean fields with "false" value and empty collections. The recommended approach is to make sure the client code assumes "false" as a value for not present boolean attributes/elements.

"`projectName`" of `buildType` node now contains full project name (with the names of the parent projects) instead of the short name of the project.

In the lists of builds, "`startDate`" attribute is not longer included in the "`build`" node. It has become an element instead of attribute to match the full build data representation. If your REST API usage is affected, check [a way](#) to get that element in a request for the list of builds.

Requests `/app/rest/buildTypes/XXX/parameters/YYY` and `/app/rest/projects/XXX/parameters/YYY` now support "text/plain" and "application/xml" responses. To get plain text response (which was the only supported way before 8.1) you will need to supply "Accept: text/plain" header to the request.

Password properties of the VCS roots are now included into the responses, just without values.

CCTray-format XML (`app/rest/cctray/projects.xml`) does not include paused build configurations now.

Response to the experimental request `/app/rest/buildTypes/XXX/investigations` has changed the format and got additional fields to cover tests and problem investigations. There is an internal property `rest.beans.buildTypeInvestigationCompatibility` to include removed sub-items. Please let us know via [support email](#) if you need to use the internal property.

#### Eclipse plugin

Dropped support of Subversion 1.4-1.6. Now only Subversion 1.7-1.8 working copies formats supported.

#### Changes from 8.0.5 to 8.0.6

No noteworthy changes.

#### Changes from 8.0.4 to 8.0.5

No noteworthy changes.

#### Changes from 8.0.3 to 8.0.4

##### First Cleanup

First Cleanup after server upgrade might take a bit more time then regularly if there are many builds on the server. Following cleanups will then run a bit faster then in previous versions.

#### Changes from 8.0 to 8.0.3

No noteworthy changes.

#### Changes from 7.1.x to 8.0

This version introduces user-assignable IDs for projects and build configurations. This new ID is now used instead of internal id (projectN and btNNN) in at least:

- URLs of the web pages and artifact downloads
- in REST API
- project IDs are also used in directory names on the server under <TeamCity Data Directory>\system\artifacts instead of project names used prior to TeamCity 8.0

If you used any of the above, please, verify if you are affected by the change.

Learn more about IDs at [Identifier](#).

On upgrade, all the projects get automatically generated IDs based on their names.

Build configuration IDs are set to be equal to internal (btNNN) ids and can be later changed from the Administration UI via the Regenerate ID or Bulk Edit IDs actions.

Please note that the names of the projects and build configurations are no longer unique server-wide (are only unique within the direct parent project) and can contain any symbols which might be relevant if you used these in directory or file names.

#### Project settings format on disk

The format of the project settings storage on the disk under <TeamCity Data Directory>\config has been changed.

If you used any tools to read or update `project-config.xml` files, you will need to update the tools. It is recommended to use REST API or TeamCity open API (Java) to make changes so that the tools are not hugely affected by the format change.

#### Build Configuration templates

In version 8.0 build configuration templates support project hierarchy and TeamCity uses new rules:

- The TeamCity administration UI limits the use of templates only to those from the current project and its parents. On copying a project or a build configuration, the templates which do not belong to the target project or one of its parents are automatically copied.
- TeamCity no longer allows attaching a build configuration to a template if the template does not belong to the current project or one of its parents.
- Before version 8.0 it was possible to extract templates from a build configuration of one project to an unrelated project or to associate a build configuration in one project with a template in another. After upgrade to TC 8.0, such templates will become inaccessible in the current project. To reuse build configuration templates from an unrelated project, it is recommended to manually move them into the common parent project (or the Root project if you want them to be globally available).

#### JVM-originated agent parameters (os.arch and others)

The agent no longer reports system properties which come from the agent JVM: `system.os.arch`, `system.os.name`, `system.os.version`, `system.user.home`, `system.user.name`, `system.user.timezone`, `system.user.language`, `system.user.country`, `system.user.variant`, `system.path.separator`, `system.file.encoding`, `system.file.separator`.

All the aforementioned parameters are now reported as configuration parameters with the `teamcity.agent.jvm.` prefix instead.

If you used any of the parameters, make sure you update them to the new values.

#### IntelliJ IDEA project runner

IntelliJ IDEA project runner now uses IntelliJ IDEA's external make tool to build projects. Since this tool requires Java 1.6 to work, IntelliJ IDEA project runner now requires Java 1.6 (at least) too.

#### Clean-up for build configurations with feature branches

Build configurations with feature branches now process clean-up rules per-branch which can result in more builds preserved during clean-up than in previous versions. See [details](#).

#### Team Foundation Server integration

TFS now prefers Team Explorer 2012 to Team Explorer 2010 (if both are installed) for TFS operations

#### Compatibility with YouTrack

If you use JetBrains YouTrack and use its TeamCity integration features, please note that only YouTrack version 4.2.4 and later are compatible with TeamCity 8.0.

If you need earlier YouTrack versions to work with TeamCity 8.0, please let us know.

## REST API

### External ids

There are changes in the API related to the new external ids for project/build types/templates as well as other changes. The old API compatible with TeamCity 7.1 is still provided under "/app/rest/7.0" URL.

If you used URLs with locators having "id" for projects, build configuration or templates (like .../app/rest/projects/id:XXX or .../app/rest/buildTypes/id:XXX), please, update the locators to one of the following:

- (recommended) "id:EXTERNAL\_ID" (you can get the external ID in web UI URLs or via a request to .../app/rest/projects/internalId:OLD\_ID/id)
- just "ANY\_ID" to find the entity either by its internal, external id or name (use with caution: you can find more than you expect)
- "internalId:INTERNAL\_ID" to find the entity by the internal id  
You can also use the "/app/rest/7.0/" URL prefix instead of "/app/rest/" to work with 7.0-version of REST API which still uses internal IDs except for finish build trigger properties.

Also, it is possible to set the internal property `rest.compatibility.allowExternalIdAsInternal=true` to turn on the compatibility mode so that id:xxx locators will search also by the internal id. Note that this will be dropped in the future versions of TeamCity and is not recommended for use.

### Other Changes

Requests for builds ".../builds/<locator>/..." and ".../builds?locator=<locator>" no longer return personal and canceled builds by default. To include those, make sure you add ",personal:any,canceled:any" to the locators.

The "relatedIssues" element of the build entity no longer contains a full list of related issues. It has only the "href" attribute whose value can be used to get the related issues via a separate request.

There is also an internal property "rest.beans.build.inlineRelatedIssues" which can be set to `true` to return the "relatedIssues" node back for compatibility. See [TW-20025](#) for details. Also, the ".../builds/xxx/related-issues" URL is renamed to ".../builds/xxx/relatedIssues".

The "source\_buildTypeId" property is dropped from snapshot and artifact dependency nodes. Instead, the "source-buildType" sub-element is added with a reference to the build type.

Creating dependencies is still supported with the "source\_buildTypeId" property, but is deprecated. There is an internal property "rest.compatibility.includeSourceBuildTypeInDependencyProperties" which can be set to `true` to include the "source\_buildTypeId" property back.

In version 8.0 VCS roots support project hierarchy:

- When creating a VCS root, the `project` element should always be provided now. The element supports the `locator` attribute to specify the project.
- the `shared` attribute is dropped from the VCS root: after upgrade, such VCS roots are attached to the root project (with the "`_Root`" ID) and become globally available.
- when copying projects and build configurations, the `shareVCSRoots` attribute is no longer present. To make the VCS root available to projects and build configurations, move it to the parent/root project and then proceed with the copying.

It is recommended to create projects hierarchy which corresponds to organizational/settings sharing structure and move the VCS roots to most nested umbrella projects. Users then can be granted "Create / delete VCS root" role in the project to be able to edit VCS roots. Please note that users can edit a VCS root only if it is used in the projects they have "Edit project" permission for.

The "template" attribute in a build configuration template node is renamed to "templateFlag".

PUT for /users/<locator>/roles and /userGroups/<locator>/roles now accepts list of roles as it should and replaces existing roles instead of accepting single riles and adding it.

Many of PUT and POST requests which used to return nothing now return the entities created.

### Open API changes

#### See [details](#)

#### Shared Resources plugin

If you used the [Shared Resources plugin](#) with TeamCity 7.1.x, make sure to remove it as it is now bundled. See the [upgrade instructions](#).

#### Queue Manager plugin

If you used the [QueueManager plugin](#), make sure to remove it as it is now bundled. See the [upgrade instructions](#)

#### Bundled Maven

Maven bundled with TeamCity upgraded to version 3.0.5.

#### HTTPS connections from agents to server

If your agents connect to the TeamCity server by HTTPS protocol, and after upgrade agents fail to connect with error messages like:

```
javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure
```

then you should change Tomcat SSL connector configuration, i.e. add the following attribute to SSL connector and restart TeamCity server:

```
sslEnabledProtocols="TLSv1,SSLv3,SSLv2Hello"
```

The issue only manifests when the server runs under Java 1.7.

See also:

- [http://mail-archive.apache.org/mod\\_mbox/tomcat-users/201302.mbox/%3C512559F7.4080001@gmail.com%3E](http://mail-archive.apache.org/mod_mbox/tomcat-users/201302.mbox/%3C512559F7.4080001@gmail.com%3E)
- <http://youtrack.jetbrains.com/issue/TW-30221#comment=27-561843>

#### Changes from 7.1.4 to 7.1.5

`teamcity.build.branch` parameter semantics has changed, see <http://youtrack.jetbrains.com/issue/TW-23699#comment=27-448002>

#### Changes from 7.1.3 to 7.1.4

No noteworthy changes.

#### Changes from 7.1.2 to 7.1.3

No noteworthy changes.

Please check up-to-date list of known regressions for the version in our issue tracker.

#### Changes from 7.1.1 to 7.1.2

##### Possible issues with hg server-side checkout

There is a known issue with 7.1.2 release: [TW-24405](#) which can reproduce when server-side checkout, labeling or file content viewing are used for Mercurial repository.

If you experience the error with message "abort: destination 'hg1' is not empty", please install the patch attached to the issue.

##### Other known issues

Please also check a list of known regressions for the version in our issue tracker.

#### Changes from 7.1 to 7.1.1

No noteworthy changes.

#### Changes from 7.0.x to 7.1

##### Windows service configuration

Since version 7.1, TeamCity uses its own service wrapping solution for the TeamCity server as opposed to that of default Tomcat one in previous versions.

This changes the way TeamCity service is configured (data directory and server startup options including memory settings) and makes it unified between service and console startup.

Please refer to the updated [section](#) on configuring the server startup properties.

Agent windows service started to use OS-provided environment variables. Once Agent server (and JVM) are x86 processes, agent will report x86 environment variables. The change may affect your CPU bitness checks. See [MSDN Blog](#) on how to check if machine supports x64 by reported environment variables

##### Default location for TeamCity Data Directory when installed with Windows installer

This is only relevant for fresh TeamCity installations with Windows installer. Existing settings are preserved if you upgrade an existing installation.

Windows installer now uses `%ALLUSERSPROFILE%\JetBrains\TeamCity` location as default one for [TeamCity Data Directory](#). In TeamCity 7.0 and previous versions that used to be `%USERPROFILE%\BuildServer`.

##### Windows domain login module

When TeamCity server runs under Windows and Windows domain user authentication is used, TeamCity now uses another

library (Waffle) to talk to the Windows domain.

Under Linux the behavior is unchanged: jcIFS library is used as it were.

Unless you specified specific settings for jcIFS library in ntlm-config.properties file, your installation should not be affected. If you experience any issues with login into TeamCity with your Windows username/password after upgrade, please provide details [to us](#). In the mean time you can switch to using old jcIFS library. For this, add `teamcity.ntlm.use.jcifs=true` line into [internal properties file](#).

Please note that jcIFS library approach can be deprecated in future versions of TeamCity, so the property specification is not recommended if you can go without it.

#### Checkout directory change for Git and Mercurial

Build configurations that have either Git or Mercurial VCS roots and use default checkout directory will perform clean checkout upon upgrade. The clean checkout will be triggered by changed default checkout directory name. Further builds will reuse the checkout directory more aggressively (all builds using different branches but using the same VCS root will use the same directory). This affects agent- and server-side checkouts.

#### Perforce agent checkout workspace names change

Build configurations using Perforce agent-side checkout will perform clean checkout once after server upgrade. This is related to changed names for automatically generated Perforce workspaces.

#### SVN revision format

For changes, detected in external repositories, SVN revision got format `NNN_MMM:EXTUUID_CHANGEDATE`, where `NNN` - revision of the main repository, `MMM` - revision of externals repository, `EXTUUID` - UUID of externals repository, `CHANGEDATE` - change timestamp. This change may affect plugins/REST api clients which use revision of the last build change somehow.

#### Eclipse IDE plugin compatibility

Since TeamCity 7.1, Eclipse version 3.3 (Europa) is no longer supported by TeamCity Eclipse plugin.

Eclipse 3.8 and Eclipse 4.2 (Juno) are now supported.

#### Default schema when Microsoft SQL Server is used as an external database

Starting with version 7.1 TeamCity works only with a single database schema unlike previous versions when it could work with tables in any schemas of the database server.

TeamCity-related tables should now be located in the database schema which is set as default one for the database user used by TeamCity to connect to the database.

This change may require reconfiguration of the database to set default schema for the user used by TeamCity server to connect to the database.

Please check that all TeamCity-related tables are located in the default user's schema before performing the upgrade. (e.g. [using](#) the 'sys.tables' view)

If the default user's schema is not set right, TeamCity can report "TeamCity database is empty or doesn't exist. If you proceed, a new database will be created." message on the first start of newer TeamCity.

To change user's default schema, use the 'alter user' SQL command.

For the default schema description, see the "Default Schemas" section in the [corresponding documentation](#).

#### Open API changes

See [details](#)

### Changes from 7.0.1 to 7.0.4

No noteworthy changes.

### Changes from 7.0 to 7.0.1

#### HTML report tabs URLs Change

If you use direct links for build-level or project-level [report tabs](#), please update the links as they will [change](#) after upgrade. The change is necessary to make the feature more reliable.

### Changes from 6.5.x to 7.0

(Known issue) Build can hang or produce memory error for NUnit and other .Net test runners

Affected are: .Net test runners (NUnit, MSTest, MSpec) as well as TeamCity NUnit console launcher.  
Reproduces when path to test assemblies has several deep paths without wildcards ("\*").

Visible outcome: build hangs or fails with OutOfMemoryException error after "Starting  
...JetBrains.BuildServer.NUnitLauncher.exe" link in the build log.

The issue ([TW-20482](#)) is fixed and the fix will be included in the next release.  
Patch with a fix is [available](#).

## Minimum Supported Project JDK for Ant Runner

Starting with this version Ant runner requires minimum of JDK 1.4 in runtime build part (was 1.3 previously). This means that you will not be able to use TeamCity Ant runner if your project uses JDK 1.3 for compilation or tests running. For projects that require JDK 1.3 you can use command-line runner instead and configure "XML report processing" build feature to parse test reports.

## Supported Java for Server and Agent

Starting with this version the following requirements

- TeamCity server should be run with JRE 1.6 or above (was 1.5 previously). TeamCity .exe distribution is already bundled with appropriate Java. For .tar.gz or .war TeamCity distributions you might need to install and configure server manually.
- TeamCity agent should be run with JRE 1.6 or above (was 1.5 previously). Agent .exe distribution is already bundled with appropriate Java. If you used .zip agent distribution or installed the TeamCity agent with TeamCity version 5.0 or earlier, you might need [manual steps](#). If you run TeamCity 6.5.x, please check "Agents" page of your existing TeamCity server: the page will have a yellow warning in case any of the connected agents are running JDK less than 1.6.



### "Important!"

If any of your agents are running under JDK version less than 1.6, the agents will fail to upgrade and will stop running on the server upgrade. You will need to recover them manually by installing JDK 1.6 and making sure the agents will use it.

## Project/Template parameters override

In TeamCity 7.0 project parameters have higher priority than parameters defined in template, i.e. if there is a parameter with some name and value in the project and there is parameter with the same name and different value in template of the same project, value from the project will be used. This was not so in TeamCity 6.5 and was [changed](#) to be more flexible when template belongs to another project.

Build configuration parameters have the highest priority, as usual.

## Support for Sybase is discontinued

From this version support for Sybase as external database is shifted back into "experimental" state.

The reason for this decision is that it does not seem like the database is actively used with TeamCity, and supporting it requires a significant effort from TeamCity team which otherwise can be directed to improving more important areas of the product. While it should be still [possible](#), we do not recommend using Sybase as an external database and we are not planning to provide support for the Sybase-related issues.

Please consider using one of the other [databases supported](#). If you use Sybase, please migrate to another database before upgrading TeamCity.

## REST API Changes

- Several objects got additional attributes and sub-elements (e.g. BuildType, VcsRoot). Please check that your parsing code still works.  
/buildTypes/ path: BuildType object dropped runParameters field (as well as /<locator>/runParameters path is dropped) in favor of steps collection and /<locator>/steps/ path.
- A [bug](#) fixed which resulted in non-array JSON representation of single element arrays for some resources. Please check if your code is affected.
- in build object, "dependency-build" element is renamed to "snapshot-dependencies", revisions/revision/vcs-root is renamed to revisions/revision/vcs-root-instance (and it points to resolved VCS root instance now), revisions/revision/display-version is renamed to "version".
- in buildType object, "vcs-root" element is renamed to "vcs-root-entries"

Old version of the REST API is available via `/app/rest/6.0/...` URL in TeamCity 7.0. Please update your REST-using code as future versions of TeamCity might drop support for 6.0 protocol.

## Minimum version of supported Tomcat

If you use TeamCity .war distribution, please note that Tomcat 5.5 is no longer supported. Please update Tomcat to version 6.0.27 or above (Tomcat 7 is recommended).

## Swabra

`swabra.handle.exe.path` and `handle.exe.path` configuration parameters are no longer supported for providing path to Sysinternals handle.exe on agents. See the [plugin page](#) for the details.

## Open API Changes

Classes from `jetbrains.buildServer.messages.serviceMessages` package like `jetbrains.buildServer.messages.serviceMessages.BuildStatus` no longer depend on `jetbrains.buildServer.messages.Status` class. To make your code compatible with TeamCity 6.0 - 7.0 you can use `jetbrains.buildServer.messages.serviceMessages.ServiceMessage#asString` methods, for example:

```
ServiceMessage.asString("buildStatus", new HashMap<String, String>() {{  
    put("text", "Errors found");  
    put("status", "FAILURE");  
}});
```

See also Open API Changes

Changes from 6.5.4 to 6.5.6

No noteworthy changes

Changes from 6.5.4 to 6.5.5

(Known issue infex in 6.5.6) .NET Duplicates finder may stop working, the patch is available, please see this comment: <http://youtrack.jetbrains.net/issue/TW-18784#comment=27-261174>

Changes from 6.5.3 to 6.5.4

No noteworthy changes

Changes from 6.5.3 to 6.5.4

No noteworthy changes

Changes from 6.5.2 to 6.5.3

No noteworthy changes

Changes from 6.5.1 to 6.5.2

Maven runner

Working with MAVEN\_OPTS has changed again. Hopefully for the last time within the 6.5.x iteration. (see <http://youtrack.jetbrains.net/issue/TW-17393>)

Now TeamCity acts as follows:

1. If MAVEN\_OPTS is set TeamCity takes JVM arguments from MAVEN\_OPTS
2. If "JVM command line parameters" are provided in the runner settings, they are taken instead of MAVEN\_OPTS and MAVEN\_OPTS is overwritten with this value to propagate it to nested Maven executions.

Those who after upgrading to 6.5 had problems of not using MAVEN\_OPTS and who had to copy its value to the "JVM command line parameters" to make their builds work, now don't need to change anything in their configuration. Builds will work the same way they do in 6.5 or 6.5.1.

Changes from 6.5 to 6.5.1

(Fixed known issue) Long upgrade time and slow cleanup under Oracle

Changes from 6.0.x to 6.5

(Known issue) Long upgrade time and slow cleanup under Oracle

On first upgraded server start the database structures are converted and this can take a long time (hours on a large database) if you use Oracle external database ([TW-17094](http://youtrack.jetbrains.net/issue/TW-17094)). This is already fixed in 6.5.1.

Agent JVM upgrade

With this version of TeamCity we added semi-automatic upgrade of JVM used by the agents. If there is a Java 1.6 installed on the agent, and the agent itself is still running under the Java 1.5, TeamCity will ask to switch agent to Java 1.6. All you need is to review that detected path to Java is correct and confirm this switch, the rest should be done automatically. The operation is per-agent, you'll have to make it for each agent separately. Note that we recommend to switch to Java 1.6, as at some point TeamCity will not be compatible with Java 1.5. Make sure newly selected java process has same firewall rules (i.e. port 9090 is opened to accept connections from server)

IntelliJ IDEA Coverage data

Coverage data produced by IntelliJ IDEA coverage engine bundled with TeamCity 6.5 can only be loaded in IntelliJ IDEA 10.5+.

Due to coverage data format changes older versions of IntelliJ IDEA won't be able to load coverage from the server.

IntelliJ IDEA 8 is not supported

Plugin for IntelliJ IDEA no longer supports IntelliJ IDEA 8.

Unsupported MySQL versions

Due to [bugs](#) in MySQL 5.1.x TeamCity no longer supports MySQL versions in range 5.1 - 5.1.48. TeamCity won't start with appropriate message if unsupported MySQL version is detected. Please upgrade your MySQL server to version 5.1.49 or later.

Finish build properties are displayed

Finished builds now display all their properties used in the build on "Parameters" tab. This can potentially expose names and values of parameters from other builds (those that the given build uses as artifact or snapshot dependency). Please make sure this is acceptable in your environment. You can also manage users who see the tab with "View build runtime parameters and data" permissions which is assigned "Project Developers" role by default.

PowerShell runner is bundled

If you installed [PowerShell plugin](#) manually, please remove it from `.BuildServer/plugins` as a fresh version is now bundled with TeamCity.

Changed settings location

- XML test reporting settings are moved from runner settings into a dedicated [build feature](#).
- "Last finished build" artifact dependency on a build which has snapshot dependency is automatically converted into dedicated "Build from the same chain" source build setting.

Responsibility is renamed to Investigation

A responsibility assigned for a failing build configuration or a test is now called investigation. This is just a terminology change to make the action more neutral.

If you have any email processing rules for TeamCity investigation assignment activity, please check if they need updating to use new text patterns.

REST API Changes

Several objects got additional attributes and sub-elements (e.g. "startDate" in reference to a build, "personal" in a change). Please check that your parsing code still works.

Cleaning Non-default Checkout Directories

In previous releases, if you have specified build [checkout directory](#) explicitly using absolute path, TeamCity would not clean the content of the directory to free space on the disk.

This is no longer the case.

So if you have absolute path specified for the checkout directory and you need the directory to be present on agent for other build or for the machine environment, please set `system.teamcity.build.checkoutDir.expireHours` property to "never" and re-run the build. Please take into account that using [custom checkout directory](#) is not recommended.

If you are using one of `system.teamcity.build.checkoutDir.expireHours` properties and it is set to "never" to prevent the checkout directory from automatic deletion, the directory might be deleted once after TeamCity upgrade. Running the build in the build configuration once after the upgrade (and within 8 days from the previous build) will ensure that the directory preserves the "protected" behavior and will not be automatically removed by TeamCity.

Free disk space

This release exposes [Free disk space feature](#) in UI that was earlier only available via setting build configuration properties. While the old properties still work and take precedence, it is highly recommended to remove them and specify the value via "Disk Space" build feature instead. Future TeamCity versions might stop to consider the properties specified manually.

Command line runner

`@echo off` which turns off command-echoing is added to scripts provided by "Custom script" runner parameter. To enable command-echoing add `@echo on` to the script.

Windows Tray Notifier

You will need to upgrade windows tray notifier by uninstalling it and installing it again. Unfortunately, auto-upgrade will not work due to issues in old version of Windows Tray Notifier.

Maven runner

- In earlier TeamCity versions Maven was executed by invoking the 'mvn' shell script. You could specify some parameters in `MAVEN_OPTS` and some in UI. Maven build runner created its own `MAVEN_OPT` by concatenating these two (`%MAVEN_OPTS%+jvmArgs`). In this case, if some parameter was specified twice - in `MAVEN_OPTS` and in UI, only the one specified in `MAVEN_OPTS` was effective. Starting with TeamCity 6.5 Maven runner forms direct java command. While this approach solves many different problems, it also means that `MAVEN_OPTS` isn't effective anymore and all JVM command line parameters should be specified in build runner settings instead of `MAVEN_OPTS`.
- Those who had to manually setup surefire XML reporting for Maven release builds in TeamCity 6.0.x because otherwise tests weren't reported, now can forget about that. Since TeamCity 6.5 surefire tests run by `release:prepare` or `release:perform` goals are automatically detected. So don't forget to switch surefire XML reporting off in the build configuration settings to avoid double-reporting!

Email sending settings

Please check email sending settings are working correctly after upgrade (via Test connection on Administration > Server Configuration > EMail Notifier). If no authentication is needed, make sure login and password fields are blank. Non-blank fields may cause email sending errors if SMTP server is not expecting authentication requests.

#### XML Report Processing

Tests from Ant JUnit XML reports can be reported twice (see [TW-19058](#)), as we no longer automatically ignore TESTS-xxx.xml report.

To workaround this avoid using \*.xml mask and specify more concrete rules like TEST-\*.xml or alike that will not match report with name starting with "TESTS-"

#### Open API Changes

Several return types have changes in TeamCity open API, so plugins might need recompilation against new TeamCity version to continue working.

Also, some API was deprecated and will be discontinued in later releases. It is recommended to update plugins not to use deprecated API.

See also [Open API Changes](#)

### Changes from 6.0.2 to 6.0.3

No noteworthy changes

### Changes from 6.0.1 to 6.0.2

#### Maven and XML Test Reporting Load CPU on Agent

If you use Maven or XML test reporter and your build is CPU-intensive, you might find important the [known issue](#). Patch is available, fixed in the following updates.

### Changes from 6.0 to 6.0.1

No noteworthy changes

### Changes from 5.1.x to 6.0

#### Visual Studio Add-in and Perforce

There is critical bug in TeamCity 6.0 VS Add-in when Perforce is enabled. This can cause Visual Studio hangs and crashes. The fixed add-in version is [available](#). (related [issue](#)). The issue is fixed in TeamCity 6.0.1.

#### TFS checkout on agent

TFS checkout on agent might refuse to work with errors. Patch is available, see the [comment](#). Related [issue](#). The issue is fixed in TeamCity 6.0.1.

#### Error Changing Priority class

You may encounter a browser error while changing priority number of a priority class. A patch is available in a related [issue](#). The issue is fixed in TeamCity 6.0.1.

#### IntelliJ IDEA Compatibility

IntelliJ IDEA 6 and 7 are no longer supported in TeamCity plugin for IntelliJ IDEA.

Also, if you plan to upgrade to IntelliJ IDEA X (or other JetBrains IDE) please review this [known issue](#).

#### Build Failure Notifications

TeamCity 6.0 differentiates between build failure occurred while running a build script and one occurred while preparing for the build. The errors occurring in the latter case are called "failed to start" errors and are hidden by default from web UI (see "Show canceled and failed to start builds" option on Build Configuration page).

Since TeamCity 6.0, there is a separate notification rule "The build fails to start" which applies for "failed to start" builds. All the rest build failure notifications relate to build script-related failures.

Please note that on upgrade, all users who had "The build fails" notification on, will automatically get "The build fails to start" option to preserve old behavior.

#### Properties Changes

`teamcity.build.workingDir` property should no longer be used in non-runner settings. For backward compatibility, the property is supported in non-runner settings and is resolved to the working directory of the first defined build step.

#### Swabra and Build Queue Priorities Plugins are Bundled

If you have installed the plugins previously, please remove them (typically from `.BuildServer/plugins`) before starting upgraded TeamCity version.

Maven runner

Java older than 1.5 is no longer supported by the agent part of Maven runner. Please make sure you specify 1.6+ JVM in Maven runner settings or ensure JAVA\_HOME points to such JVM.

#### NUnit and MSTest Tests

If you had NUnit or MSTest tests configured in TeamCity UI (sln and MSBuild runners), the settings are extracted from the runners and converted to a new runner of corresponding type.

Please note that implementation of tests launching has changed and this affected relative paths usage: in TeamCity 6.0 the working directory and all the UI-specified wildcards are resolved based on the build's [checkout directory](#), while they used to be based on the directory containing .sln file. Simple settings are converted on TeamCity upgrade, but you might need to verify the runners contain appropriate settings.

#### "%" Escaping in the Build Configuration Properties

Now, two percentage signs (%%) in values defined in Build Configuration settings are treated as escape for a single percentage sign. Your existing settings are converted on upgrade to preserve functioning like in previous versions. However, you might need to review the settings for unexpected "%" sign-related issues.

#### .Net Framework Properties are Reported as Configuration Parameters

In previous TeamCity versions, installed .Net Frameworks, Visual Studios and Mono were reported as System Properties of the build agents.

This made the properties available in the build script.

In order to reduce number of TeamCity-specific properties pushed into the build scripts, the values are now reported via Configuration Parameters (that is, without "system." prefix) and are not available in the build script by default. They still be used in the Build Configuration settings via %-references by their previous names, just without "system." prefix.

#### Ipr runner is deprecated in favor of IntelliJ IDEA Project runner

Runner for IntelliJ IDEA projects was completely rewritten. It is not named "IntelliJ IDEA Project" runner. Previously available Ipr runner is also preserved but is marked as deprecated and will be removed in one of the further major releases of TeamCity. It is highly recommended to migrate your existing build configurations to the new runner.

Please note that the new runner uses different approach to run tests: you need to have a shared Run Configuration created in IntelliJ IDEA and reference it in the runner settings.

#### Cleanup for Inspection and Duplicates data

Starting from 6.0 Inspection and Duplicates reports for the builds are cleaned when build is cleaned from history, not when build's artifacts are cleaned as it used to be.

#### Inspection and Duplicates runners require Java 1.6

"Inspections" and "Duplicates (Java)" runners now require Java JDK 1.6. Please ensure Java 1.6 is installed on relevant agents and check it is specified in the "JDK home path" setting of the runners.

#### XML Report Validation

If you had invalid settings of "XML Report Processing" section of the build runners, you might find the Build Configurations reporting "Report paths must be specified" messages upon upgrade. In this case, please go to the runner settings and correct the configuration. (related [issue](#))

#### Open API Changes

See [Open API Changes](#)

Several jars in `devPackage` were reordered, some moved under `runtime` subdirectory. Please update your plugin projects to accommodate for these changes.

#### REST API Changes

Several objects got additional attributes and sub-elements. Please check that your parsing code still works.

#### Perforce Clean Checkout

All builds using Perforce checkout will do a clean checkout after server upgrade. Please note that this can impose a high load on the server in the first hours after upgrade and server can be unresponsive while many builds are in "transferring sources" stage.

## Changes from 5.1.2 to 5.1.3

#### Path to executable in Command line runner

The [bug](#) was fully fixed. The behavior is the same as in pre-5.1 builds.

## Changes from 5.1.1 to 5.1.2

Jabber notification sending errors are displayed in web UI for administrators again (these messages were disabled in 5.1.1). If you do not use Jabber notifications, please pause the Jabber notifier on the Jabber settings server settings page.

## Changes from 5.1 to 5.1.1

#### Path to executable in Command line runner

The [bug](#) was partly fixed. The behavior is the same as in pre-5.1 builds except for the case when you have the working directory specified and have the script in both checkout and working directory. The script from the working directory is used.

Path to script file in Solution runner and MSBuild runner

The [bug](#) was fixed. The behavior is the same as in pre-5.1 builds.

## Changes from 5.0.3 to 5.1

 If you plan to upgrade from version 3.1.x to 5.1, you will need to modify some dtd files in <TeamCity Data Directory>/config before upgrade, read more in the issue: [TW-11813](#)

 NCover 3 support may not work. See [TW-11680](#)

### Notification templates change

Since 5.1, TeamCity uses [new template engine](#) (Freemarker) to generate notification messages. New default templates are supplied and customizations to the templates made prior to upgrading are no longer effective.

If you customized notification templates prior to this upgrade, please review the new notification templates and make changes to them if necessary. Old notification templates are copied into <TeamCity Data Directory>/config/\_trash/\_notifications directory. Hope, you will enjoy the new templates and new extended customization capabilities.

### External database drivers location

JDBC drivers can now be placed into <TeamCity Data Directory>/lib/jdbc directory instead of WEB-INF/lib. It is recommended to use the new location. See details at [Setting up an External Database#Database Driver Installation](#).

PostgreSQL jdbc driver is no more bundled with TeamCity installation package, you will need to [install](#) it yourself upon upgrade.

### Database connection properties

Database connection properties template files have changed their names and are placed into database.<database-type>.properties.dist files under <TeamCity Data Directory>/config directory. They follow [.dist files convention](#).

It is recommended to review your database.properties file by comparing it with the new template file for your database and remove any options that you did not customize specifically.

### Default memory options change

We changed the default [memory option](#) for PermGen memory space and if you had -Xmx JVM option changed to about 1.3G and are running on 32 bit JVM, the server may fail to start with a message like: Error occurred during initialization of VM Could not reserve enough space for object heap Could not create the Java virtual machine.

On this occasion, please consider either:

- switching to 64 bit JVM. Please consider the [note](#).
- reducing PermGen memory via -XX:MaxPermSize [JVM option](#) (to previous default 120m)
- reducing heap memory via -Xmx [JVM option](#)

### Vault Plugin is bundled

In this version we bundled [SourceGear Vault VCS plugin](#) (with experimental status). Please make sure to uninstall the plugin from .BuildServer/plugins (just delete plugin's zip) if you installed it previously.

### Path to executable in Command line runner

A [bug](#) was introduced that requires changing the path to executable if working directory is specified in the runner. The bug is partly fixed in 5.1.1 and fully fixed in 5.1.3.

### Path to script file in Solution runner and MSBuild runner

A [bug](#) was introduced that requires changing the path to script if working directory is specified in the runner. The bug is fixed in 5.1.1.

### Open API Changes

See [Open API Changes](#)

## Changes from 5.0.2 to 5.0.3

No noteworthy changes.

 There is a known issue with .NET duplicates finder: [TW-11320](#)  
Please use the patch attached to the issue.

## Changes from 5.0.1 to 5.0.2

### External change viewers

The `relativePath` variable is now replaced with relative path of a file without checkout rules. The previous value can be accessed via `relativeAgentPath`. More information at [TW-10801](#).

## Changes from 5.0 to 5.0.1

No noteworthy changes.

## Changes from 4.5.6 to 5.0

### Pre-5.0 Enterprise Server Licenses and Agent Licenses need upgrade

With the version 5.0, we announce changes to the upgrade policy: Upgrade to 5.0 is not free. Every license (server and agent) bought since 5.0 will work with any TeamCity version released within one year since the license purchase. Please review the detailed information at [Licensing and Upgrade](#) section of the official site.

### Bundled plugins

If you used standalone plugins that are now bundled in 5.0, do not forget to remove the plugins from `.BuildServer/plugins` directory.

The newly bundled plugins are:

- Mercurial
- Git (JetBrains)
- REST API (was provided with YouTrack previously)

### Other plugins

If you use any plugins that are not bundled with TeamCity, please make sure you are using the latest version and it is compatible with the 5.0 release. e.g. You will need the latest version of [Groovy plug](#) and other properties-providing extensions. Pre-5.0 notifier plugins may lack support for per-test and assignment responsibility notifications.

### Obsolete Properties

The system property "build.number.format" and environment variable "BUILD\_NUMBER\_FORMAT" are removed. If you need to use build number format in your build (let us know why), you can define build number format as `%system.<property name>%` and define `<property name>` system property in the build configuration (it will be passed to the build then).

### Oracle database

If you use TeamCity with Oracle database, you should add an addition privilege to the TeamCity Oracle user. In order to do it, log in to Oracle as user SYS and perform

```
grant execute on dbms_lock to <TeamCity_User>;
```

### PostgreSQL database

TeamCity 5.0 supports PostgreSQL version 8.3+.

So if the version of your PostgreSQL server is less than 8.3 then it needs to be upgraded.

### Open API Changes

See [Open API Changes](#)

## Changes from 4.5.2 to 4.5.6

No noteworthy changes.

## Changes from 4.5.1 to 4.5.2

Here is a critical issue with Rake runner in 4.5.2 release. Please see [TW-8485](#) for details and a fixing patch.

## Changes from 4.5.0 to 4.5.1

No noteworthy changes.

## Changes from 4.0.2 to 4.5

### Default User Roles

The roles assigned as default for new users will be moved to "All Users" groups and will be effectively granted to all users already registered in TeamCity.

## Running builds during server restart

Please ensure there are no running builds during server upgrade.

If there are builds that run during server restart and these builds have test, the builds will be canceled and re-added to build queue ([TW-7476](#)).

## LDAP settings rename

If you had LDAP integration configured, several settings will be automatically converted on first start of the new server. The renamed settings are:

- `formatDN` — is renamed to `teamcity.auth.formatDN`
- `loginFilter` — is renamed to `teamcity.auth.loginFilter`

## Changes from 4.0.1 to 4.0.2

### Increased first cleanup time

The first server cleanup after the update can take significantly more time. Further cleanups should return to usual times.

During this first cleanup the data associated with deleted build configuration is cleaned. It was not cleaned earlier because of a bug in TeamCity versions 4.0 and 4.0.1.

## Changes from 4.0 to 4.0.1

### "importData" service message arguments

`id` argument renamed to `type` and `file` to `path`. This change is backward-compatible. See [Using Service Messages](#) section for examples of new syntax.

## Changes from 3.1.2 to 4.0

### Initial startup time

On the very first start of the new version of TeamCity, the database structure will be upgraded. This process can increase the time of the server startup. The first startup can take up to 20 minutes more than regular one. This time depends on the size of your builds history, average number of tests in a build and the server hardware.

### Users re-login will be forced after upgrade

Upon upgrade, all users will be automatically logged off and will need to re-login in their browsers to TeamCity web UI. After the first login since upgrade, Remember me functionality will work as usual.

### Previous IntelliJ IDEA versions support

IntelliJ IDEA plugin in this release is no longer compatible with IntelliJ IDEA 6.x versions. Supported IDEA versions are 7.0.3 and 8.0.

### Using VCS revisions in the build

`build.vcs.number.N` system properties are replaced with `build.vcs.number.<escaped vcs root name>` properties (or just `build.vcs.number` if there is only one root). If you used the properties in the build script you should update the usages manually or switch compatibility mode on. References to the properties in the build configuration settings are updated automatically. Corresponding environment variable has been affected too.

[Read more.](#)

### Test suite

Due to the fact that TeamCity started to handle tests suites, the tests with suite name defined will be treated as new tests (thus, test history can start from scratch for these tests.)

### Artifact dependency pattern

Artifact dependencies patterns now support [Ant-like wildcards](#).

If you relied on "" pattern to match directory names, please adjust your pattern to use "/" instead of single "\*".

If you relied on the "" pattern to download only the files without extension, please update your pattern to use "." for that.

### Downloading of artifacts with help of Ivy

If you downloaded artifacts from the build scripts (like Ant build.xml) with help of Ivy tasks you should modify your ivyconf.xml file and remove all statuses from there except "integration". You can take the ivyconf.xml file from the following page as reference: <http://www.jetbrains.net/confluence/display/TCD4/Configuring+Dependencies>

### Browser caches (IE)

To force Internet Explorer to use updated icons (i.e. for the Run button) you may need to force page reload (Ctrl+Shift+R) or delete "Temporary Internet Files".

## Changes from 3.1.1 to 3.1.2

No noteworthy changes.

## Changes from 3.1 to 3.1.1

No noteworthy changes.

## Changes from 3.0.1 to 3.1

### Guest User and Agent Details

Starting from version 3.1, the Guest user does not have access to the agent details page. This has been done to reduce exposing potentially sensitive information regarding the agents' environment. In the Enterprise Edition, the Guest user's roles can be edited at the Users and Groups page to provide the needed level of permission.

### StarTeam Support

#### Working Folders in Use

Since version 3.1 when checking out files from a StarTeam repository TeamCity builds directory structure on the base of the working folder names, not just folder names as it was in earlier versions. So if you are satisfied with the way TeamCity worked with StarTeam folders in version 3.0, ensure the working folders' names are equal to the corresponding folder names (which is so by default).

Also note, that although StarTeam allows using absolute paths as working folders, TeamCity supports relative paths only and doesn't detect absolute paths presence. So be careful and review your configuration.

### StarTeam URL Parser Fixed

In version 3.0 a user must have followed a wrong URL scheme. It was like `starteam://server:49201/project/view/rootFolder/su bfolder/...` and didn't work when user tried to refer a non-default view. In version 3.1 the native StarTeam URL parser is utilized. This means you now don't have to specify the root folder in the URL, and the previous example should look like `startea m://server:49201/project/view/subfolder/...`

## Changes from 3.0 to 3.0.1

### Linux Agent Upgrade

- Due to an issue with Agent upgrade under Linux, Agent auxiliary Launcher processes may have been left running after agent upgrades. Versions 3.0.1 and up fix the issue. To get rid of the stale running processes, after automatic agent upgrade, please stop the agent (via `agent.sh kill` command) and kill any running `java jetbrains.buildServer.agent.Launcher` processes and start the agent again.

## Changes from 2.x to 3.0

### Incompatible changes

Please note that TeamCity 3.0 introduces several changes incompatible with TeamCity 2.x:

- `build.working.dir` system property is renamed to `teamcity.build.checkoutDir`. If you use the property in your build scripts, please update the scripts.
- `runAll.bat` script now accepts a required parameter: `start` to start server and agent, `stop` to stop server and agent.
- Under Windows, `agent.bat` script now accepts a required parameter: `start` to start agent, `stop` to stop agent. Note that in this case agent will be stopped only after it becomes idle (no builds are run). To force immediate agent stopping, use `agent.bat stop force` command that is available under both Windows and Linux (`agent.sh stop force`). Under Linux you can also use `agent.sh stop kill` command to stop agents not responding to `agent.sh stop force`.

### Build working directory

Since TeamCity 3.0 introduces ability to configure VCS roots on per-Build Configuration basis, rather than per-Project, the default directory in which build configuration sources are checked out on agent now has generated name. If you need to know the directory used by a build configuration, you can refer to `<agent home>/work/directory.map` file which lists build configurations with the directory used by them. See also [Build Checkout Directory](#)

### User Roles when upgrading from TeamCity 1.x/2.x/3.x Professional to 3.x Enterprise

When upgrading from TeamCity 1.x/2.x/3.x Professional to 3.x Enterprise for the first time TeamCity's accounts will be assigned the following [roles](#) by default:

- Administrators become System Administrators
- Users become Project Developers for all of the projects
- The Guest account is able to view all of the projects
- Default user roles are set to Project Developer for all of the projects

## Changes from 1.x to 2.0

## Database Settings Move

Move your database settings from the <TeamCity installation folder>/ROOT/WEB-INF/buildServerSpring.xml file to the database.properties file located in the TeamCity configuration data directory (<TeamCity Data Directory>/config).

See also:

[Administrator's Guide: Licensing Policy](#)

.NET inspection and duplicates

inherited

 If you were using the TeamCity-GitHub [third-party plugin](#) prior to TeamCity 10.0, you can safely [remove](#) it: the built-in TeamCity integration will detect the existing connection to GitHub issue tracker and pick up your settings automatically.

## teamcity.TruncateIgnoreReasonConverter.copyReasons

```
Error
  collecting changes
for
  VCS repository
...
Failed
  to collect TFS changes
-
From
  version x
is
  greater
then
  current version y
```

The [bug with temp tool folders|<https://youtrack.jetbrains.com/issue/TW-46648>] introduced in the previous version has been fixed.

## Upgrade

 Unless specifically noted, TeamCity does not support downgrade between major/minor releases (changes in first two numbers of the version). It is strongly recommended to [back up your data](#) before any upgrade.

TeamCity supports upgrades from any of the previous versions to the later ones. All the settings and data are preserved unless noted in the [Upgrade Notes](#).

It is recommended to plan for regular upgrades to run the latest TeamCity version at least after several bugfix updates are released. This way you run a fully [supported version](#) with the latest fixes and security patches.

On this page:

- Before Upgrade
  - Licensing
- Upgrading TeamCity Server
  - Automatic Update

- [Manual Upgrade](#)
  - [Upgrading Using Windows Installer](#)
  - [Manual Upgrading using .tar.gz or .war Distributions](#)
  - [Upgrading TeamCity started from Docker images](#)
  - [Upgrading TeamCity started from AWS CloudFormation template](#)
- [IDE Plugins](#)
- [Upgrading Build Agents](#)
  - [Automatic Build Agent Upgrading](#)
  - [Upgrading Build Agents Manually](#)
    - [Upgrading the Build Agent Windows Service Wrapper](#)
      - [Upgrading from TeamCity version 1.x](#)
      - [Upgrading from TeamCity version 2.x](#)

## Before Upgrade

Before upgrading TeamCity:

1. For a major upgrade, review what you will be getting in [What's New](#) (follow the links at the bottom of What's New if you are upgrading not from the previous major release)
2. [Check your license keys](#) unless you are upgrading within bugfix releases of the same major X.X version
3. Download the new TeamCity version ([extended download options](#))
4. Carefully review the [Upgrade Notes](#)
5. Consider probing the upgrade on a [test server](#)
6. If you have non-bundled plugins installed, check plugin pages for compatibility with the new version and upgrade/uninstall the plugins if necessary

To upgrade the server:

1. [Back up the current TeamCity data](#)
2. Perform the upgrade steps:
  - [Upgrading Using Windows Installer](#)
  - [Manual Upgrading on Linux and for WAR Distributions](#)

If you plan to upgrade a production TeamCity installation, it is recommended to install a [test server](#) and check its functioning in your environment before upgrading the main one.

## Licensing

Before upgrading, please make sure the maintenance period of your licenses is not yet elapsed (use Administration | Licenses TeamCity server web UI page to list your license keys). The licenses are valid only for the versions of TeamCity with the effective release date within the maintenance period. Check the effective release date on the [releases list](#).

Typically all the bugfix updates (indicated by changes in the Z part of the X.Y.Z TeamCity version) use the same effective release date (that of the major/minor release).

If not all the licenses cover the target version release date, consider [renewing the licenses](#) before the upgrade (you can replace the old license keys with the renewed ones even before the upgrade).

If you are only evaluating a newer version, you can get an evaluation license on the [download page](#). Please note that each TeamCity version can be evaluated only once. To extend the evaluation period, [contact the JetBrains sales department](#).

When upgrading from TeamCity 4.x or earlier, note that the licensing policy in TeamCity versions 5.0 and later are different from that of the previous TeamCity versions. Review the [Licensing Policy](#) page and the [Licensing and Upgrade](#) section on the official site.

## Upgrading TeamCity Server

TeamCity supports upgrades from any of the previous versions to the current one.

Unless specifically noted, downgrades with preserving the data are not possible with changing major.minor version and are possible within bugfix releases (without changing major.minor version).

The general policy is that bugfix updates (indicated by changes in the Z part of the X.Y.Z TeamCity version) do not change data format, so you can freely upgrade/downgrade within the bugfix versions. However, when upgrading to the next major or minor version (changed X or Y in X.Y.Z TeamCity version), you will not be able to downgrade with the data preservation: you will need to [restore a backup](#) of the appropriate version.

On upgrade, all the TeamCity configuration settings and other data are preserved unless noted in [Upgrade Notes](#). If you have customized TeamCity installation (like Tomcat server settings change), you will need to repeat the customization after the upgrade.

The general approach to upgrade is to remove all the files of the previous installation in the TeamCity server home and place the new files into the same location. Make sure to preserve the [TeamCity Data Directory](#) and the database intact (making a backup beforehand), backing up and restoring previously customized settings (e.g. in ...\\conf\\server.xml, ...\\conf\\web.xml

files) is also necessary. The logs directory (...\\logs) can be left with the old installation files.

Agents connected to the server are upgraded [automatically](#).



#### Important note on TeamCity data structure upgrade

TeamCity server stores its data in the database and in [TeamCity Data Directory](#) on the file system. Different TeamCity versions use different data structure of the database and data directory. Upon starting newer version of TeamCity, the data is kept in the old format until you confirm the upgrade and data conversion on the Maintenance page in the web UI. Until you do so, you can back up the old data; however, once the upgrade is complete, the data is updated to a newer format.

Once the data is converted, downgrade to the previous TeamCity versions which uses different data format is not possible!

There are several important issues with data format upgrade:

- Data structure downgrade is not possible. Once newer TeamCity version changes the data format of database and data directory, you cannot use this data to run an older TeamCity version. Please ensure you [backup](#) the data before upgrading TeamCity.
- Both the database and the data directory should be upgraded simultaneously. Ensure that during the first start of the newer server it uses the correct [TeamCity Data Directory](#) that in its turn has the correct database [configured](#) in the <TeamCity Data Directory>\\config\\database.properties file. Also make sure the data directory is complete (e.g. all the build logs, artifacts, etc. are in place), no data directory content supports copying from the data directory of the older server versions.

If you accidentally performed an inconsistent upgrade, check the [recovery instructions](#).

## Automatic Update

When a new version of TeamCity is detected, the server displays the corresponding health item for system administrators. The item points to the server's Administration | Updates page, where all the versions available for the update are listed. The page contains notes about licenses compatibility, the new version description and controls to perform the automatic upgrade if you want to use that instead of performing the manual updating procedure.

The automatic update procedure is as follows:

1. The TeamCity server is stopped.
2. The update script is run to do the following:
  - a. Create a backup of the current installation in the [TeamCity home/.old](#) directory.
  - b. Update the stopped server to the new version.
3. Next, the updated server starts.  
The update progress is logged to the [TeamCity home/logs/teamcity-update.log](#) file.



In case of an automatic update failure, perform the following to restore your TeamCity to the state prior to the update:

1. Stop your TeamCity server if it is running
2. In your [TeamCity home](#) directory, remove the folders with the same name as the ones in the <TeamCity server home>/.old directory
3. Copy the contents of the <TeamCity server home>/.old directory to the <TeamCity server home> directory
4. Start the TeamCity server

Current automatic update limitations:

- some customizations, e.g. installations with [changed server context](#), are not supported by automatic update
- only manual upgrade is possible if the server is deployed from a [.war distribution](#), or runs under the official [TeamCity Docker container](#), started with [AWS CloudFormation template](#) or Azure Resource Manager template.
- the Windows uninstaller is not updated during the upgrade, so after several updates, old TeamCity version will still be noted in Windows lists. During the uninstallation, not all of the TeamCity installation files might be deleted.
- the bundled Java is not updated
- with several nodes installation, only the main TeamCity server can be auto-updated, the Running Builds node needs to be updated manually.

## Manual Upgrade

### Upgrading Using Windows Installer

1. [Create a backup](#). When upgrading from TeamCity 6.0+ you will also have a chance to create a backup with the "basic" profile on the [TeamCity Maintenance Mode](#) page on the updated TeamCity start.

2. Note the username used to run the TeamCity server. You will need it during the new version installation.
3. If you have any of the Windows service settings customized, store them to repeat the customizations later.
4. Note if you are using 64 bit Java to run the service (e.g. check for "64" in "Java VM info" on the server's Administration | Diagnostics or in a thread dump), consider backing up <TeamCity home>\jre directory.
5. (optional as these will not be overwritten by the upgrade) If you have any customizations of the bundled Tomcat server (like port, https protocol, etc.), JRE, etc. Backup those to repeat the customizations later.
6. Note if you have local agent installed (though, it is **not recommended** to have a local agent) so that you can select the same option in the installer.
7. Run the new installer and point it to the same place TeamCity is installed into (the location used for installation is remembered automatically). Confirm uninstalling the previous installation. The TeamCity uninstaller ensures proper uninstallation, but you might want to make sure the [TeamCity server installation directory](#) does not contain any non-customized files after uninstallation finishes. If there are any, backup/remove them before proceeding with the installation.

 The main server configuration file <TeamCity Home Directory>/conf/server.xml is updated automatically when there were no changes to it since the last installation. If modification were made, the installer will detect them and backup the old server.xml file displaying a warning about the overwrite and the backup file location. Other files under conf can be overwritten to their default content as well, so if you have made manual modifications in those, check them after the upgrade.

8. If prompted, specify the <TeamCity data directory> used by the previous installation.
9. (Optional as these will not be overwritten by the upgrade) Make sure you have the external database driver **installed** (this applies only if you use an external database).
10. Check and restore any customizations of Windows services and Tomcat configuration that you need. When upgrading from versions 7.1 and earlier, make sure to transfer the [server memory setting](#) to the [environment variables](#).
11. If you were using 64 bit Java to run the server restore the <TeamCity home>\jre directory previously backed up or repeat the 64 bit Java [installation steps](#).
12. If you use a customized Log4j configuration in the conf\teamcity-server-log4j.xml file and want to preserve it (note, however, that customizing the file is actually not recommended, use [logging presets](#) instead), compare and merge conf\teamcity-server-log4j.xml.backup created by the installer from the existing copy with the default file saved with the default name. Compare the conf\teamcity-\*\log4j.xml.dist file with the corresponding conf\teamcity-\*\log4j.xml file and make sure that .xml file contains all the .dist file defaults. It is recommended to copy the .dist file over to the corresponding .xml file until you really need the changed logging configuration.
13. Start up the TeamCity server (and agent, if it was installed together with the installer).
14. Review the [TeamCity Maintenance Mode](#) page to make sure there are no problems encountered, and confirm the upgrade by clicking the corresponding button. Only after that all data will be converted to the newer format.

If you encounter errors which cannot be resolved, make sure old TeamCity is not running/does not start on boot, restart the machine, and repeat the installation procedure.

## Manual Upgrading using .tar.gz or .war Distributions

Please note that it is recommended to use .tar.gz or .exe TeamCity distribution. Using .war is not a recommended way to install TeamCity.

1. [Create a backup](#).
2. Backup files customized since previous installation (most probably [TOMCAT\_HOME]/conf/server.xml)
3. Remove old installation files (the entire <TeamCity Home Directory> or [TOMCAT\_HOME]/webapps/TeamCity/\* if you are installing from a war file). It's advised to backup the directory beforehand.
4. Unpack the new archive to the location where TeamCity was previously installed.
5. If you use a Tomcat server (your own or bundled in .tar.gz TeamCity distribution), it is recommended to delete the content of the work directory. Please note that this may affect other web applications deployed into the same web server.
6. Restore customized settings backed up in step 2 above. If you have the customized [TOMCAT\_HOME]/conf/server.xml file, apply your changes into the appropriate sections of the default file.
7. Make sure the previously configured [TeamCity server startup properties](#) (if any) are still actual.
8. Start up the TeamCity server.
9. Review the [TeamCity Maintenance Mode](#) page to make sure there are no problems encountered, and confirm the upgrade by clicking the corresponding button. Only after that, all the configuration data and database scheme are updated by TeamCity converters.

## Upgrading TeamCity started from Docker images

If you made no changes to the container, you can just stop the running container, pull the new version of the [official TeamCity image](#) and the server in it via the usual command. If you changed the image, you will need to replicate the changes to the new TeamCity server image.

## Upgrading TeamCity started from AWS CloudFormation template

Please see the [dedicated page](#).

## IDE Plugins

It is recommended for all users to regularly update their IDE plugins to the latest version compatible with the TeamCity server version in use. At least to the version available from the TeamCity server's Tools section on user profile.

Generally, versions of the IntelliJ IDEA TeamCity plugin, Eclipse TeamCity plugin, and Visual Studio TeamCity Addin have to be the same as the TeamCity server version. Users with non-matching plugin versions get a message on an attempt to log in to the TeamCity server with a non-matching version.

The only exception is TeamCity versions 9.0 - 9.1.x, which use a compatible protocol, and any plugin of these versions can be used with any server of these versions. Updating IDE plugins to the matching server version is still recommended.

## Upgrading Build Agents

- [Automatic Build Agent Upgrading](#)
- [Upgrading Build Agents Manually](#)
  - [Upgrading the Build Agent Windows Service Wrapper](#)

### Automatic Build Agent Upgrading

On starting TeamCity server (and updating agent distribution or plugins on the server), TeamCity agents connected to the server and [correctly installed](#) are automatically updated to the version corresponding to the server. This occurs for both server upgrades and downgrades. If there is a running build on the agent, the build finishes. No new builds are started on the agent unless the agent is up to date with the server.

The agent update procedure is as follows: The agent (`agent.bat`, `agent.sh`, or agent service) will download the current agent package from the TeamCity server. When the download is complete and the agent is idle, it will start the upgrade process (the agent is stopped, the agent files are updated, and the agent is restarted). This process may take several minutes depending on the agent hardware and network bandwidth. Do not interrupt the upgrade process, as doing so may cause the upgrade to fail and you will need to manually reinstall the agent.

If you see that an agent is identified as "Agent disconnected (Will upgrade)" in the TeamCity web UI, do not close the agent console or restart the agent process, but wait for several minutes.

Various console windows can open and close during the agent upgrade. Please be patient and do not interrupt the process until the agent upgrade is finished.

### Upgrading Build Agents Manually

All connected agents upgrade automatically, provided they are [correctly installed](#), so manual upgrade is not necessary.

If you need to upgrade agent manually, you can follow the steps below:

As TeamCity agent does not hold any unique information, the easiest way to upgrade an agent if to

- back up the `<Agent Home>/conf/buildAgent.properties` file.
- uninstall/delete existing agent.
- install the new agent version.
- restore the previously saved `buildAgent.properties` file to the same location.
- start the agent.

If you need to preserve all the agent data (e.g. to eliminate clean checkouts after the upgrade), you can:

- stop the agent.
- delete all the directories in the agent installation present in the agent .zip distribution except `conf`.
- unpack the .zip distribution to the agent installation directory, skipping the "conf" directory.
- start the agent.

In the latter case, if you run the agent under Windows using a service, you may also need to upgrade the Windows service as described [below](#).

### Upgrading the Build Agent Windows Service Wrapper

#### Upgrading from TeamCity version 1.x

Version 2.0 of TeamCity migrated to the new way of managing Windows service (service wrapper) for the build agent: Java Service Wrapper library.

One of the advantages of using the new service wrapper is the ability to change any JVM parameters of the build agent process.

1.x versions installed Windows service under the name of agentd, while 2.x versions use the TeamCity Build Agent Service <build number> name.

The service wrapper will not be migrated to the new version automatically. You do not need to use the new service wrapper unless you need its functionality (like changing the agent's JVM parameters).

To use the new service wrapper, uninstall the old version of the agent (with Control Panel | Add/Remove Programs) and then install a new one.

If you customized the user under which the service is started, do not forget to customize it in the newly installed service.

#### Upgrading from TeamCity version 2.x

If the service wrapper needs an update, the new version is downloaded into the <agent>/launcher.latest folder, however the changes are not applied automatically.

To upgrade the service wrapper manually, do the following:

1. Ensure the <agent>/launcher.latest folder exists.
2. Stop the service using <agent>\bin\service.stop.bat.
3. Uninstall the service using service.uninstall.bat.
4. Backup the <agent>/launcher/conf/wrapper.conf file.
5. Delete <agent>/launcher.
6. Rename <agent>/launcher.latest to <agent>/launcher.
7. Edit the <agent>/launcher/conf/wrapper.conf file. Check that the 'wrapper.java.command' property points to the j ava.exe file. Leave it blank to use the registry to look up java. Leave 'java.exe' to look up java.exe in PATH. For a standalone agent, the service value should be ../jre/bin/java, for an agent installation on the server the value should be ../../jre/bin/java. The backup version of the wrapper.conf file can be used.
8. Install the service using <agent>\bin\service.install.bat.
9. Make sure the service is running under the proper user account. Please note that using SYSTEM can result in failing builds which use MSBuild/Sln2005 configurations.
10. Start the service using <agent>\bin\service.start.bat.



This procedure is applicable ONLY to an agent running with new service wrapper. Make sure you are not running the agentd service.

See also:

Concepts: [TeamCity Data Directory](#)

Administrator's Guide: [TeamCity Maintenance Mode](#) | [TeamCity Data Backup](#)

## TeamCity Maintenance Mode

If you see the TeamCity Maintenance page on the TeamCity startup, that means this TeamCity instance requires technical maintenance. For security reasons it is to be performed by a system administrator who has access to the computer where the TeamCity server is installed. In most cases this page appears if the data format doesn't correspond to the required format; for example, during [Upgrade](#) TeamCity will display this page before converting the data to a newer format.

If you do not have access to the computer where TeamCity is installed, inform your system administrator that TeamCity requires technical maintenance.

If you are a TeamCity system administrator, to confirm that enter the Super User Token into the corresponding field on this page. This token can be found in the teamcity-server.log file under <TeamCity home>/logs.

After you have provided this token, you can review the details on what kind of maintenance is required. The need in technical maintenance may be caused, for example, by one of the following:

- [TeamCity Data Upgrade](#)
- [TeamCity Data Format is Newer](#)
- [TeamCity Startup Error](#)
- [TeamCity Database Creation](#)

## TeamCity Data Upgrade

When upgrading your TeamCity instance, the newly installed version of TeamCity checks if the TeamCity data directory and database use the old data format when it is started for the first time. If the newer version requires data conversion, this page is displayed with data format details. Please, review them carefully.

 If you haven't backed up your data, do it at this point. Once TeamCity converts the data, downgrade won't be possible. If you need to return to an earlier TeamCity version, you will be able to do that only by restoring the data from a corresponding backup. Please refer to the [TeamCity Data Backup](#) section for instructions.

When you are sure you have backed up your data, click Upgrade.

 If you are upgrading from TeamCity 6.0 (or higher), this page contains an option to perform backup automatically.

## TeamCity Data Format is Newer

TeamCity has detected that the data format corresponds to more recent TeamCity version than you try to run. Since downgrade is not supported, TeamCity cannot start until you provide the data that matches the format of the TeamCity version you want to run. To do so, please restore the required data from backup. Refer to the [Restoring TeamCity Data from Backup](#) page for the instructions.

## TeamCity Startup Error

If on TeamCity startup a critical error was encountered, this page will display the error message. Please, fix the error cause and restart the TeamCity server.

## TeamCity Database Creation

If you see this screen when [settings up TeamCity with an external database](#) or [after migrating to an external database](#), click Proceed to create a new database for TeamCity.

See also:

[Installation and Upgrade: Upgrade](#)  
[Administrator's Guide: TeamCity Data Backup](#)

## Setting up an External Database

TeamCity stores build history, users, build results and some run time data in an SQL database. See also the description of what is stored where on the [Manual Backup and Restore](#) page.

If you evaluated TeamCity with the internal database which is [not recommended for production](#), please refer to [Migrating to an External Database](#).

The current database in use is shown on the "Administration | Global Settings" page in the "Database" field and also is mentioned in the teamcity-server.log on the server startup. "HSQL\*" means that internal database is in use.

On this page:

- [Default Internal Database](#)
- [Selecting External Database Engine](#)
  - [Supported Databases](#)
- [General Steps](#)
- [Database-specific Steps](#)
  - [MySQL](#)
    - [On MySQL server side](#)
    - [On TeamCity server side \(with MySQL\)](#)
  - [PostgreSQL](#)
    - [On PostgreSQL server side](#)
    - [On TeamCity server side \(with PostgreSQL\)](#)
  - [Oracle](#)
    - [On Oracle server side](#)
    - [On TeamCity server side \(with Oracle\)](#)
  - [Microsoft SQL Server](#)
    - [On MS SQL server side](#)
    - [On TeamCity server side \(with MS SQL\)](#)

- jTDS driver
- Database Configuration Properties

## Default Internal Database

On the first TeamCity run, using an internal database based on the HSQLDB database engine is suggested by default. The internal database suits evaluation purposes only; it works out of the box and requires no additional setup.

However, we strongly recommend using an external database as a back-end TeamCity database in a production environment. An external database is usually more reliable and provides better performance: the internal database may crash and lose all your data (e.g. on the "out of disk space" condition). Also, the internal database may become extremely slow on large data sets (say, database storage files over 200Mb). Please also note that our support does not cover any performance or database data loss issues if you are using the internal database.

In short, do not EVER use internal HSQLDB database for production TeamCity instances. [Migrate to an external database](#) the moment you start to rely on the data stored in TeamCity server.

## Selecting External Database Engine

As a general rule you should use the database that better suits your environment and that you can maintain/configure better in your organization.

While we strive to make sure TeamCity functions equally well under all of the supported databases, issues can surface in some of them under high TeamCity-generated load.

You may also want to estimate [the required database capacity](#).

## Supported Databases

TeamCity supports the following databases:

- MySQL
- PostgreSQL
- Oracle
- MS SQL

## General Steps

1. Configure the external database to be used by TeamCity (see the [database-specific sections](#) below).
2. Run the TeamCity server [for the first time](#).
3. Select an external database to be used and specify the database connection settings.  
If required, you can later manually modify your [database connection settings](#).  
 Note that TeamCity creates its own database schema on the first start and actively modifies it during the upgrade. The schema is not changed when TeamCity is working normally.  
The user account used by TeamCity should have permissions to create new, modify and delete existing tables in its schema, in addition to usual read/write permissions on all tables.
4. You may also need to download the JDBC driver for your database.  
Due to licensing terms, TeamCity does not bundle driver jars for external databases. You will need to download the Java JDBC driver and put the appropriate .jar files (see driver-specific sections below) from it into the `<TeamCity Data Directory>/lib/jdbc` directory.  
Please note that the .jar files should be compiled for the Java version not greater than the one used to run TeamCity, otherwise you might see "Unsupported major.minor version" errors related to the database driver classes.



See [additional information](#) if Amazon Aurora DB Cluster is used as the TeamCity database server.

## Database-specific Steps

The section below describes the required configuration on the database server and the TeamCity server.

### MySQL

#### Supported versions

##### On MySQL server side

Recommended database server settings:

- use InnoDB storage engine
- use [UTF-8 character set](#)
- use case-sensitive collation
- see also [recommendations for MySQL server settings](#)

The MySQL user account that will be used by TeamCity must be granted all permissions on the TeamCity database. This can be done by executing the following SQL commands from the MySQL console:

```
create database <database-name> collate utf8_bin;
create user <user-name> identified by '<password>';
grant all privileges on <database-name>.* to <user-name>;
grant process on *.* to <user-name>;
```

## On TeamCity server side (with MySQL)

JDBC driver installation:

1. Download the MySQL JDBC driver from <http://dev.mysql.com/downloads/connector/j/>. If the MySQL server version is 5.5 or newer, the JDBC driver version should be 5.1.23 or newer.
2. Place `mysql-connector-java-*.jar` from the downloaded archive into the [TeamCity Data Directory](#)/lib/jdbc. Proceed with the TeamCity setup.

## PostgreSQL

[Supported versions](#)

## On PostgreSQL server side

1. Create an empty database for TeamCity in PostgreSQL.
  - Make sure to set up the database to use UTF8.
  - Grant permissions to modify this database to the user account used by TeamCity to work with the database.
2. See also [recommendations for PostgreSQL server settings](#)

TeamCity does not specify which schema will be used for its tables. By default, PostgreSQL creates tables in the 'public' schema ('public' is the name of the schema). TeamCity can also work with other PostgreSQL schemas. To switch to a different schema, do the following:

Create a schema named exactly as the user name: this can be done using the pgAdmin tool or with the following SQL:

```
create schema teamcity authorization teamcity;
```

The schema has to be empty (it must not contain any tables).

## On TeamCity server side (with PostgreSQL)

Download the required [PostgreSQL JDBC42 driver](#) and place it into the [TeamCity Data Directory](#)/lib/jdbc. Proceed with the TeamCity setup.

## Oracle

[Supported versions](#)

## On Oracle server side

1. Create an Oracle user account/schema for TeamCity.

 TeamCity uses the primary character set (char, varchar, clob) for storing internal text data and the national character set (nchar2, nvarchar, nclob) to store the user input and data from external systems, like VCS, NTLM, etc.

- Make sure that the national character set of the database instance is UTF or Unicode.

- Grant the CREATE SESSION, CREATE TABLE, permissions to a user whose account will be used by TeamCity to work with this database. TeamCity, on the first connect, creates all necessary tables and indices in the user's schema. (Note: TeamCity never attempts to access other schemas even if they are accessible)



Make sure TeamCity user has quota for accessing table space.

## On TeamCity server side (with Oracle)

### 1. Get the Oracle JDBC driver.

Supported driver versions are 11.1 and higher.  
Place the following files:

- ojdbc8.jar (or ojdbc6.jar, ojdbc7.jar depending on your database version)
- orai18n.jar (can be omitted if missing in the driver version) into the `<TeamCity Data Directory>/lib/jdbc` directory.



The Oracle JDBC driver must be compatible with the Oracle server.

It is strongly recommended to locate the driver in your Oracle server installation. Contact your DBA for the files if required.

Alternatively, download the Oracle JDBC driver from the [Oracle web site](#). Make sure the driver version is compatible with your Oracle server.

### 2. Proceed with the TeamCity setup.

## Microsoft SQL Server

### Supported versions

For step-by-step instructions, see the [dedicated page](#). The current section provides key details required for the setup.

## On MS SQL server side



TeamCity uses the primary character set (char, varchar, text) for storing internal text data and the national character set (nchar, nvarchar, ntext) to store the user input and data from external systems, like VCS, NTLM, etc.

- Create a new database. As the primary collation, use the case-sensitive collation (collation name ending with '\_CS\_AS') corresponding to your locale.
- Create a TeamCity user and ensure that this user is the owner of the database (grant the user dbo rights), which will give the user the ability to modify the database schema.  
For SSL connections, ensure that the version of MS SQL server and the TeamCity version of java are compatible. We recommend using the latest update of SQL server.
- Allocate sufficient transaction log space depending on how intensively the server will be used. The recommended setup is not less than 1Gb.
- Make sure SQL Server Browser is running.
- Make sure TCP/IP protocol is enabled for SQL Server instance.
- Make sure that the "no count" setting is disabled: the enabled "no count" setting prevents TeamCity from starting queued builds.

## On TeamCity server side (with MS SQL)

- Download the [Microsoft JDBC driver v6.4+](#) (sqljdbc\_6.\*.x package, .exe or .tar.gz depending on your TeamCity server platform) from the Microsoft Download Center.
- Unpack the downloaded package into a temporary directory.
- Copy the mssql-jdbc-\* .jar file from the just downloaded package into the `TeamCity Data Directory/lib/jdbc` directory. MS SQL integrated security (Windows authentication) requires installing sqljdbc\_auth.dll from the driver package as per [instructions](#).
- Proceed with the TeamCity setup.

## jTDS driver

It is not recommended to use jTDS JDBC driver. There are at least known issues with using Unicode characters when the driver is in use.

If you use the driver ("jtds" text appears in the connectionUrl of database.properties), it is highly recommended to switch the native driver.

The due procedure for the switch is to:

- create the server [backup](#) including the database
- stop the server and configure the server to use native Microsoft JDBC driver as noted in the section above
- restore the database from the backup into a new MS SQL database
- run the server

## Database Configuration Properties

The database connection settings are stored in `<TeamCity Data Directory>\config\database.properties` file. The file is a Java [properties file](#). You can modify it to specify required properties for your database connections.

TeamCity uses Apache DBCP for database connection pooling. Please refer to <http://commons.apache.org/dbcp/configuration.html> for detailed description of configuration properties.

 For all supported databases there are [template files](#) with database-specific properties located in the `<TeamCity Data Directory>/config` directory. The files have the following naming format: `database.<database_type>.properties.dist` and can be used as a reference on the required settings.

See also:

[Installation and Upgrade: Common database-related problems](#) | [Migrating to an External Database Concepts](#): TeamCity Data Directory  
[Administrator's Guide: TeamCity Data Backup](#)

## Setting up TeamCity with MS SQL Server

- Prerequisites
- Configure MS SQL server to be used with TeamCity
  - Enable TCP/IP protocol for your SQL server
  - Create new database
  - Set up TeamCity database user
    - Create dedicated user for TeamCity with SQL server authentication
    - Create SQL database user with Windows authentication
  - Set up JDBC Driver for SQL Server database
    - Additional settings for Windows authentication (MS SQL integrated security)
- Start TeamCity
- Post-Setup notes

### Prerequisites

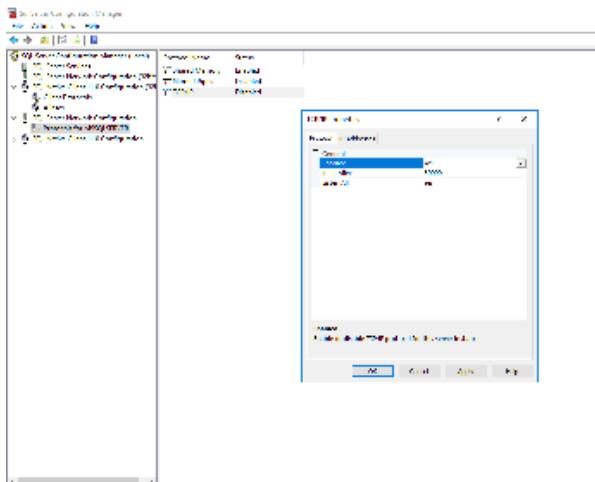
- MS SQL server
- MS SQL Server Management Studio
- [Installed TeamCity Server](#)

### Configure MS SQL server to be used with TeamCity

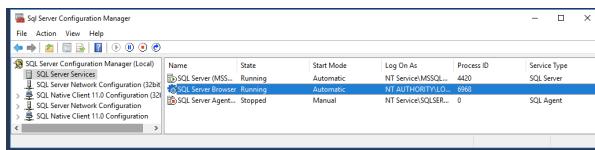
## Enable TCP/IP protocol for your SQL server

Open SQL Server Configuration Manager and do the following:

1. Expand SQL Server Network Configuration, click Protocols for MS SQL.
2. In the right pane, right-click TCP/IP, and then click Enable and select Yes. (MS SQL comes with TCP/IP disabled by default.) Click OK.



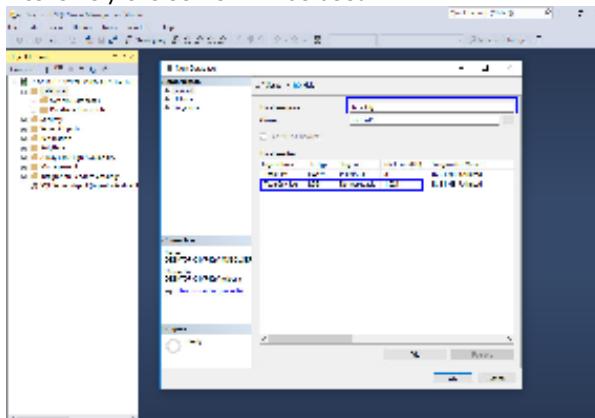
3. You need to restart your MSSQL server for the changes to take effect: in the left pane, click SQL Server Services, in the right pane, right-click SQL Server, and then click Restart.
4. Check that SQL Server Browser is running.



## Create new database

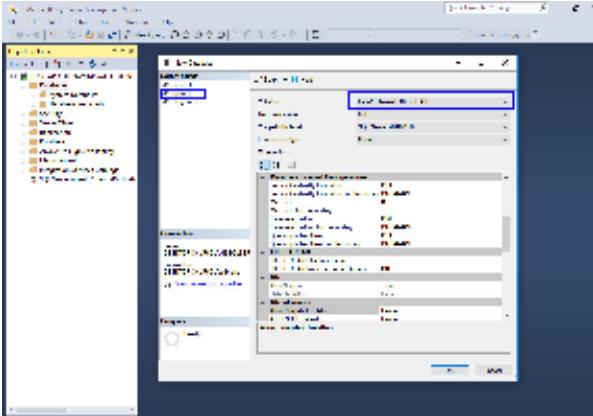
In MS SQL Server Management Studio:

1. Connect to your database server, right-click the Databases node in the Object Explorer and select New database.
2. On the General page, specify the database name, 'TeamCity' in the image below and allocate sufficient transaction log space. The recommended minimum is 1Gb (1024 Mb) in the image below. The requirements vary depending on how intensively the server will be used.

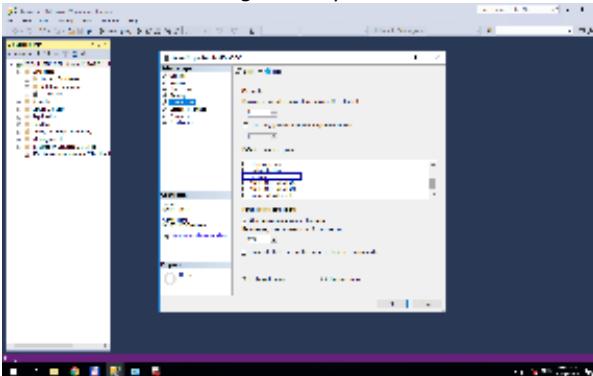


3. Next we need to specify primary collation: Go to the Option node in the left pane and select a collation on the right. We recommend a case-sensitive collation (with the collation name ending with '\_CS\_AS') corresponding to your locale and

click Ok to save the settings:



- Now make sure that the "no count" setting is disabled as follows: Right-click the server instance in Object Explorer, Choose Properties, Select the Connections tab. In the Default Connection Options frame, "no count" must be unchecked. Save changes if any.



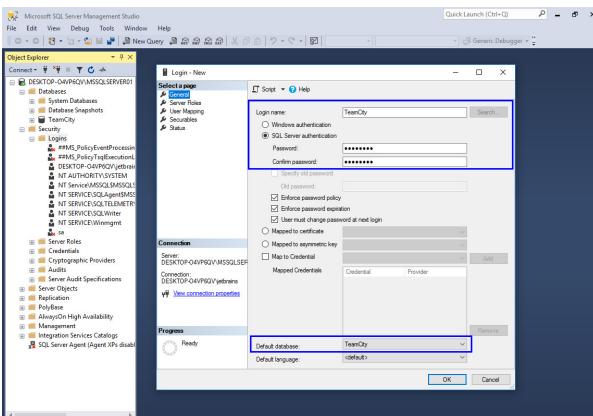
## Set up TeamCity database user

SQL Server supports two ways of authentication: SQL Server authentication and Windows authentication mode.

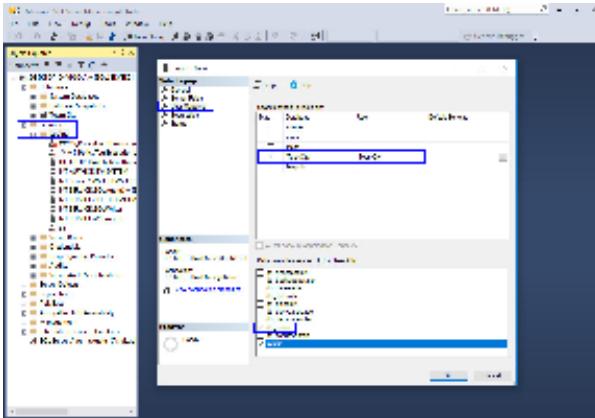
- SQL Authentication requires specifying username and a password in the database settings. It is recommended to start with this authentication before you try to use Windows authentication.
- Windows authentication (MS SQL integrated security) allows the TeamCity server running under a specific Windows user connect to the SQL server as that user, without providing a username and a password. However, it requires [additional setup](#)

### Create dedicated user for TeamCity with SQL server authentication

- Go to the Security node, right-click Logins, select New Login and in the General window that opens provide the login name, TeamCity in the image below, select the SQL server authentication and provide the password for the user. Set the default database to TeamCity.

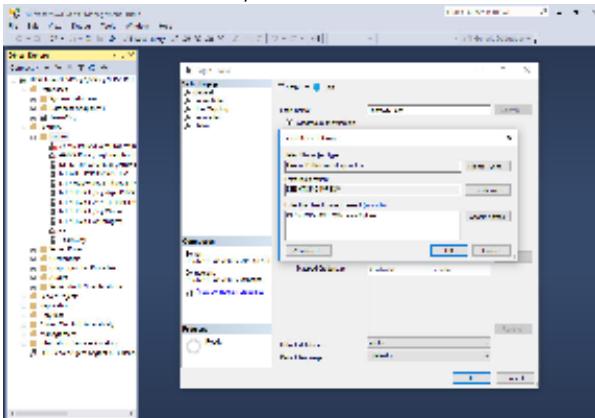


- Next select User Mapping in the left pane, in the upper right pane check the TeamCity database in the list and in the lower pane grant the user TeamCity database owner rights: check the db\_owner box. Click Ok.

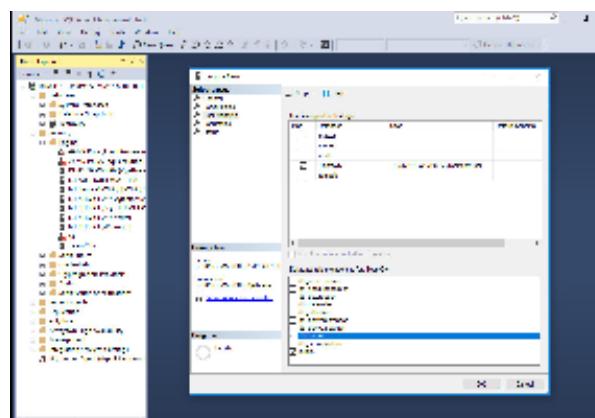


#### Create SQL database user with Windows authentication

1. Go to the Security node, right-click Logins, select New Login and in the General window that opens provide the login name, select Windows authentication and click the search button.
2. In the Select user or Group dialog, specify the user account which will be used to run TeamCity. Click Check names. Once the user is found, click OK.



3. Now grant this user db owner permissions: select User Mapping in the left pane, in the upper right pane check the TeamCity database in the list and in the lower pane grant the user TeamCity database owner rights: check the db\_owner box. Click Ok.



#### Set up JDBC Driver for SQL Server database

1. Download the [Microsoft JDBC driver v6.0+](#) (sqljdbc\_6.0.x package, .exe or .tar.gz depending on your TeamCity server platform) from the Microsoft Download Center.



Note that Microsoft JDBC driver v6.0+ has compatibility issues with Microsoft SQL Server 2005. For MS SQL Server 2005, use [JDBC driver v4.0](#) (exe or .tar.gz depending on your TeamCity server platform).

2. Unpack the downloaded package into a temporary directory.
3. Copy the sqljdbc42.jar from the just downloaded package into the TeamCity Data Directory/lib/jdbc directory.

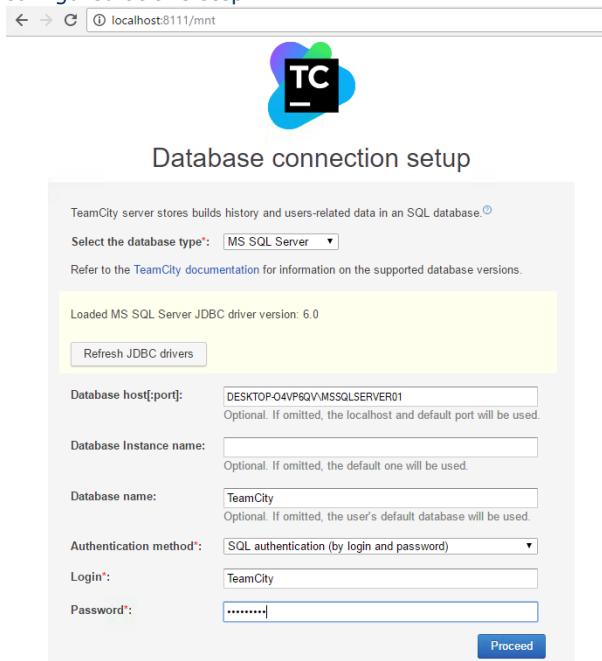
Additional settings for Windows authentication (MS SQL integrated security)

For Windows authentication (MS SQL integrated security), in addition to the JDBC driver, it is necessary to install native driver library `sqljdbc_auth.dll` from the JDBC driver package. The required version of the library depends on the bitness of the Java version used by the server.

For the default 32-bit JVM bundled with the TeamCity server, copy the `<sql_jdbc_home>/enu/auth/x86/sqljdbc_auth.dll` file into `TeamCity Data Directory\lib\jdbc\native\windows-i386`. For the **64-bit JVM** used to run the TeamCity server, use the `<sql_jdbc_home>/enu/auth/x64/sqljdbc_auth.dll` and place it into the `TeamCity Data Directory\lib\jdbc\native\windows-amd64` directory.

## Start TeamCity

1. Start the TeamCity server.  
For Windows authentication (MS SQL integrated security), make sure the server is running under the user configured at [this step](#).
2. Select MS SQL as the database.
3. Click the Refresh the JDBC drivers if asked.
4. Specify the connection settings:
  - Database host - your SQL server host
  - Port - optional,
  - Database instance name - leave blank for the default SQL server instance.  If a named instance is used, provide its name here.
  - Database name - the name of the newly created database
5. Select the required authentication type, for SQL Server authentication provide the credentials of the dedicated SQL user configured at [this step](#).



Database connection setup

TeamCity server stores builds history and users-related data in an SQL database. 

Select the database type\*: **MS SQL Server**

Refer to the [TeamCity documentation](#) for information on the supported database versions.

Loaded MS SQL Server JDBC driver version: 6.0

Database host[:port]:   
Optional. If omitted, the localhost and default port will be used.

Database Instance name:   
Optional. If omitted, the default one will be used.

Database name:   
Optional. If omitted, the user's default database will be used.

Authentication method\*:

Login\*:

Password\*:

6. Continue with the setup. It'll take some time to initialize the database schema and the components.

## Post-Setup notes

It is recommended to monitor the database performance regularly. For example, perform [reindexing](#) once every several months.

## Configuring UTF8 Character Set for MySQL

To create a MySQL database which uses the utf8 character set:

1. Create a new database:

```
create database <database_name> character set UTF8 collate utf8_bin
```

2. Open `<TeamCity data directory>/config/database.properties`, and add the `characterEncoding` property:

```
connectionProperties.characterEncoding=UTF-8
```

To change the character set of an existing MySQL database to utf8:

1. Shut the TeamCity server down.
2. Being in the TeamCity bin directory, export the database using the maintainDB tool:

```
maintainDB backup -D -F database_backup
```

(more details about backup are [here](#))

3. Create a new database with utf8 as the default character set, as described above.
4. Modify the `<TeamCity data directory>/config/database.properties` file --- change `connectionUrl` property to:

```
5. jdbc:mysql://<host>/<new_database_name>
```

6. Import data the new database:

```
maintainDB restore -D -F database_backup -T <TeamCity data  
directory>/config/database.properties
```

7. Start the TeamCity server up

## Using AWS Aurora Database Cluster

This page provides details on using an [Amazon Aurora](#) cluster as the TeamCity database server.

On this page:

- [Overview](#)
- [General Recommendations](#)
- [Forcing TeamCity to Connect to New Writer](#)

### Overview

When using an AWS Aurora cluster with TeamCity pointing to the `cluster` end-point as the database server, it is important to understand what happens when an AWS Aurora cluster fails over.

Both AWS Aurora DB instances are rebooted (so for a short period of time TeamCity entirely loses connection to the cluster) and

- the original DB instance is started with `innodb_read_only` flag set (the new reader instance);
- the former failover instance is the new writer and the cluster endpoint DNS record is changed to point to the new writer instance.

By default, TeamCity JVM caches DNS name lookups, which essentially means that TeamCity will stay connected to the original DB instance until DNS cache expires. This in turn leads to the database connection pool on the TeamCity side to be populated with the connections to the new reader.

It will take some time for the JVM-specific cache in TeamCity to expire and for the invalid connections to be evicted from the pool.

### General Recommendations

When working with a failover cluster, it is recommended you decrease the JVM-specific DNS caching on TeamCity by setting the `TTL` to 60:

1. Add the

```
-Dsun.net.inetaddr.ttl=60
```

JVM option to the `TEAMCITY_SERVER_OPTS` environment variable.

2. Restart TeamCity for the changes to take effect.



You may also wish to manually flush your OS-specific DNS cache, but this is the action to be taken each time Aurora fails over.

### Forcing TeamCity to Connect to New Writer

You can force TeamCity to connect to the new writer either manually or automatically.

To force the connection manually :

- reboot the new DB reader instance again: the reboot will take up to 2 minutes which is sufficient for the DNS cache to expire as well as for the invalid connections to be evicted from the pool
- alternatively, restart TeamCity manually and all the connections will be created anew.

To force TeamCity to automatically connect to the new primary instance as soon as it's up and running:



The following options may affect your TeamCity server performance.

Configure the database connection pool to use a special validation query, so that the connections to the DB instance are tested before and/or after use and if a connection to the read-only database is detected, they are evicted from the pool.

1. Add the following lines to the `<TeamCity Data Directory>/config/database.properties` file:

```
testOnBorrow=true  
testOnReturn=true  
testWhileIdle=true  
timeBetweenEvictionRunsMillis=60000  
validationQuery=select case when @@read_only + @@innodb_read_only \= 0 then 1 else  
(select table_name from information_schema.tables) end as `1`
```

2. Restart TeamCity.

Once you do that, the following SQL query:

```
select case when @@read_only + @@innodb_read_only = 0 then 1 else (select table_name  
from information_schema.tables) end as `1`
```

will be executed for all connections whenever they are borrowed from or returned to the pool, and also every 1 minute (60000 milliseconds) for idle connections, raising `error 1242 (ER_SUBSELECT_NO_1_ROW)` for each connection to the read-only database.

## Migrating to an External Database

For details on using an external database from the first TeamCity start, as well as the general external database information and the database-specific configuration steps, refer to the [Setting up an External Database](#) page.

The current section covers the steps required to migrate TeamCity data from one database to another. The most typical case is when you evaluated TeamCity with the default internal database and need to switch to an external database to prepare your TeamCity installation for production use. The steps here are also applicable when switching from one external database to another.



Database migration cannot be combined with the server upgrade. If you want to upgrade at the same time, you should first [upgrade](#), run the new version of TeamCity, and then migrate to another database.

There are several ways to migrate data into a new database:

- **Switch** with no data migration: build configurations settings will be preserved, but not the historical builds data or users.
- **Full Migration**: all the data is preserved except for any database-stored data provided by the third-party plugins.
- **Backup and then restore**: the same as full migration, but using the two-step approach.

## Switch with No Data Migration

If you want a fast switch to an external database and do not want to preserve existing data like users and builds on the server, follow the steps below. See [#Full Migration](#) for preserving all the data.

After the switch, the server will start with an empty database, but preserve all the settings stored under TeamCity Data Directory (see [details](#) on what is stored where).

Steps to perform the switch:

1. [Create and configure an external database](#) to be used by TeamCity.
2. Shut down the TeamCity server.
3. [Create a backup copy](#) of the `<TeamCity Data Directory>` used by the server.
4. Clean up the `system` folder: you must remove the `messages` and `artifacts` folders from the `/system` folder of your `<TeamCity data directory>`; you may delete the old HSQLDB files: `buildserver.*` to remove the no longer needed internal storage data.
5. Start the TeamCity server.



If you see the TeamCity Maintenance screen, click the "I'm a server administrator, show me the details" link and enter the [Super User Token](#). Follow the instructions to create a new TeamCity database.

## Full Migration

These steps describe switching to another database preserving all data. The TeamCity migration tool, the `maintainDB` command line utility, is available for this purpose.

The `maintainDB.[cmd|sh]` shell/batch script is located in the `<TeamCity Home>/bin` directory and is used for migrating as well as for [backing up and restoring](#) TeamCity data.

The utility is only available in the TeamCity `.tar.gz` and `.exe` distributions. If you installed TeamCity using the `.war` distribution and need to perform data migration, use the tool from the `.tar.gz` distribution.

TeamCity supports HSQLDB, MySQL, Oracle, PostgreSQL and Microsoft SQL Server; the migration is possible between any of these databases.



The target database must be empty before the migration process (it must NOT contain any tables).



If an error occurs during migration, do not use the new database as it may result in the database data corruption or various errors that can uncover later. In case of an error, investigate the reason logged into the console or in the migration logs (see below), and, if a solution is found, clean the target database and repeat the migration process.

To migrate all your existing data to a new external database:

1. [Create and configure an external database](#) to be used by TeamCity and install the database driver into TeamCity. Do not modify any TeamCity settings at this stage.
2. Shut down the TeamCity server.
3. Create a temporary properties file with a custom name (for example, `database.<database_type>.properties`) for the target database using the corresponding template (`<TeamCity Data Directory>/config/database.<database_type>.properties.dist`). Configure the properties and place the file into any temporary directory. Do not modify the original `database.<database_type>.properties` file.
4. Run the `maintainDB` tool with the `migrate` command and specify the absolute path to the newly created target database properties file with the `-T` option:

```
maintainDB.[cmd|sh] migrate [-A <path to TeamCity Data Directory>] -T <path to database.properties file>
```

**i** If you have the TEAMCITY\_DATA\_PATH environment set (pointing to the [TeamCity Data Directory](#)), you do not need the -A <path to TeamCity Data Directory> parameter of the tool.

Upon the successful completion of the database migration, the temporary file will be copied to (<TeamCity Data Directory>/config/database.properties) file which will be used by TeamCity. The temporary file can be safely deleted.

If you are migrating between external databases, the original database.properties file for the source database will be replaced with the file specified via the -T option. The original database.properties file will be automatically re-named to database.properties.before.<timestamp>.

5. Start the TeamCity server. This must be the same TeamCity version that was run last time (TeamCity [upgrade](#) must be performed as a separate procedure).

After you make sure the migration succeeded, you may delete the old HSQLDB files: buildserver.\* to remove the no longer needed internal storage data.

## Backup and Restore

You can [create a backup](#) and then [restore it](#) using different target database settings. You will probably need to specify restore options to restore only the database data.

## Troubleshooting

- Extended information during migration execution is logged into the logs\teamcity-maintenance.log file. Also, logs\t eamcity-maintenance-truncation.log contains extended information on possible data truncation during the migration process.
- If you encounter an "out of memory" error, try increasing the number in the -Xmx512m parameter in the maintainDB script. On a 32-bit platform, the maximum is about 1300 megabytes.  
Alternatively, run HSQLDB in the standalone mode via

```
java -Xmx256M -cp ..\webapps\ROOT\WEB-INF\lib\hsqldb.jar org.hsqldb.Server -database.0 <TeamCity data directory>\system\buildserver -dbname.0 buildserver
```

and then run the Migration tool pointing to the database as the source: jdbc:hsqldb:hsq://localhost/buildserver sa ''

- If you get "The input line is too long." error while running the tool (e.g. this may happen on Windows 2000), change the script to use an alternative classpath method.  
For maintainDB.bat, remove the lines below "Add all JARs from WEB-INF\lib to classpath" comment and uncomment the lines below "Alternative classpath: Add only necessary JARs" comment.

## See also:

[Installation and Upgrade: Common database-related problems](#)  
[Installation and Upgrade: Setting up an External Database](#)  
[Concepts: TeamCity Data Directory](#)  
[Administrator's Guide: TeamCity Data Backup](#)

## User's Guide

This guide is intended for anyone using TeamCity. Explore TeamCity and learn how you can

- [Modify your user account](#)
- [Subscribe to notifications](#)
- [View how your changes have affected the builds](#)
- [View problematic build configurations and tests](#)

- Review build results
- Investigate build problems
- View project and build configuration statistics
- Search TeamCity

## Managing your User Account

To manage your account settings, in the top right corner of the screen, click the arrow next to your username and select My Settings & Tools from the drop-down list.

In this section:

- [Changing Your Password](#)
- [Managing Version Control Username Settings](#)
- [Customizing UI](#)
- [Viewing your Roles and Permissions](#)

### Changing Your Password

On the General tab of the page, type your new password in the Password and Confirm password fields.

 The password can only be changed for the built-in authentication. If you don't see these fields, this means that TeamCity is configured to use external authentication and the password should be changed in the corresponding external system.

Since TeamCity 2017.1, it is possible to reset your built-in authentication password via the login page.

### Managing Version Control Username Settings

On the General tab of the My Settings & Tools page, you can see the list of your version control usernames in the Version Control Username Settings area.

By default, TeamCity uses your login name as the VCS username. Click Edit to provide actual usernames for version control systems you use. Make sure the user names are correct.

 These settings are not used for authentication for the particular VCS, etc.

These settings enable you to:

- track your changes status on the [Changes](#),
- highlight such builds on the [Projects](#) page if the appropriate [option is selected](#),
- notify you on such builds when the [Builds affected by my changes](#) option is selected in [notifications settings](#).

### Customizing UI

On the General tab of the My Settings & Tools page, you can customize the following UI settings:

- Highlight my changes and investigations: Select to highlight builds that include your changes (changes committed by a user with the VCS username provided in the [Version Control Username Settings](#) section) and problems you were assigned to investigate on the [Projects](#) page, Project Home Page, Build Configuration Home Page.
- Show date/time in my timezone: Check the option, if you want TeamCity to automatically detect your time zone and show the date and time (for example, build start, vcs change time, etc.) according to it.
- Show all personal builds
- Add builds manually triggered by you to your [favorites](#).

### Viewing your Roles and Permissions

1. In the top right corner of the screen, click the arrow next to your username, and select My Settings & Tools from the drop-down list.
  - To view the list of user groups you are in, go to the Groups tab.
  - To view your roles and permissions in different projects, go to the Roles tab. Note, that roles are assigned to the user by the system administrator.

See also:

Concepts: [Role and Permission](#)

User's Guide: [Viewing Your Changes](#) | [Subscribing to Notifications](#)

## Subscribing to Notifications

TeamCity provides a wide range of notification possibilities to keep developers informed about the status of their projects.

Notifications can be sent by e-mail, Jabber/XMPP instant messages or can be displayed in the IDE (with the help of TeamCity plugins) or the Windows system tray (using TeamCity Windows Tray Notifier). Notifications can also be received via Atom/RSS syndication feeds.

Notifications in TeamCity are sent per-user according to the user's configured notification rules. Notification rules can also be configured at the user group level, in which case they will apply to all the users in the group.

You can [customize](#) notification templates.

- Subscribing to Notifications
  - What Will Be Watched
  - Which Events Will Trigger Notifications
- Rules Processing Order
- Unsubscribing and Overriding Existing Rules
- Customizing RSS Feed Notifications
  - Feed URL Generator Options
  - Additional Supported URL Parameters
  - Example

## Subscribing to Notifications

TeamCity allows you to flexibly adjust the notification rules, so that you receive notifications only on the events you are interested in. To subscribe to notifications:

1. In the top right corner of the screen, click the arrow next to your username, and select My Settings & Tools from the drop-down list. Open the Notification Rules tab.
  2. Click the required notifications type:
    - Email Notifier: to be able to receive email notifications, your email address must be specified in the General area on the My Settings & Tools page.
-  Note that TeamCity comes with a default notification rule. It will send you an email notification if a build with your changes has failed. This rule starts working after you enter the email address.

  - IDE Notifier: to receive notifications right in your IDE, the required TeamCity plugin must be installed in your IDE. For the details on installing TeamCity IDE plugins, refer to [Installing Tools](#).
  - Jabber Notifier: to receive notifications of this type, specify your Jabber account either on the Notification Rules | Jabber notifier page, or the My Settings & Tools page in the Watched Builds and Notifications area. Note that instead of Jabber you can specify your Google Talk account here if this option is configured by the System Administrator.
  - Windows Tray Notifier: to receive this type of notifications, [Windows Tray Notifier](#) must be installed.
3. Click Add new rule and specify the rule in the dialog. The notification rules are comprised of two parts: [what you will watch](#) and [which events you will be notified about](#). See the details below.

 Email and Jabber notifications are sent only if the System Administrator has configured the TeamCity server email and Jabber settings. System Administrators can also [change the templates](#) used for notifications.

### What Will Be Watched

Events related to the following projects and build configurations	<p>Select the projects / build configurations whose builds you want to monitor. Notification rules defined for a project will be propagated to its subprojects. To monitor all of the builds of all the projects' build configurations, select Root project.</p> <p>Use the following options for granular control over the notifications in the selected projects / build configurations:</p> <ul style="list-style-type: none"><li>• Branch filter - Select this option to receive alerts only on the builds from the specified branches. By default, only the default branch is monitored. You can edit the branches to be monitored.</li><li>• Builds with my changes only ( since TeamCity 10.0) - Select this option to limit notifications to builds containing your changes only.</li></ul> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"><p> Make sure your Version Control Username Settings are correct.</p></div> <ul style="list-style-type: none"><li>• My favorite builds only - You can limit notifications to your <a href="#">favorite builds</a>.</li></ul>
---	--

System wide events

Select to be notified about investigations assigned to you.

## Which Events Will Trigger Notifications

Build fails	<p>Check this option to receive notifications about all of finished failed builds in the specified projects and build configurations. If an investigation for a build configuration was assigned to someone while the build was in progress, the failed build notification is sent only to that user.</p> <p>Note that if Builds with my changes only is selected in the Watch area, TeamCity will notify you if a build with your changes fails, and will keep sending notifications on each 'incomplete' build afterwards until a successful one. An incomplete build is a finished build which failed with at least one of the following errors:</p> <ul style="list-style-type: none"> <li>• Execution timeout</li> <li>• JVM crashed</li> <li>• JVM Out of memory error</li> <li>• Unable to collect changes</li> <li>• Compilation error</li> <li>• Artifacts publishing failed</li> <li>• Unknown Failure reason</li> </ul>	
	<p>Only notify on the first failed build after successful</p> <p>Check this option to be notified about only the first failed build after a successful build or the first build with your changes. When using this option, you will not be notified about subsequent failed builds.</p>	
	Only notify on new build problem or new failed test	Check this option to be notified only if a build fails with a new build problem or a new failed test.
	<p>Build is successful</p> <p>Check this option to receive notifications when a build of the specified projects and build configurations executed successfully.</p>	
The first error occurs	Only notify on the first successful build after failed	Check this option to receive notifications when only the first successful build occurs after a failed build. Notifications about subsequent successful builds will not be sent.
	<p>Build starts</p> <p>Check this option to receive notifications as soon as a build starts.</p>	
Build fails to start	<p>Check this option to receive notifications when a build <a href="#">fails to start</a>.</p>	
Build is probably hanging	<p>Check this option to receive notifications when TeamCity identifies a build as <a href="#">hanging</a>.</p>	
Investigation is updated	<p>Check this option to receive notifications on changing a build configuration or test investigation status, e.g. someone is investigating the problem, or problems were fixed, or the investigator changed.</p>	
Tests are muted or unmuted	<p>Check this option to receive notifications on the test <a href="#">mute</a> status change in the affected build configurations.</p>	
Investigation assigned to me	<p>This option is available only if the System wide events option is selected in the Watch area. Check the option to be notified each time you start investigating a problem.</p>	

## Rules Processing Order

TeamCity applies the notification rules in the order they are presented. TeamCity checks whether a build matches any notification rule, and sends a notification according to the first matching rule; the further check for matching conditions for the same build is not performed. You can reorder the configured notification rules.

The user rules are applied first, then the group rules are applied.

The group rules are processed in hierarchical order: starting from child groups to parent ones.

If there are more than one parent with their own rules set, the inherited rules are processed top-bottom corresponding to the

order they are presented on the Notification Rules user profile tab (alphabetically).

## Unsubscribing and Overriding Existing Rules

You may already have some notification rules configured by your System Administrator for the user group you're included in.

- To unsubscribe from or override group notifications, add your own rule with the same watched builds and different notification events.
- To unsubscribe from all events, add a rule with the corresponding watched builds and no events selected.

## Customizing RSS Feed Notifications

TeamCity allows obtaining information about the finished builds or about the builds with the changes of particular users via RSS. You can customize the RSS feed from the TeamCity Tools sidebar of My Settings & Tools page using the [Syndication Feed](#) section (click customize to open the Feed URL Generator options) or from the home page of a build configuration. TeamCity produces a URL to the syndication feed on the basis of the values specified on the Feed URL Generator page.

### Feed URL Generator Options

Option	Description
Select Build Configurations	
List build configurations	<p>Specify which build configurations to display in the Select build configurations or projects field. The following options are available:</p> <ul style="list-style-type: none"><li>• With the external status enabled: if this option is selected, the next field shows the list of build configurations with the <a href="#">Enable status widget option set</a>, and build configurations visible to everybody without authentication.</li><li>• Available to the Guest user: Select this option to show the list of build configurations, which are visible to a user with the Guest permissions.</li><li>• All: Select this option to show a complete list of build configurations; HTTP authorization is required. Selecting this option enables the <a href="#">Feed authentication settings</a> field. If the external status for the build configuration is not enabled, the feed will be available only for authorized users.</li></ul>
Select build configurations or projects	Use this list to select the build configurations or projects you want to be informed about via a syndication feed.
Feed Entries Selection	
Generate feed items for	Specify the events to trigger syndication feed generation. You can opt to select builds, changes or both.
Include builds	<p>Specify the types of builds to be informed about:</p> <ul style="list-style-type: none"><li>• All builds</li><li>• Only successful</li><li>• Only failed</li></ul>
Only builds with changes of the user	Select the user whose changes you want to be notified about. You can get a syndication feed about the changes of all users, yours only, or of a particular user from the list of users registered to the server.
Other Settings	The following options are available only if All is selected in the List build configurations section.
Feed Authentication Settings:	<p>Include credentials for HTTP authentication - Check this box to specify the user name and password for automatic authentication. If this option is not checked, you will have to enter your user name and password in the authorization dialog box of your feed reader.</p> <p>TeamCity User, Password - configurable when the Include credentials... option is checked. Type the user name and password which will be used for HTTP authorization.</p>
Copy and Paste the URL into Your Feed Reader or Subscribe	This field displays the URL generated by TeamCity on the basis of the values specified above. You can either copy and paste it to your feed reader or click the Subscribe link.

## Additional Supported URL Parameters

In addition to the URL parameters available in the Feed URL Generator, the following parameters are supported:

Parameter Name	Description
itemsCount	a number; limits the number of items to return in a feed. Defaults to 100.
sinceDate	a negative number; specifies the number of minutes. Only builds finished within the specified number of minutes from the moment of feed request will be returned. Defaults to 5 days.
template	name of the custom template to use to render the feed (<template_name>). The file <TeamCity Data Directory>\config\<template_name>.ftl should be present on the server. See the <a href="#">corresponding section</a> on the file syntax.

By default, the feed is generated as an Atom feed. Add &feedType=rss\_0.93 to the feed URL to get the feed in RSS 0.93 format.

### Example

Get builds from the TeamCity server located at "http://teamcity.server:8111" address, from the build configuration with ID "bt1", limit the builds to those started with the last hour but no more than 200 items:

```
http://teamcity.server:8111/httpAuth/feed.html?buildTypeId=bt1&itemsType=builds&sinceDate=-60&itemsCount=200
```

### See also:

[Administrator's Guide: Customizing Notifications](#)

## Viewing Your Changes

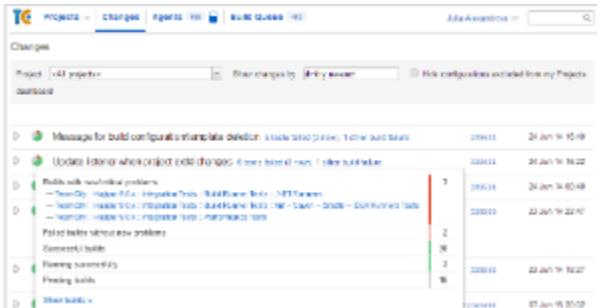
Monitoring the quality of the codebase is essential for a development team: a project developer needs to see whether his/her commit brought a build failure or not; for a project leader it is important to detect the code at fault for a build failure to be able to have the situation rectified early, so other members of the team are not inconvenienced.

The Changes page of the TeamCity Web UI allows you to review the commits made by all TeamCity users and see how they have affected builds. You can filter the results with the user selector on the page.

 Changes made by a user are displayed correctly only when appropriate [VCS usernames](#) are defined.

By default, the page does not show the commits to the build configurations hidden by the current user on the Projects dashboard. To remove this filter and view all build configurations, deselect the Hide configurations excluded from my Projects box.

Each change now has a new pie-chart icon with pie slices showing the relative size of pending, successful, as well as old and new problematic builds affected by the change. Hovering over/clicking the pie-chart icon gives a visual representation of how the user commit has affected different builds. Builds with new/critical problems are listed by default. Expanding the change or clicking the See builds link lists all builds with the change.



From this page you can:

- View all commits and changes included into personal builds.
- View how changes have affected the builds.
- See whether there are new failed tests caused by changes.
- Navigate to the issue tracker, if the [issue tracker integration is configured](#).
- Open possibly problematic files in your IDE: the option is available if the plugin for this IDE is [installed](#) and you are logged in to TeamCity from within this IDE.
- Navigate to change details.
- View detailed data for each change on the dedicated tabs. To switch between tabs for the currently selected change, use Tab/Shift+Tab or mnemonics: 'T' for tests, 'B' for builds, 'F' for files.
- View builds with any of the changes. By default, successful and pending build configurations are hidden from display. Uncheck the corresponding box to view all builds with the change.

Note, that problems which have an investigator/responsible are not considered critical (unless you are the investigator).

See also:

[Concepts: Build State | Change](#)

## Working with Build Results

In TeamCity a build goes through several states:

- upon some event the build trigger adds the build to the queue where the build stays waiting for a free agent
- the build starts on the agent and performs all configured build steps
- the build finishes and becomes a part of the build history of this build configuration.

In TeamCity all information about a particular build, whether it is queued, running or finished, is accumulated on the build results page. The page can be accessed by clicking the build number or build status link.

Besides providing the build information, this page enables you to:

- [run a custom build](#) using the Run... button
- use the Actions menu to do the following:
  - add a build to [favorites](#)
  - add a comment
  - pin the build
  - tag the build
  - change the build status, marking the build as [failed](#) or [successful](#)
  - [label this build sources](#)
  - remove the build
- [edit the configuration settings](#)

## Build Details

The build results page can be accessed the Build Configuration home page and from various places in the TeamCity web UI where a build number or build status appears as a link.

Some data is accessible only after the build is finished, some details like [Changes](#), [Build parameters](#), and [Dependencies](#) are also applicable to the build while it is waiting in the queue.

The following build information is available on the page:

- [Build Overview](#)
  - [Tests](#)
- [Changes](#)
- [All Tests](#)
  - [Test History](#)
  - [Test Duration Graph](#)
- [Build Log](#)
- [Parameters](#)
- [Dependencies](#)
- [Related Issues](#)
- [Build Artifacts](#)
- [Code Coverage Results](#)
- [Code Inspection Results](#)
- [Duplicates](#)
- [Maven Build Info](#)
- [Internal Build ID](#)

Depending on the build runners enabled as your build steps, the number of tabs on the page and the information on the Overview tab may vary.

## Build Overview

The Overview Tab displays general information about the build, such as the build duration, the agent used, trigger, dependencies, etc.

If a build is queued, the tab displays the position of the build in the queue, the time the build is supposed to start, etc.

If a build is running, the tab displays the build progress. You can also stop a running build using the corresponding link on the Overview tab or the appropriate option from the Actions button drop-down.

If another build of this same build configuration is concurrently running, a small window appears on the Overview tab with the following information:

- The build results of that build linking to the build results page
- A changes link with a drop-down list of changes included in that build
- The build number, time of start, and a link to the agent it is running on.

If a build is probably hanging, the corresponding notification is displayed at the top of the Overview tab. In this case TeamCity provides a link to view the process tree of the running build and the thread dumps of each Java or .Net process in a separate frame. If the process is not Java or .Net, its command line is retrieved. The possibility to view the thread dump is supported for the following platforms:

- Windows, with JDK 1.3 and higher
- Windows, with JDK 1.6-1.8, using jstack utility
- Non-Windows, with JDK 1.5-1.8, using jstack utility

The information on the tab may vary depending on the build runners enabled. If configured, you will see the Code Coverage Summary and a link to the full report or/ and the number of duplicates found in your code and a link opening the Duplicates tab, etc. Refer to the sections below for details on [Code Coverage](#) and [Duplicates Tabs](#).

If there were problems in the build, the Overview tab also displays them.

If the build has failed tests, you can view them on the Overview tab of the build results page.

## Tests

Successful or failed tests in this build (if any) are displayed on the Overview tab.

For each failed test, you can view its stacktrace, the diff between the expected and actual values, jump to the test [history](#), assign a team member to investigate its failure, open the test in your IDE and/or start fixing it right away.

To view all tests related to the build, use the dedicated Tests tab. [Learn more](#).

## Changes

From the Changes tab you can:

- Review all changes included in the build with their corresponding [revisions](#) in version control
  - review changes included in the build that the current build depends on: if the current build configuration has an artifact dependency, and the artifact downloaded in the current build has changed compared to the one downloaded in the previous build of the current configuration, the Artifact dependencies changes node appears displaying the build which was used to download artifact dependencies from, and the changes which were included in that build.
- [Label the build sources](#) (prior to TeamCity 8.1)
- [Configure the VCS settings](#) of the build configuration (if you have enough permissions).

For each change on this page you can:

- Explore the change in details
- View which dependent build the changes come from or builds with snapshot dependencies with the "Show changes from snapshot dependencies" option enabled (since TeamCity 2017.1). On hovering over the  icon, the number of the dependent build is displayed; clicking the link opens the Changes tab of the dependent build.
- Navigate to the Change Details by clicking a changed file link
- [Trigger a custom build](#) with this change
- Download patch
- Download patch to your IDE
- Review the change in an [external change viewer](#), if configured by the administrator.

The Changes tab provides advanced filtering capabilities for the list of changes. Enabling Show graph displays the changes as a graph of commits to the VCS roots related to this build.

The graph appears on the left of the list and allows you to get the view of the changes with a variable level of detailing. You can:

- View the VCS roots which were changed in this build, each of the roots being represented as a bar.
- Navigate to a graph node to display the VCS root revision number.
- Click a bar to select a single VCS root. The changes not pertaining to this root are grayed out.
- If there have been merges between the branches of the repository, the graph displays them. To collapse a bar, navigate to its darker area and click it to hide history of merges. The dotted line will indicate that the bar is expandable.
- If your VCS root has subrepositories (marked S in the list of changes), navigate to a node in the parent to see which commits in subrepositories are referenced by this revision in the parent.

You can select to view the modified files by checking the Show files box. Clicking a file name opens the diff viewer.

## All Tests

To view all the tests for a particular build, open the build results page, and navigate to the Tests tab.  
On this page:

Item	Description
Download all tests in CSV	Click the link to download a file containing all the build tests results.
Filtering options	Use this area to filter the test list: <ul style="list-style-type: none"><li>• Select the type of items to view: tests, suites, packages/namespaces or classes.</li><li>• Type in the string (e.g. test name from the list) thus providing change of scope for the list.</li><li>• Select a test status.</li></ul>
Show	Select the number of tests to be shown on a page.
Status	Shows the status (OK, Ignored, and Failure) of the test. Failed tests are shown as red Failure links, which you can click to view and analyze the test failure details. Click the header above this column to sort the table by status.
Test	Click the name of a class, a namespace/package, or a suite to view only the items included in it. Click the arrow next to the test name to view the test history, open the test in the Build Log, start investigation of the failed test, or open the failed test in IDE.
Duration	Shows the time it took to complete the test. You can view the Test Duration Graph described below by clicking this icon:
Order#	Shows the sequence in which the tests were run. Click the header above this column to sort by the test order number.

## Test History

To navigate to the history of a particular test, click the arrow next to the test name and select Test History from the drop-down.

There are several places where tests are listed and from where you can open Test History.  
For example:

- Project Home page | Current Problems tab
- Project Home page | Current Problems tab | Problematic Tests
- Build Results page | Overview tab
- Build Results page | Tests tab
- Projects | <build with failed tests> | build results drop-down

Clicking the Test history link opens the Test details page where you can find following information:

- The test details section including test success rate and test's run duration data:
- the Test Duration Graph. For more information, refer to the [Test Duration Graph](#) description below.
- Complete test history table containing information about the test status, its duration, and information on the builds this test was run in.

## Test Duration Graph

The Test Duration graph (see the screenshot above) is useful for comparing the amount of time it takes individual tests to run on the builds of this build configuration.

Test duration results are only available for the builds which are currently in the build history. Once a build has been cleaned up, this data is no longer available.

You can perform the following actions on the Test Duration Graph:

- Filter out the builds that failed the test by clearing the Show failed option.
- Calculate the daily average values by selecting the Average option.
- Click a dot plotted on the graph to jump to the page with the results of the corresponding build.
- View a build summary in the tooltip of a dot on the graph and navigate to the corresponding Build Results page.
- Filter information by agents selecting or clearing a particular agent or by clicking All or None links to select or clear all agents.

## Build Log

For each build you can view and download its build log. More information on build logs in TeamCity is available [here](#).

## Parameters

All system properties and environmental variables which were used by a particular build are listed on the Parameters tab of the build results page. [Learn more about build parameters](#).

The Reported statistic values page shows [statistics values](#) reported for the build and displays a statistics chart for each of the values on clicking the View Trend icon .

## Dependencies

If a finished build has artifact and/or snapshot dependencies, the Dependencies tab is displayed on the build results page. Here you can explore builds whose artifacts and/or sources were used for creating this build (Downloaded artifacts) as well as the builds which used the artifacts and/or sources of the current build (Delivered artifacts).

Additionally, you can view indirect dependencies for the build. That is, for example, if build A depends on build B which depends on builds C and D, then these builds C and D are indirect dependencies for build A.

## Related Issues

If you have [integration with an issue tracker](#) configured, and if there is at least one issue mentioned in the comments for the included changes or in the comments for the build itself, you will see the list of issues related to the current build in the Issues tab.



If you need to view all the issues related to a build configuration and not just to particular build, you can navigate to the Issues Log tab available on the build configuration home page, where you can see all the issues mapped to the comments or filter the list to particular range of builds.

## Build Artifacts

If the build produced [artifacts](#), they all are displayed on the dedicated Artifacts tab.

## Code Coverage Results

If you have code coverage configured in your build runner, a dedicated tab with the full HTML code coverage report appears on the build results page.

By clicking the links in the Coverage Breakdown section, you can drill-down to display statistics for different scopes, e.g. Namespace, Assembly, Method, and Source Code.

## Code Inspection Results

If configured, the results of the [Code Inspection](#) build step are shown on the Code Inspection tab. Use the left pane to navigate through the inspection results; the filtered inspections are shown in the right pane.

- Switch from the Total to Errors option, if you're not interested in warnings.
- Use the scope filter to limit the view to the specific directories. This makes it easier for developers to manage specific code of interest.
- Use the inspections tree view under the scope filter to display results by specific inspection.
- Note that TeamCity displays the source code line number that contains the problem. Click it to jump the code in your IDE.

## Duplicates

If your build configuration has Duplicates build runner as one of the build steps, you will see the Duplicates tab in the build results.

The tab consists of:

- A list of duplicates found. The new only option enables you to show only the duplicates that appeared in the latest build.
- A list of files containing these duplicates. Use the left and right arrow buttons to show selected duplicate in the respective pane in the lower part of the tab.
- Two panes with the source code of the file fragments that contain duplicates.
- Scope filter in the the upper-left corner lists the specific directories that contain the duplicates. This filtering makes it easier for developers to manage the code of interest.

## Maven Build Info

For each Maven build the TeamCity agent gathers Maven specific build details, which are displayed on the Maven Build Info tab of the build results after the build is finished.

## Internal Build ID

In the URL of the build result page you can find parameter "buildId" with a numeric value. This number is internal build id uniquely identifying the build in the TeamCity installation.

You might need this ID when constructing URL manually. For example for [REST API](#), [downloading artifacts](#).

See also:

Concepts: [Build Log](#) | [Build Artifact](#) | [Change](#) | [Code Coverage](#)

User's Guide: [Investigating and Muting Build Problems](#) | [Viewing Tests and Configuration Problems](#)

Administrator's Guide: [Creating and Editing Build Configurations](#)

## Investigating and Muting Build Problems

When for some reason a build fails, TeamCity provides handy means of figuring out which changes might have caused the build

failure.

You can assign some of your team members to investigate what caused the failure of this build or to start the investigation yourself. When an investigator is set, he/she receives the corresponding notification.

TeamCity also provides a way to mute a build problem (similarly to [muting failed tests](#)) so that it not affect build status for future builds.

## Investigating or Muting Build Problems

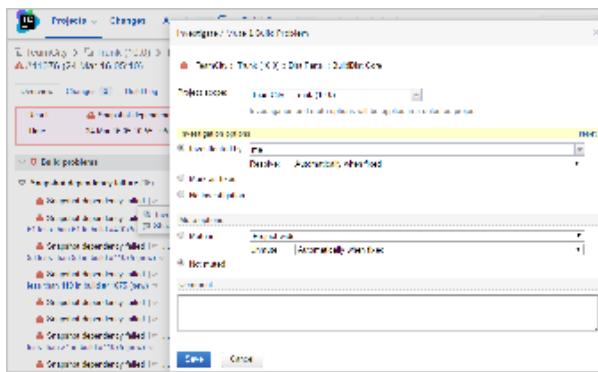
To assign an investigation of a failed build:

1. Navigate to the Overview tab of the Build Configuration Home page, or the Overview tab of the Build Results page, click the Start investigation... link.
2. Select a team member's name from the User drop-down list.
3. Select the condition to remove the investigation: when the build configuration becomes green or the test passes (default) or specify that an investigation should be resolved manually.  
**Manual resolution** is useful with so called "flickering tests", i.e. when a test fails from time to time and the "green" status of a build is not an indicator of the problem resolution.
4. Save your investigation.

To investigate a problem in more than one build configuration, click more and select the desired build configurations.

To assign an investigation / mute a particular build problem:

1. Navigate to the Overview tab of the [Build Results page](#), click the arrow next to the build problem, click the Investigate.. link.
2. The Investigate / Mute Build problem dialogue opens. Select the investigation or muting options and save them:

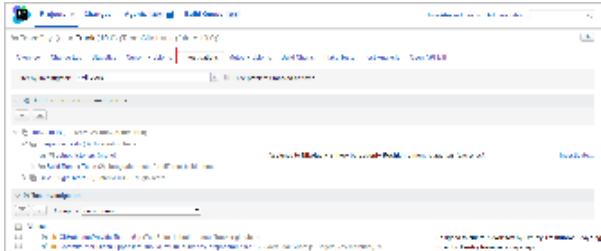


3.

To assign an investigation / mute a particular test, navigate to the [Tests tab](#), [Current Problems tab](#) or [Problematic Tests](#) of the Build Results page, click the arrow next to the test name and select Investigate....

## Viewing Investigations and Mutes

A subject for investigating/ muting is not necessarily the whole build, it can also be a build problem, a particular test. For each project you can find out what problems are currently being investigated and by whom using the dedicated Investigations page. All mutes are listed on the Muted Problems tab:



## My Investigations

To view all problems assigned to you to investigate, click the box with a number next to your name in the top tight-hand corner.

From My Investigations page you are able not only to see the investigations assigned to you in different projects, but you can view the problems in the build log, manage them: mark as fixed, give up the investigation, re-assign, or mute.

Note that for each failed test on this page you can get a lot of information instantly, without leaving this page. For example:

- you can see all build configurations where this test is currently failing
- you can see the current stacktrace and the information about the build where the test is currently failing
- you can also see the information about the first failure of this test, again with the stacktrace and build.

The investigations assigned to you are also highlighted in the Web UI if you enable the "Highlight my changes and investigations" option on in your [profile settings](#).

See also:

[Concepts: Build Configuration Status, Muting Test Failures](#)

[User's Guide: Viewing Tests and Configuration Problems](#)

## Viewing Tests and Configuration Problems

In this section you will read about:

- [Viewing Problems on Project Overview Page](#)
- [Using Current Problems Tab](#)
- [Flaky Tests](#)

### Viewing Problems on Project Overview Page

The project overview page also displays the number of tests failed in build configurations as well as other problems. When the data in the problematic build configuration is not expanded, the link under the tests number takes you to the Current Problems page (see [below](#)).

When you expand the build configuration data, the problem summary appears. The presentation of problems is shortened if your browser does not have enough horizontal space.

### Using Current Problems Tab

To view problematic build configurations and tests in your project, open the Project Home page and go to the Current Problems tab.

By default, the Current Problems tab displays data for all build configurations within a project. To limit the data displayed, use the Filter by build configuration drop-down.

From this page you can view problems in your project divided into the following groups:

- build configuration problems,
- failed tests,
- [muted test failures](#),
- build problems.

Each of the sections is expandable and you can further drill down to the smallest relevant item using the up and down arrows



The links appearing in build configuration problems and build problems sections enable you to monitor a great deal of useful data, e.g. you can navigate to build results, view build log, changes, etc. More options are available when you click the arrow next to a link.

The failed test and muted failures sections have grouping options and allow viewing the test stacktrace available when clicking the test name link. You can also view the test history, open the test in the active IDE, start investigating a particular test failure, fix and unmute a test or start investigating a particular test failure.

### Flaky Tests

TeamCity detects flaky tests and displays them on the dedicated tab for a given project.

A flaky test is a test that is unstable (can exhibit both a passing and a failing result) with the same code.

Flaky test detection is based on the following heuristics:

1. High flip rate (Frequent test status changes). A flip in TeamCity is a test status change — either from OK to Failure, or vice versa. The Flip Rate is the ratio of such "flips" to the invocation count of a given test, measured per agent, per build configuration, or over a certain time period (7 days by default). A test which constantly fails, while having a 100% failure rate, will have its flip rate close to zero; a test which "flips" each time it is invoked will have the flip rate close to 100%.  
If the flip rate is too high, TeamCity will consider the test flaky.
2. Different test status for build configurations with the same VCS change: if two builds of different configurations are run on the same VCS change and the test result is different in these builds, the test is reported as flaky. This may be an indication of environmental issues.
3. If the status of a test 'flipped' in the new build with no changes, i.e. a previously successful test failed in a build without changes or a previously failing test passed in a build without changes, TeamCity will consider the test flaky.
4. Different test status for multiple invocations in the same build (flaky failure): if the same test is invoked multiple times and the test status flips, TeamCity will consider the test flaky.

This heuristic is supported for [TestNG](#) unit tests with `invocationCount` [1][2] greater than 1.



There is a known issue with parameterized test invocations with TestNG: TeamCity relies on the Surefire and Failsafe Maven plugins for the purposes of test result reporting, and these plugins ignore test parameters in their XML output. As a result, parameterized test invocations within the same build are erroneously treated as multiple identical test invocations, and, if some of the invocations fail, the test is marked as flaky with the Flaky Failure reason. See [the related issue](#).

For JUnit tests, a 3rd party [tempus-fugit](#) library can be used together with JUnit. It is sufficient to annotate a test with `@Intermittent` and use the `IntermittentTestRunner` test runner, as in the minimal example below:

```
import org.junit.Test;
import org.junit.runner.RunWith;

import com.google.code.tempusfugit.concurrency.IntermittentTestRunner;
import com.google.code.tempusfugit.concurrency.annotations.Intermittent;

@RunWith(IntermittentTestRunner.class)
public class MultipleViaTempusFugit {
    @Test
    @Intermittent(repetition = 10)
    public void test() {
        // ...
    }
}
```

If such a test flakes at least once at multiple invocations within a single build, the Flaky Failure heuristic will be triggered.

Such tests are displayed on the dedicated project tab, Flaky Tests, along with the total number of test failures, the flip rate for the given test and reasons for qualifying the test as a flaky one.

You can also see if the test is flaky when viewing the expanded stacktrace for a failed test on the build results page.

Result: Tests failed: 13 (2 new), passed: 16706, ignored: 138, muted: 21  
 Time: 30 Jul 16 03:02 - 10:02 (6h:59m)

Agent: W7-esxi-13 (TeamCity pool)  
 Triggered by: Schedule Trigger on 30 Jul 16 03:00

Code coverage summary [View full report >](#)

Classes:	70.1%	Methods:	61.8%	Blocks:	35.2%	Lines:	61.9%
	10466/14920		50631/81864		1016105/2882385		194580/314148
Diff:	-0.06%	Diff:	-0.02%	Diff:	-0.01%	Diff:	-0.02%

▼ 13 tests failed (2 new)

All tests

- Integration tests: jetbrains.buildServer.agent.impl (1)
  - \* UpdateSourcesAutoTest should do server checkout if agent cannot checkout from repository | [View test history >](#)
  - This test looks flaky.
  - Different test status of build configurations with the same VCS change
  - Frequent test status changes: 2 changes out of 38 invocations
- Maintenance Suite: jetbrains.buildServer.serverSide.maintenance (1)

As with any failed test, you can assign investigations for a flaky test (or multiple tests). For flaky tests the resolution method is automatically set to 'Manual'; otherwise the investigation will be automatically removed once the test is successful, which does not mean that the flaky test has been fixed.

Note that if **branches** are configured for a VCS Root, flaky tests are detected for the default branch only.

## See also:

[Concepts: Testing Frameworks](#)  
[User's Guide: Working with Build Results](#)

## Viewing Build Configuration Details

The Build Configuration Home page provides the configuration details and enables you to:

- [run a custom build](#) using the Run... button
- using the Actions menu
  - [pause triggers](#)
  - [check for pending changes](#)
  - [enforce clean checkout](#)
  - [assign an investigation](#)
- [edit the configuration settings](#)

The build configuration details are separated into several tabs whose number may vary depending on your server or project configuration, e.g. [TeamCity integration with other tools](#), etc.

By default the following tabs are available:

- [Overview](#)
- [History](#)
- [Change Log](#)
- [Statistics](#)
- [Compatible Agents](#)
- [Pending Changes](#)
- [Settings](#)

## Overview

Provides information on:

- Pending Changes also listed as a separate tab, see details [below](#)
- the Current Status of the build configuration, and, if applicable:
  - the number of queued builds
  - for a running build - the progress details with the Stop option to terminate the build
  - for a failed build - the number and [agent](#), etc.

- the [Recent History](#) section lists builds of the current build configuration

## History

Displays [Build History](#) on a separate page and allows filtering builds by build agents, [tagging builds](#) and filtering them by tags (if available).

## Change Log

By default, lists changes from builds finished during the last 14 active days. Use the [show all](#) link to view the complete change log.

The page shows the change log with its graph of commits to the monitored branches of all VCS repositories used by the current build configurations and the repositories used by the [dependencies](#) and [dependent configurations](#) of the current configuration.

## Statistics

Displays the collected statistical data as [visual charts](#).

## Compatible Agents

Lists all authorized agents. Agents meeting [Agent Requirements](#) are listed as compatible. For each incompatible agent, the reason is provided.

The agents belonging to the [pool\(s\)](#) associated with the current project are listed first.

## Pending Changes

Lists [changes](#) waiting to be included in the next build on a separate page.

## Settings

Lists the current [build configuration settings](#) on a separate page.

## Statistic Charts

On this page:

- [Project Statistics](#)
- [Build Configuration Statistics](#)
  - Success Rate
  - Build Duration (excluding the checkout time)
  - Time Spent in Queue
  - Test Count
  - Artifacts Size
  - Time to Fix Tests
  - Code Coverage
  - Code Duplicates
  - Code Inspection
- [Tests Statistics](#)
- [Custom Charts](#)

To help you track the condition of your projects and individual build configurations over time, TeamCity gathers statistical data across all their history and displays it as visual charts. The statistical charts can be divided into the following categories:

- [Project-level statistics](#) available at the Project home page | Statistics tab.
- [Build Configuration-level statistics](#) available at the Build Configuration home page | Statistics tab.

Regardless of the selected statistics level in the Statistics tab, you can:

- Use the branch filter to view the results from the specified branches only
- Download each chart data in the CSV format using the  icon
- Configure the Y-axis settings for each chart using the  icon in the upper left corner
- Select a time range for each type of statistics from the Range drop-down list.
- Filter the information by data series, for example, by the Agent name or result type.
- View average values by selecting the Average check box.
- Filter out failed builds and show only successful builds with the unchecked Show Failed option.
- View the build summary information when you mouse-over a build and navigate to the build results page using the

build number link.

**⚠** Statistics include information about all the builds across all its history. However, according to the clean-up policy, some of the build results may be removed. In this case, you can only view the summary information about the build, but cannot jump to the build results page.

## Project Statistics

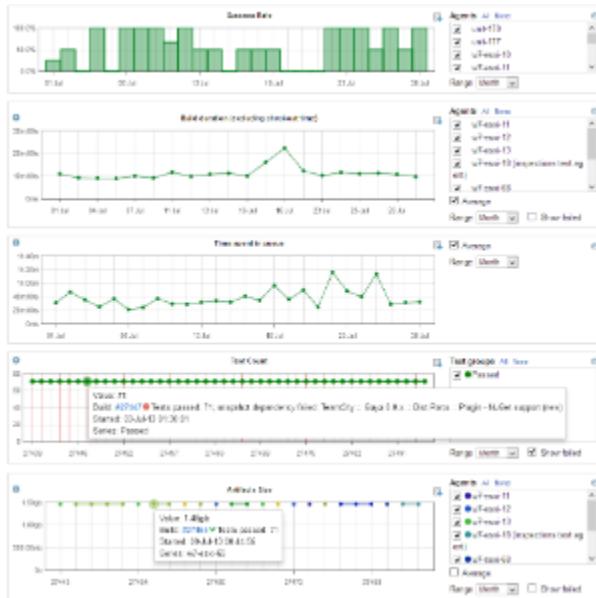
For each project TeamCity provides visual charts with statistics gathered from all build configurations included in the project over the entire history of the project. These charts show statistics for code coverage, code inspections and code duplicates for build configurations within the project when the corresponding data is available for the builds of this project's configurations.

You can adjust the set of project-level charts in the following ways:

- Disable charts of particular type
- Specify build configurations to be used in the chart
- Add custom project-level charts

## Build Configuration Statistics

Statistics information is also available at the build configuration level. These charts demonstrate the successful build rate, the build duration, time builds spent in queue, time that took to fix tests, artifact size, and test count. The charts also show code coverage, duplicates and inspection results if these are included in the respective build configuration.



It is also possible to add custom charts. Unlike project-level charts, pre-defined charts on the build configuration level cannot be disabled.

The charts generated automatically by TeamCity include the following types:

### Success Rate

This chart tracks the build success rate over the selected period of time.

### Build Duration (excluding the checkout time)

This chart allows the viewer to monitor the duration of the build. To get a better idea of the build duration changes, select a single build agent or build agents with similar processors.

### Time Spent in Queue

This chart tracks the time it took to actually start a build after it was scheduled. This information is helpful for managing the build agent and prioritizing build configurations.

## Test Count

Green, grey and red dots show the number of tests (JUnit, NUnit, TestNG, etc.) that passed, were ignored, or failed in the build respectively. All invocations of the same test within a single build are counted as one test. The information about individual tests is available on the build results page.

## Artifacts Size

This chart tracks the total size of all artifacts produced by the build.

## Time to Fix Tests

Time to fix a test is reported for a finished build with a newly failed test when a subsequent build where this test passed finishes (this means that the metric can be updated in a finished build on new builds finishing). The time is calculated as the difference between the start times of these builds.

This chart tracks the maximum amount of time it took to fix the tests of a particular build. If not all build tests were fixed, a red vertical stripe is displayed.

## Code Coverage

Blue, green, dark cyan, and purple dots show respectively the percentages of the classes blocks, lines and methods covered by the tests.

## Code Duplicates

This chart tracks the number of duplicates discovered in the code.

## Code Inspection

This chart displays red and yellow dots to track the number of discovered errors and warnings respectively.

## Tests Statistics

You can also find some useful statistics for a particular test: Test duration graph on "Test History" page, which allows comparing the amount of time it takes individual tests to run on the builds of this build configuration. For more details, please refer to [related page](#).

## Custom Charts

It is possible to [customize project-level charts](#) or/and configure your own statistical charts, e.g. to display the total build duration, including the checkout time, the duration of all build stages, artifact resolving and artifact publishing or a chart displaying the duration of each build stage, etc. See a [dedicated page](#) for details

## See also:

[Concepts: Build Configuration | Build State | Change](#)  
[Administrator's Guide: Customizing Statistics Charts](#)

## Search

After you have installed and started running TeamCity, it collects the information on builds, tests and so on and indexes it. You can search builds by build number, tag, build configuration name and other different parameters specifying one or several keywords. You can also search for builds by text in build logs, and by the [external id](#) of a build configuration.

On this page:

- Search Query
  - Differences from Lucene Syntax
  - Performing Fuzzy Search
  - Boolean Operators and Wildcards Support
- Complete List of Available Search Fields, Shortcuts, and Keywords
  - Search Fields
  - Shortcuts
    - Using Double-Colon
  - "Magic" Keywords
- Search by Build Log
- Resetting Search Index

## Search Query

In TeamCity you can search for builds using the [Lucene query syntax](#); however, a TeamCity search query has two major differences described [below](#).

To narrow your search and get more precise results, use the available search fields - indexed parameters of each build. For complete list of available search fields (keywords), refer to [this section of the page](#).

### Differences from Lucene Syntax

When using a search query in TeamCity, mind the following major differences from the Lucene native syntax:

1. By default, TeamCity uses the "AND" operator in a query. That is, if you type in the following query: "failed @agent123", then you will get a list of all builds that have the keyword "failed" in any of its search fields, and were run on the build agent named "agent123".
2. By default, TeamCity uses the "prefix search", not the exact matching like Lucene. For example, if you search for "c:main", TeamCity will find all builds of the build configuration whose name starts with the "main" string.

### Performing Fuzzy Search

You also have a possibility to perform fuzzy search using the tilde ("~") symbol at the end of a single word term to search for items with similar spelling.

### Boolean Operators and Wildcards Support

You can combine multiple terms with Boolean operators to create more complex search queries. In TeamCity, you can use AND, "+", OR, NOT and "-".

 When using Boolean operators, type them ALL CAPS.

- AND (same as a plus sign). All words that are linked by the "AND" are included in the search results. This operator is used by default.
- NOT (same as minus sign in front of a query word). Exclude a word or phrase from search results.
- OR operator helps you to fetch the search terms that contain either of the terms you specify in the search field.

TeamCity also supports the "\*" and "?" wildcards in a query.

 It is not recommended to use the asterisk (\*) at the beginning of the search term as it may require a significant amount of time for TeamCity to search its database. For example, the \*onfiguration search term is incorrect.

## Complete List of Available Search Fields, Shortcuts, and Keywords

### Search Fields

When using search keywords, use the following query syntax:

<search field name>:<value to search>

Search Field	Shortcut	Description	Example
agent		Find all builds that were run on the specified agent.	agent:unit-77, or agent:agent14*

build		Find all builds that include changes with the specified string.	build:254 or build:failed
buildLog		Find all builds that include certain text in build logs. It is <a href="#">disabled by default</a> .	buildLog: "NUnit report"
changes		Find all builds that include changes with the specified string.	changes:(fix test)
committers		Find all build that include changes committed by the specified developer.	committers:ivan_ivanov
configuration	c	Find all builds from the specified build configuration.	configuration:IPR c:(Nightly Build)
file_revision		(not supported by default since TeamCity 2017.1) Find all builds that contain a file with the specified revision.	file_revision:5
files		(not supported by default since TeamCity 2017.1) Find all builds that include files with the specified file name.	files:
labels	l	Find all builds that include changes with the specified VCS label.	label:EAP l:release
pin_comment		Find all builds that were pinned and have the specified word (string) in the pin comment.	pin_comment:publish
project	p	Find all builds from the specified project.	project:Diana p:Calcutta
revision		Find all builds that include changes with the specified revision (e.g., you can search for builds with a specific changelist from Perforce, or revision number in Subversion, etc.).	revision:4536
stamp		Find all builds that started at the specified time (search by timestamp).	stamp:200811271753
status		Find all builds with the specified text in the build status text.	status:"Compilation failed"
tags	t	Find all builds with the specified tag.	tags:buildserver t:release
tests		Find all builds that include specified tests.	tests:
triggerer		Find all builds that were triggered by the specified user.	triggerer:ivan.ivanov
vcs		Find builds that have the specified VCS.	vcs:perforce
build problem		Since TeamCity 10.x, Find builds with the specified build problem	buildProblem:Compilation failed

## Shortcuts

In addition to above mentioned search fields, you can use the following shortcuts in your query:

**i** Note that when you use these shortcuts, do not insert the colon after it. That is, the query syntax is as follows: <shortcut><value to search>

Shortcut	Description	Example
#	Search for the specified build number.	#<number>, e.g. #1234
@	Find all builds that were run on the specified agent.	@<agent's name>, e.g. @buildAgent1

## Using Double-Colon

You can use the double-colon sign (::) to search for a project and/or build configuration by name:

- pro::best — search for builds of configurations with the names starting with "best", and in the projects with the names

- starting with "pro".
- mega:: — search for builds in all projects with names starting with "mega"
- ::super — search for builds of build configurations with names starting with "super"

## "Magic" Keywords

TeamCity also provides "magic" keywords (see table below for the complete list). These magic keywords are formed with the '\$' sign and a word itself. The word can be shortened up to one (first) syllable, that is, the \$labeled, \$l, and \$lab keywords will be equal in a query. For example, to search for pinned builds of the "Nightly build" configuration in the "Mega" project you can use any of the following queries:

- configuration:nightly project:Mega \$pinned
- c:nigh p:mega \$pin
- M::night \$pin

Magic word	Description
\$tagged	Search for builds with tags. For example, Calcutta::Master \$t query will result in a list of all builds marked with any tag of build configurations whose name starts with "Master" from projects with names beginning with "Calcutta".
\$pinned	Search for pinned builds.
\$labeled	Search for builds that have been labeled in VCS. For example, to find labeled builds of the Main project you can use following queries: p:Main \$labeled, or project:Main \$l, or m:: \$lab, etc.
\$commented	Search for builds that have been commented.
\$personal	Search for personal builds. For example, using -\$p expression in your query will exclude all personal builds from search results.

## Search by Build Log

By default, TeamCity does not search for builds by a certain text in build logs.

To enable search by the build logs, perform the following:

1. Since the logic will increase server memory usage, you need to increase [memory size](#) in -Xmx JVM option on at least 5 Gb (more if you have large build logs/many builds). You will need to switch to [64 bit JVM](#) for that.
2. Set the `tc.search.indexBuildLog=true` TeamCity [internal property](#)
3. [Reset](#) the search index

After re-indexing, TeamCity will be able to perform searching by specified text in the build logs and will list the relevant builds.

## Resetting Search Index

The search uses an "index" cached on the disk. Only the data previously added to the index is searchable and appears in the search results. Typically, data updates (such as new builds) cause only incremental updates to the index. This means that when some indexing setting is changed and the cached index on the disk gets out of sync, the index needs to be reset to reindex all the builds anew.

To reset the cached search index, click "reset" for the "search" entry on the Administration | Server Administration | Diagnostics, Caches tab or manually delete files from `<TeamCity Data Directory>\system\caches\search` while the server is not running. After that, reindexing will start which is a resource-intensive operation on the server and can take hours (depending on the number and "size" of builds, as well as the server machine and database performance). You can monitor the progress in the UI on the Search page or in the server logs.

## Maven-related Data

### Maven Project Data

In TeamCity you can find information about settings specified in your Maven project's `pom.xml` file on the dedicated Maven tab of build configuration. In addition to getting a quick overview of the settings, you can find Provided parameters in the upper section of this page, e.g. `maven.project.name`, `maven.project.groupId`, `maven.project.version`, `maven.project.artifactId`. You can use these parameters within your build. You can reference them within the build number pattern using %-notation. For example: `%maven.project.version%.{0}`.

### Maven Build Information

For each Maven build TeamCity agent gathers Maven specific build details, that are displayed on the Maven Build Info tab of the build results after the build is finished.

This page can be useful for build engineers when adjusting build configurations.

## Administrator's Guide

In this section:

- TeamCity Configuration and Maintenance
- Several Nodes Setup
- Configuring Cross-Server Projects Popup
- Managing Projects and Build Configurations
- Licensing Policy
- Integrating TeamCity with Other Tools
- Managing User Accounts, Groups and Permissions
- Customizing Notifications
- Assigning Build Configurations to Specific Build Agents
- Patterns For Accessing Build Artifacts
- Mono Support
- Maven Server-Side Settings
- Tracking User Actions
- Jenkins to TeamCity Migration Guidelines

## TeamCity Configuration and Maintenance

 Server configuration is only available to the System Administrators.

To edit the server configuration:

On the Administration page, select Global Settings.

The TeamCity Configuration section of the page displays the following information:

Setting	Description
Database:	The <a href="#">database</a> used by the running TeamCity server.
Data directory:	The <a href="#">&lt;TeamCity data directory&gt;</a> path with the ability to browse the directory.
Artifact directories:	<p>The list of the root directories used by the TeamCity server to store <a href="#">build artifacts</a>, build logs and other build data. The default location is <a href="#">&lt;TeamCity Data Directory&gt;/system/artifacts</a>. Note that artifacts can also be stored on <a href="#">external storage</a>.</p> <p>The list can be changed by specifying a new-line delimited list of paths. Absolute and relative (to TeamCity Data Directory) paths are supported.</p> <p>All the specified directories use the same <a href="#">structure</a>.</p> <p>When looking for build artifacts, the specified locations are searched for the directory corresponding to the build. The search is done in the order the root directories are specified. The first found build artifacts directory is used as the source of artifacts of this build.</p> <p>Artifacts for the newly starting builds are placed under the first directory in the list.</p>
Caches directory:	The directory containing TeamCity internal caches (of the VCS repository contents, search index, other), which be manually deleted to clear <a href="#">caches</a> .
Server URL	The <a href="#">configurable URL</a> of the running TeamCity server.

The Build Settings section allows configuring the following settings:

Setting	Description
Maximum build artifact file size:	Maximum size in bytes. KB, MB, GB or TB suffixes are allowed. -1 indicates no limit
Default build execution timeout:	Maximum time for a build. Can be overridden when defining <a href="#">build failure conditions</a> .

The Version Control Settings controls the following:

Setting	Description
Default VCS changes check interval:	Set to 60 seconds by default. Specifies how often TeamCity polls the VCS repository for VCS changes. Can be overridden when <a href="#">configuring VCS roots</a> .
Default VCS trigger quiet period:	Set to 60 seconds by default. Specifies a period (in seconds) that TeamCity maintains between the moment the last VCS change is detected and a build is added into the queue. Can be overridden when <a href="#">configuring VCS triggers</a> .

This section also includes:

- [Configuring Authentication Settings](#)
- [TeamCity Data Backup](#)
- [Projects Import](#)
- [Project Export](#)
- [TeamCity Startup Properties](#)
- [Configuring Server URL](#)
- [Configuring TeamCity Server Startup Properties](#)
- [Setting up Google Mail and Google Talk as Notification Servers](#)
- [Using HTTPS to access TeamCity server](#)
- [TeamCity Disk Space Watcher](#)
- [TeamCity Server Logs](#)
- [Build Agents Configuration and Maintenance](#)
- [TeamCity Memory Monitor](#)
- [Disk Usage](#)
- [Server Health](#)
- [Build Time Report](#)
- [TeamCity Monitoring and Diagnostics](#)
- [Uploading SSL Certificates](#)

## Configuring Authentication Settings

TeamCity can authenticate users via an internal database, or can integrate into your system and use external authentication sources such as Windows Domain or LDAP.

- [Configuring Authentication](#)
  - [Simple Mode](#)
  - [Advanced Mode](#)
- [User Authentication Settings](#)
  - [Special User Accounts](#)
- [Credentials Authentication Modules](#)
  - [Built-in Authentication](#)
  - [Windows Domain Authentication](#)
  - [LDAP Authentication](#)
- [HTTP Authentication Modules](#)
  - [Basic HTTP Authentication](#)
  - [NTLM HTTP Authentication](#)

### Configuring Authentication

Authentication is configured on the Administration | Authentication page; the currently used authentication modules are also displayed here.

TeamCity provides several preconfigured authentication options (presets) to cover the most common use-case described [below](#). The presets are combinations of authentication modules supported by TeamCity: three credentials authentication modules and two HTTP authentication modules:

- [Credentials Authentication Modules](#)
  - [Built-in](#)
  - [Windows Domain Authentication](#)
  - [LDAP Integration \(separate page\)](#)
- [HTTP Authentication Modules](#)
  - [Basic HTTP Authentication](#)
  - [NTLM HTTP Authentication \(separate page\)](#)

 If you are using [JetBrains Hub](#), you can configure single sign-on (SSO) via JetBrains Hub from TeamCity login form and IDE using a [separate plugin for TeamCity](#).

When you first log in to TeamCity, the default authentication including the Built-in and Basic HTTP Authentication modules is enabled and editing authentication settings in the [simple mode](#) is active.

- To modify the existing settings, click the Edit link in the table next to the description of the enabled authentication module.
- To switch to a different preconfigured scheme, use the Load preset button. For more options, switch to the [Advanced mode](#).

 Any changes made to authentication in the UI will be reflected in the `<TeamCity data directory>/config/auth-config.xml` file, which can be used to configure authentication if editing via the Web UI is not suitable for some reason. The detailed description is available in the [previous documentation version](#).

## Simple Mode

Simple mode (default) allows you to select presets created for the most common use cases. To override the existing authentication settings, use the Load preset... button, select one of the options and Save your changes. The following presets are available:

- Default ([built-in authentication - Basic HTTP](#))
- [LDAP](#)
- Active directory ([LDAP with NTLM](#))
- Microsoft Windows Domain ([Basic HTTP](#) and [NTLM](#))

## Advanced Mode

TeamCity allows enabling several authentication modules simultaneously using the advanced mode in the TeamCity Web UI.

When a user attempts to log in, all the modules will be tried one by one. If one of them authenticates the user, the login will be successful; if all of them fail, the user will not be able to log into TeamCity.

 Since TeamCity 10.0.2, if the System Administrator creates users without password with several authentication modes enabled on the server including the [Built-in](#) one, and later changes authorization from mixed one to the build-in one, users with no password will be unable to log in to TeamCity.

 It is possible to use a combination of internal and external authentication. The recommended approach is to configure [LDAP Integration](#) for your internal employees first and then to add [Built-in](#) authentication for external users.

1. Switch to advanced mode with the corresponding link on the Administration | Authentication page.
2. Click Add Module and select a module from the drop-down.
3. Use the properties available for modules by selecting/deselecting checkboxes in the Add Module dialog.
4. Click Apply and Save your changes.

Also, TeamCity plugins can provide [additional authentication modules](#).

## User Authentication Settings

The very first time TeamCity server starts with no users (and no administrator) so you will be prompted for the administrator account. If you are not prompted for the administrator account, please refer to [How To Retrieve Administrator Password](#) for a resolution.

The TeamCity administrator can modify the authentication settings for every user on their profile page.

The TeamCity list of users and authentication modules just map external credentials to the users. This means that a single TeamCity user can authenticate using different modules, provided the entered credentials are mapped to the same TeamCity user. Authentication modules regularly have a configuration on how to map external user data to a TeamCity user and some (Windows Domain, JetBrains Hub) allow to edit the external user linking data on the TeamCity user profile.

Handling of the user mapping by the bundled authentication modules:

- Built-in authentication stores a TeamCity-maintained password for each user
- Windows Domain authentication allows specifying the default domain and assumes the Domain account name is equal to the TeamCity user. The domain account can be edited on the user profile page
- LDAP Integration allows setting LDAP property to get TeamCity username from user's LDAP entry

Care should be taken when modifying authentication settings: there can be a case when the administrator cannot login after changing authentication modules.

Let's imagine that the administrator had the "jsmith" TeamCity username and used the default authentication. Then the authentication module was changed to Windows domain authentication (i.e. Windows domain authentication module was added and the default one was removed). If, for example, the Windows domain username of that administrator is "john.smith", he/she is not able to login anymore: he/she cannot login using the default authentication since it is disabled, and cannot login using Windows domain authentication since his/her Windows domain username is not equal to TeamCity username. The solution nevertheless is quite simple: the administrator can login using the super user account and change his/her TeamCity username or specify his/her Windows domain username on his/her own profile page.

## Special User Accounts

By default, TeamCity has a [Super User](#) account with maximum permissions and a [Guest User](#) with minimal permissions. These accounts have no personal settings such as the [Changes](#) page and Profile information as they are not related to any particular person but rather intended for special use cases.

## Credentials Authentication Modules

### Built-in Authentication

By default, TeamCity uses the built-in authentication, meaning that users and their passwords are maintained by TeamCity. When logging to TeamCity for the first time, the user will be prompted to create the TeamCity username and password which will be stored in TeamCity and used for authentication. If you installed TeamCity and logged into it, it means that built-in authentication is enabled and all user data is stored in TeamCity.

In the beginning the user database is empty and new users are either [added by the TeamCity administrator](#) or users are self-registered: the default settings allow the users to register from the login page. All newly created users belong to the [All Users](#) group and have all roles assigned to this group. If some specific [roles](#) are needed for the newly registered users, these roles should [be granted](#) via the All Users group.

By default, the users are allowed to change their password on their profile page.

### Windows Domain Authentication

Allows user login using Windows domain name and password.

The credential check is performed on the TeamCity server side, so the server should be aware of the domain(s) users use to log in.

The supported syntax for the username is `DOMAIN\user.name` as well as `<username>@<domain>`.

In addition to logging in using the login form, you can enable [NTLM HTTP Authentication](#) single sign-on.

If you select the "Microsoft Windows Domain" preset, in addition to the login via a Windows domain, the [Basic HTTP](#) and [NTLM](#) authentication modules are enabled by default.

### Specifying Default Domain

To enable users to enter the system using the login form without specifying the domain as a part of the user name, do the following:

1. Go to the Administration| Authentication page.
2. Click the edit link in the table next to the Microsoft Windows domain authentication description.
3. Set the name in the Default domain: field.
4. Click Done and Save your changes.

### Registering New Users on Login

The default settings allow users to register from the login page and TeamCity user names for the new users will be the same as their Windows domain account.

All newly created users belong to the [All Users](#) group and have all roles assigned to this group. If some specific [roles](#) are needed for the newly registered users, these roles should [be granted](#) via the All Users group.

To disable new user registration on login:

1. Go to the Administration| Authentication page.
2. Click the edit link in the table next to the Microsoft Windows domain authentication description. Uncheck the Allow user registration from the login page box.
3. Click the edit link in the table next to the NTLM HTTP authentication description. Uncheck the Allow user registration from the login page box.

## Linux-Specific Configuration

If your TeamCity server runs under Linux, JCIFS library is used for the Windows domain login. This only supports Windows domain servers with SMB (SMBv1) enabled. SMB2 is not supported.

The library is configured using the properties specified in the `<TeamCity data directory>/config/ntlm-config.properties` file. Changes to the file take effect immediately without the server restart.

JCIFS library settings which cannot be changed in run-time or settings to affect HTTP NTLM settings can only be set via a properties file passed via `-Djcifs.properties` JVM option.

If the default settings do not work for your environment, refer to <http://jcifs.samba.org/src/docs/api/> for all available configuration properties.

If the library does not find the domain controller to authenticate against, consider adding the `jcifs.netbios.wins` property to the `ntlm-config.properties` file with the address of your WINS server. For other domain services locating properties, see <http://jcifs.samba.org/src/docs/resolver.html>.

## LDAP Authentication

Please refer to the [corresponding section](#).

## HTTP Authentication Modules

### Basic HTTP Authentication

Please refer to [Accessing Server by HTTP](#) for details about basic HTTP authentication.

 For information on configuring Basic HTTP Authentication directly in the `<TeamCity data directory>/config/auth-config.xml`, refer to the [previous documentation version](#).

### NTLM HTTP Authentication

Please refer to the [corresponding section](#).

See also:

[Concepts: Authentication Modules](#)

## LDAP Integration

LDAP integration in TeamCity has two levels: authentication (login) and users synchronization:

- authentication allows you to login in to TeamCity using LDAP server credentials.
- once LDAP authentication is configured, you can enable LDAP synchronization which allows the TeamCity user-set to be automatically populated with the user data from LDAP.

LDAP integration is generic and can be configured for Active Directory or other LDAP servers.

 It is recommended to configure LDAP authentication on a test server before enabling it in the production one, because switching to an incorrectly configured authentication scheme may cause users' inability to log in to TeamCity.

LDAP integration might be not trivial to configure, so it might require some trial and error approach to get the right settings. Please review [Typical LDAP Configurations](#). If a problem occurs, [LDAP logs](#) should give you enough information to understand possible misconfigurations. If you are experiencing difficulties configuring LDAP integration after going through this document and investigating the logs, please [contact us](#) and let us know your LDAP settings with a detailed description of what you want to achieve and what you currently get.

On this page:

- [Authentication](#)
  - [Ldap-config.properties Configuration](#)
  - [Configuring User Login](#)
    - [Active Directory](#)
  - [Advanced Configuration](#)

- **Synchronization**
  - Common Configuration
  - User Profile Data
  - User Group Membership
  - Creating and Deleting Users
  - Username migration
- **Miscellaneous**
  - Switching to LDAP Authentication
  - Scrambling credentials in `ldap-config.properties` file
  - Debugging LDAP Integration

## Authentication

To allow logging into TeamCity with LDAP credentials, you need to configure LDAP connection settings in the `ldap-config.properties` file and enable LDAP authentication in the server's [Authentication section](#).

If you need to configure authentication without access to web UI refer to the [corresponding section](#) in the previous documentation version.

When the `Allow creating new users on the first login` option is selected (this is the default) a new user account will be created on the first successful login. The TeamCity user names for the new users will be derived from their LDAP data based on the configured setting. All newly created users belong to the `All Users` group and have all roles assigned to this group. If some specific [roles](#) are needed for the newly registered users, these roles can be granted via the `All Users` group.

TeamCity stores user accounts and details in its own database. For information on automatic user creation and automatic population of user details from LDAP, refer to [Synchronization](#) section below.

### `ldap-config.properties` Configuration

LDAP integration settings are configured in the `<TeamCity data directory>/config/ldap-config.properties` file on the server.

Create the file by copying `<TeamCity data directory>/config/ldap-config.properties.dist` file and renaming it to the `<TeamCity data directory>/config/ldap-config.properties`; follow the comments in the file to edit the default settings as required.

The file uses the standard Java properties file syntax, so all the values in the file must be properly [escaped](#). The file is re-read on any modification so you do not need to restart the server to apply changes in the file.

It is strongly recommended to back up the previous version of the file: if you misconfigure LDAP integration, you may no longer be able to log in into TeamCity. The users who are already logged in are not affected by the modified LDAP integration settings because users are authenticated only on login.

The mandatory property in the `ldap-config.properties` file is `java.naming.provider.url` that configures the server and root DN. The property stores the URL to the LDAP server node that is used in following LDAP queries. For example, `ldap://dc.example.com:389/CN=Users,DC=Example,DC=Com`. Please note that the value of the property should use URL-escaping if necessary, e.g. use `%20` if you need the space character.



For samples of `ldap-config.properties` file please refer to the [Typical LDAP Configurations](#) page.

The supported configuration properties are documented in comments in the `ldap-config.properties.dist` file.

## Configuring User Login

The general login sequence is as follows:

- based on the username entered in the login form by the user, an LDAP search is performed (defined by the `teamcity.users.login.filter` LDAP filter where the user-entered username is referenced via the `$login$` or `$capturedLogin$` substring within the users base LDAP node (defined by `teamcity.users.base`),
- if the search is successful, authentication (LDAP bind) is performed using the DN found during the search and the user-entered password,
- if the authentication is successful, TeamCity user is created if necessary and the user is logged in. The name of the TeamCity user is retrieved from an attribute of the found LDAP entry (the attribute name is defined via the `teamcity.users.username` property)

When users log in via LDAP, TeamCity does not store the user passwords. On each user login, authentication is performed by a direct login into LDAP with the credentials based on the values entered in the login form.

Please note that in certain configurations (for example, with `java.naming.security.authentication=simple`) the login information will be sent to the LDAP server in the unencrypted form. For securing the connection, refer to [Sun documentation](#).

Another option is to configure communications via the ldaps protocol.

The related external link: [How To Set Up Secure LDAP Authentication with TeamCity](#) by Alexander Groß.

#### Active Directory

The following template enables authentication against active directory:

Add the following code to the `<TeamCity Data Directory>/config/ldap-config.properties` file (assuming the domain name is "Example.Com" and domain controller is "dc.example.com").

```
java.naming.provider.url=ldap://dc.example.com:389/DC=Example,DC=Com  
java.naming.security.principal=<username>  
java.naming.security.credentials=<password>  
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)  
teamcity.users.username=sAMAccountName  
java.naming.security.authentication=simple  
java.naming.referral=follow
```

#### Advanced Configuration

If you need to fine-tune LDAP connection settings, you can add the `java.naming` options to the `ldap-config.properties` file: they will be passed to the underlying Java library. The default options are retrieved using `java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory`. Refer to the Java [documentation page](#) for more information about property names and values.

You can use an LDAP explorer to browse LDAP directory and verify the settings (for example, <http://www.jxplorer.org/> or <http://www.ldapbrowser.com/softerra-ldap-browser.htm>).

There is an ability to specify failover servers using the following pattern:

```
java.naming.provider.url=ldap://ldap.mycompany.com:389 ldap://ldap2.mycompany.com:389  
ldap://ldap3.mycompany.com:389
```

The servers are contacted until any of them responds. There is no particular order in which the address list is processed.

#### Synchronization

Synchronization with LDAP in TeamCity allows you to:

- Retrieve the user's profile data from LDAP
- Update the user groups membership based on LDAP groups
- Automatically create and remove users in TeamCity based on information retrieved from LDAP

TeamCity supports one-way synchronization with LDAP: the data is retrieved from LDAP and stored in the TeamCity database. Periodically, TeamCity fetches data from LDAP and updates users in TeamCity.

When synchronization is enabled, you can review the related data and run on-demand synchronization in the Administration | LDAP Synchronization section of the [server settings](#).

#### Common Configuration

You need to have LDAP authentication configured for the synchronization to function.

By default, the synchronization is turned off. To turn it on, add the following option to `ldap-config.properties` file:

```
teamcity.options.users.synchronize=true
```

You also need to specify the following mandatory properties:

- `java.naming.security.principal` and `java.naming.security.credentials` - they specify the user credentials which are used by TeamCity to connect to LDAP and retrieve data,
- `teamcity.users.base` and `teamcity.users.filter` - these specify the settings to search for users

- `teamcity.users.username` - the name of the LDAP attribute containing the TeamCity user's username. Based on this setting LDAP entries are mapped to the TeamCity users.

No users are created or deleted on enabling user's synchronization. Check the section [Creating and Deleting Users](#) for the related configuration.

#### User Profile Data

When synchronization is properly configured, TeamCity can retrieve user-related information from LDAP (e-mail, full name, or any custom property) and store it as TeamCity user's details. If updated in LDAP, the data will be updated in the user's profile in TeamCity. If modified in user's profile in TeamCity, the data will no longer be updated from LDAP for the modified fields. All the user fields synchronization properties store the name of LDAP field to retrieve the information from.

The user's profile synchronization is performed on user creation and also periodically for all users.

The list of supported user settings:

- `teamcity.users.username`
- `teamcity.users.property.displayName`
- `teamcity.users.property.email`
- `teamcity.users.property.plugin:notificator:jabber:jabber-account`
- `teamcity.users.property.plugin:vcs:<VCS type>:anyVcsRoot` — VCS username for all `<VCS type>` roots. The following VCS types are supported: svn, perforce, jetbrains.git, cvs, tfs, vss, clearcase, starteam.

Example properties can be seen by configuring them for a user in the web UI and then listing the properties via [REST API](#).

Since TeamCity 8.0, there is an experimental feature which allows mapping user profile properties in TeamCity to a formatted combination of LDAP properties rather than to a specific property on user synchronization.

To enable the mapping, add `teamcity.users.properties.resolve=true` into `ldap-config.properties`.

Then you can use the %-references to LDAP attributes in the form of `%ldap.userEntry.<attribute>%` in the user property definitions, e.g.

```
teamcity.users.property.plugin\:notificator\:jabber\:jabber-account=%ldap.userEntry.name%@jabber
.my.domain.com
```

#### User Group Membership

TeamCity can automatically update users membership in groups based on the LDAP-provided data.

To configure Group membership:

1. Create groups in TeamCity manually.
2. Specify the mapping of LDAP groups to TeamCity groups in the `<TeamCity data directory>/config/ldap-mapping.xml` file. Use the `ldap-mapping.xml.dist` file as an example: TeamCity user groups are determined by the [Group Key](#), LDAP groups are specified by the group DN.
3. Set the required properties in the `ldap-config.properties` file, the groups settings section:
  - `teamcity.options.groups.synchronize` - enables user group membership synchronization
  - `teamcity.groups.base` and `teamcity.groups.filter` - specifies the LDAP base node and filter to find the groups in LDAP (those configured in the `ldap-mapping.xml` file should be a subset of the groups found by these settings)
  - `teamcity.groups.property.member` specifies the LDAP attribute holding the members of the group.

Note that TeamCity openly operates with the users matched by `teamcity.users.base` and `teamcity.users.filter` settings, so only the users found by these properties will be processed during the group membership synchronization process.

On each synchronization run, TeamCity updates the membership of users in the groups configured in the mapping. By default, TeamCity synchronizes membership only for users residing directly in the groups.

To map nested LDAP groups to TeamCity:

- either (since TeamCity 2017.1) specify `teamcity.groups.retrieveUsersFromNestedGroups=true` in `ldap-config.properties` file and make sure all the groups hierarchy is matched by the `teamcity.groups.base' / 'teamcity.groups.filter` settings
- or copy your LDAP groups structure in TeamCity groups, together with the group inclusions. Then configure the mapping between the TeamCity groups and corresponding LDAP groups.

If either an LDAP group or a TeamCity group that is configured in the mapping is not found, an error is reported. You can review the errors found during the last synchronization run in the Administration | LDAP Synchronization section of the [server settings](#).

See also [example settings](#) for Active Directory synchronization.

#### Creating and Deleting Users

TeamCity can automatically create users in TeamCity, if they are found in one of the mapped LDAP groups and groups synchronization is turned on via `teamcity.options.groups.synchronize` option.

By default, automatic user creation is turned off. To turn it on, set `teamcity.options.createUsers` property to `true` in `ldap-config.properties` file.

TeamCity can automatically delete users in TeamCity if they cannot be found in LDAP or do not belong to an LDAP group that is mapped to predefined "All Users" group. By default, automatic user deletion is turned off as well; set `teamcity.options.deleteUsers` property to turn it on.

#### Username migration

The username for the existing users can be updated upon first successful login. For instance, suppose the user previously logged in using 'DOMAIN\user' name, thus the string 'DOMAIN\user' was stored in TeamCity as the username. To synchronize the data with LDAP, the user can change the username to 'user' using the following options:

```
teamcity.users.login.capture=DOMAIN\\(.*)  
teamcity.users.login.filter=(cn=$login$)  
teamcity.users.previousUsername=DOMAIN\\$login$
```

The first property allows you to capture the username from the input login and use it to authenticate the user (can be particularly useful when the domain 'DOMAIN' isn't stored anywhere in LDAP). The second property `teamcity.users.login.filter` allows you to fetch the username from LDAP by specifying the search filter to find this user (other mandatory properties to use this feature: `teamcity.users.base` and `teamcity.users.username`). The third property allows you to find the 'DOMAIN\user' username when login with just 'user', and replace it with either the captured login, or with the username from LDAP.

Note that if any of these properties are not set or cannot be applied, the username isn't changed (the input login name is used). More configuration examples are available [here](#).

#### Miscellaneous

##### Switching to LDAP Authentication

When you add the LDAP authentication module on a TeamCity server which already has users, the users can still log in using the previous authentication modules credentials (unless you remove the modules). On a successful LDAP login, LDAP retrieves a username as configured by the `teamcity.users.username` property and if there is already a user with such TeamCity username, user logs in as the matching user. If there is no existing user, a new one is created with the username retrieved from LDAP.

##### Scrambling credentials in `ldap-config.properties` file

Since TeamCity 9.0 `java.naming.security.credentials` property can be used to configure either plain-text or scrambled-form passwords. To get the scrambled password value, execute the following command (must be executed from <TeamCity installation directory>/webapps/ROOT/WEB-INF/lib directory):

For Windows:

```
java -cp serviceMessages.jar:common-api.jar:commons-codec.jar:commons-codec-1.3.jar:log4j-1.2.12.jar  
jetbrains.buildServer.serverSide.crypt.ScrambleMain "<text to scramble>"
```

For Linux:

```
java -cp serviceMessages.jar:common-api.jar:commons-codec.jar:commons-codec-1.3.jar:log4j-1.2.12.jar  
jetbrains.buildServer.serverSide.crypt.ScrambleMain "<text to scramble>"
```

Note that scrambling is not encryption: it protects the password from being easily remembered when seen occasionally, but it does not protect against getting the real password value when someone gets the scrambled password value.

##### Debugging LDAP Integration

Internal LDAP logs are stored in `logs/teamcity-ldap.log*` files in the server logs. If you encounter an issue with LDAP configuration, it is advised that you look into the logs as the issue can often be figured out from the messages in there. To get detailed logs of LDAP login and synchronization processes, use the "debug-ldap" [logging preset](#).

If you want to [report an LDAP issue to us](#), make sure to include LDAP settings (`<TeamCity data directory>/config/ldap-config.properties` and `ldap-mapping.xml` files, with password in `ldap-config.properties` masked) and debug LDAP logs fully covering a login/synchronization sequence (include all `teamcity-ldap.log*` files). Please make sure to describe the related structure of your LDAP (noting the attributes of related LDAP entities) and detail expected/actual behavior. The archive with the data can be sent to us via [one of the supported ways](#).

## Typical LDAP Configurations

This page contains samples of `ldap-config.properties` file for different configuration cases.

- Basic LDAP Login
  - Windows Active Directory
  - Unix
  - Specifying Backup LDAP server
- Basic LDAP Login for Users in Specific LDAP Group Only
- Active Directory With User Details Synchronization
- Active Directory With User Details Synchronization and User Creation
- Active Directory With Group Synchronization
  - Limiting the number of groups to be synchronized

### Basic LDAP Login

The examples of minimal working configurations are given below.

Windows Active Directory

```
java.naming.provider.url=ldap://dc.example.com:389/DC=example,DC=com
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName
```

Note that "sAMAccountName" is limited to 20 symbols. You might want to use another attribute which contains entire username.

Unix

```
java.naming.provider.url=ldap://dc.example.com:389/DC=example,DC=com
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
teamcity.users.login.filter=(uid=$capturedLogin$)
teamcity.users.username=uid
```

TeamCity does not store the user passwords in this case.

On each user login, authentication is performed by a direct login into LDAP with the credentials entered in the login form.  
Specifying Backup LDAP server

You can specify a backup LDAP server in the `java.naming.provider.url` property as follows:

```
# The second URL is used when the first server is down.
java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
ldap://failover.example.com:389/DC=example,DC=com
```

#### Basic LDAP Login for Users in Specific LDAP Group Only

Only users from a specific user group are allowed to log in. The users need to enter the username only the without domain part to log in. The example is for Windows Active Directory:

```
java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
java.naming.security.principal=<username>
java.naming.security.credentials=<password>

# filtering only users with specified name and belonging to LDAP group "Group1" with DN
"CN=Group1,CN=Users,DC=example,DC=com"
teamcity.users.login.filter=(&(sAMAccountName=$capturedLogin$)(memberOf=CN=Group1,CN=Users,DC=example,DC=com))

#teamcity.users.username=sAMAccountName

# Allow only username part without domain (optional)
teamcity.auth.loginFilter=[^/\\@]+

# No synchronization, just login.
teamcity.options.users.synchronize=false
teamcity.options.groups.synchronize=false
```

#### Active Directory With User Details Synchronization

Users can log in to TeamCity with their domain name without the domain part, there is an account "teamcity" with password "secret" that can read all Active Directory entries. The TeamCity user display name and email are synchronized from Active Directory.

```

java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
java.naming.security.principal=CN=teamcity,CN=Users,DC=example,DC=com
java.naming.security.credentials=secret
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName

# User synchronization: on, synchronize display name and e-mail.
teamcity.options.users.synchronize=true
teamcity.users.filter=(objectClass=user)
teamcity.users.property.displayName=displayName
teamcity.users.property.email=mail

```

#### Active Directory With User Details Synchronization and User Creation

Users can log in to TeamCity with their domain name without the domain part, there is an account "teamcity" with password "secret" that can read all Active Directory entries. The TeamCity user display name and email are synchronized from Active Directory.

The users not existing in the TeamCity database are created.

Users no longer existing in Active Directory are deleted from the TeamCity user database.

```

java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
java.naming.security.principal=CN=teamcity,CN=Users,DC=example,DC=com
java.naming.security.credentials=secret
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName

# User synchronization: on, synchronize display name and e-mail.
teamcity.options.users.synchronize=true
teamcity.users.filter=(objectClass=user)
teamcity.users.property.displayName=displayName
teamcity.users.property.email=mail

# Automatic user creation and deletion during user synchronization
teamcity.options.createUsers=true
teamcity.options.deleteUsers=true

```

#### Active Directory With Group Synchronization

There should be `ldap-mapping.xml` file with one or more group mappings defined.

`ldap-config.properties` file:

```

java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
java.naming.security.principal=CN=teamcity,CN=Users,DC=example,DC=com
java.naming.security.credentials=secret
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName

# User synchronization is on, synchronize display name and e-mail.
teamcity.options.users.synchronize=true
teamcity.users.filter=(objectClass=user)
teamcity.users.property.displayName=displayName
teamcity.users.property.email=mail

# Automatic user creation and deletion during users synchronization
teamcity.options.createUsers=true
teamcity.options.deleteUsers=true

# Groups synchronization is on
teamcity.options.groups.synchronize=true

# The group search LDAP filter used to retrieve groups to synchronize.
# The result includes all the groups configured in the ldap-mapping.xml file.

teamcity.groups.filter=(objectClass=group)

# The LDAP attribute of a group storing its members.
teamcity.groups.property.member=member

```

#### Limits the number of groups to be synchronized

The `teamcity.users.filter` property helps limit the number of processed user accounts during users synchronization.

It is recommended to create the "TeamCity Users" group in Active Directory, and include all your required groups into this group, e.g. you may have the following Active Directory structure:

- Group A with members User 1, User 2
- Group B with members User 3, User 4
- Group "TeamCity Users" with members Group A, Group B

Then update the `teamcity.users.filter` property, e.g.

```
teamcity.users.filter=(&(objectClass=user)(memberOf:1.2.840.113556.1.4.1941:=CN=TeamCity
Users,OU=Accounts,DC=domain,DC=com))
```

In this case TeamCity creates accounts only if they are members of the corresponding Active Directory group. Nested groups are supported.

Alternatively, you can list several groups:

```
teamcity.users.filter=(&(objectClass=user)(|(memberOf=CN=GroupOne,OU=myou,DC=company,DC=
tld)(memberOf=CN=GroupTwo,OU=myou,DC=company,DC=tld)))
```

To limit users who can login into TeamCity you also need to change `teamcity.users.login.filter` property:

```
teamcity.users.login.filter=(&(sAMAccountName=$capturedLogin$)(memberOf:1.2.840.113556.1.4.19  
41:=CN=TeamCity Users,OU=Accounts,DC=domain,DC=com))
```

For more details on the filter syntax refer to the [Microsoft documentation](#).  
For more details on the AD attributes refer to the [Microsoft documentation](#).

## LDAP Troubleshooting

General advice: if you experience problems with LDAP configuration, turn on the debug logging (see [Reporting Issues](#)).

### Cannot authenticate using LDAP

Check the `teamcity-ldap.log` file. For each unsuccessful login attempt there should be a reason specified. Most commonly these are:

- The login filter doesn't match the entered login ("User-entered login does not match `teamcity.auth.loginFilter=..., aborting`")
- The LDAP server rejected login with the "Invalid credentials" message ("Failed to login user '....' due to authentication error. Cause: Invalid credentials ([LDAP: error code 49 - 80090308: LdapErr: DSID-0C090334, comment: AcceptSecurityContext error, data 525, vece^@])")

The first reason means that the login can't be used for signing in because it doesn't match a certain filter. For example, by default you can't login with 'DOMAIN\username' - the filter forbids '/', '\' and '@' symbols. See the `teamcity.auth.loginFilter` property.

The second error can be caused by various things, e.g.:

- You are trying to login with your username, but LDAP server accepts only full DNs  
If all users are stored in one LDAP branch, you should use the `teamcity.auth.formatDN` property. Otherwise see the section below.
- Check your DN and the actual principal from the logs, probably there is a typo or an unescaped sequence. Try to log in with this principal using another LDAP tool.
- Try changing the security level (`java.naming.security.authentication`): it can be "simple", "strong" or "none".

Users in LDAP are stored in different branches, so the `teamcity.auth.formatDN` property can't be applied. How can the users login with their usernames?

This feature is available from version 5.0. You should specify how you want to find the user (`teamcity.users.login.filter`), e.g. by the username or e-mail. On each login TeamCity finds the user in LDAP before logging in, fetches the user DN and then performs the bind. Thus you should also define the credentials for TeamCity to perform search operations (`java.naming.security.principal` and `java.naming.security.credentials`).

### NTLM HTTP Authentication

The TeamCity NTLM HTTP authentication feature employs Integrated Windows Authentication and allows transparent/SSO login to the TeamCity web UI when using browsers/clients supporting NTLM and Negotiate HTTP authentications with NTLMv1, NTLMv1, NTLMv2 and Kerberos logic. The authentication is supported when the TeamCity server runs under Windows OS.

Generally, it allows users to log in to the TeamCity server web UI using their NT domain account without the need to enter credentials manually.

- Configuration
- Requirements
- Enabling NTLM HTTP Authentication
  - NTLM login URLs
- Using NTLM HTTP Authentication Module with LDAP Authentication
- Configuring client
  - Internet Explorer
  - Google Chrome
  - Mozilla Firefox
- Troubleshooting

## Configuration

The NTLM HTTP module is configured on the Administration | Authentication page under the "HTTP authentication modules" section.

 For information on configuring authentication the settings directly in the <TeamCity data directory>/config/auth-config.xml, refer to the [previous documentation version](#).

You can enable NTLM login with any login module once the TeamCity username is the same as the Windows domain username or the Windows domain username is specified on the user profile.

 NTLM HTTP authentication is supported only for TeamCity servers installed on Windows machines. If you need it under other platforms, feel free to [contact us](#) with details as to why.

## Requirements

1. The authenticating user should be logged in to the client workstation with the domain account that is to be used for the authentication.
2. The user's web browser should support NTLM HTTP authentication (the TeamCity server URL should be a "trusted site", etc.)

## Enabling NTLM HTTP Authentication

After the NTLM HTTP authentication module is configured, users will see a link on the login screen which, when clicked, will force the browser to send the domain authentication data.

You can force the server to announce NTLM HTTP authentication by specifying protocols in the "Force protocols" setting. This will make the server request domain authentication for any request to the TeamCity web UI. If the user's browser is run in the domain environment, the current user will be logged in automatically. If not, the browser will pop up a dialog asking for domain credentials.

Without this attribute, NTLM HTTP authentication will work only if the client explicitly initiates it (e.g. clicks the "Login using NT domain account" link on the login page), and in the usual case an unauthenticated user will be simply redirected to the TeamCity login page. The TeamCity server forces NTLM HTTP authentication only for Windows users by default. If you want to enable it for all users, set the following [internal property](#):

```
teamcity.ntlm.ignore.user.agent=true
```

## NTLM login URLs

There are two more ways to force NTLM authentication for a certain connection (there is no need to set the `forceProtocols` attribute for this case):

- Send request to <Your TeamCity server URL>/ntlmLogin.html and TeamCity will initiate NTLM authentication and redirect you to the overview page.
- Send request to <Your TeamCity server URL>/ntlmAuth/<path> and TeamCity will initiate NTLM authentication and show you the <path> page (without redirect).

## Using NTLM HTTP Authentication Module with LDAP Authentication

When using LDAP authentication, it is possible to deny login for some users. The NTLM HTTP authentication module (as well as the Windows domain credentials authentication module) does not have such functionality, so it can be possible for some users to log in using Windows domain account even if they are not allowed to log in via LDAP. To solve this problem, you should enable the `Allow creating new users on the first login` option for the corresponding authentication module.

With this property set, a user will be able to log in via their NT domain account only if he/she already has an existing account in TeamCity (i.e. if he/she has already logged into TeamCity earlier via LDAP) with a TeamCity username which equals the Windows domain username or a custom NT domain username specified on the user's profile page.

## Configuring client

Depending on your environment, you may need to configure your client to make NTLM authentication work.

### Internet Explorer

1. Open Tools | Internet Options.
2. On the Advanced tab make sure the option Security | Enable Integrated Windows Authentication is checked.
3. On the Security tab select Local Intranet | Sites | Advanced and add your TeamCity server URL to the list.

### Google Chrome

On Windows, Chrome normally uses IE's behaviour, see more information [here](#).

### Mozilla Firefox

1. Type `about:config` in the browser's address bar.
2. Add your TeamCity server URL to the `network.automatic-ntlm-auth.trusted-uris` property.

## Troubleshooting

Helpful links:

- <http://waffle.codeplex.com/wikipage?title=Frequently%20Asked%20Questions>
- <http://waffle.codeplex.com/discussions/254748>
- <http://waffle.codeplex.com/wikipage?title=Troubleshooting%20Negotiate&referringTitle=Documentation>

## Enabling Guest Login

Logging in as a [guest user](#) is turned off by default.

To enable the guest login to TeamCity:

1. Navigate to the Administration | Authentication page.
2. Select the Allow login as guest user option.
3. Save your changes.

The Log in as guest link appears on the Log in to TeamCity page.

By default, the guest user can view all the projects. To customize which projects guest user has access to:

Click the Configure guest user roles link to configure the [roles](#). The link appears when the [per-project authorization mode](#) is enabled.

See also:

Concepts: [User Account | Role and Permission](#)

Administrator's Guide: [Configuring Authentication Settings | Managing Users and User Groups](#)

## Enabling Email Verification

Since TeamCity 10.0, TeamCity administrators can enable / disable email verification (off by default).

If email verification is enabled on the TeamCity server, the Email address field in the user account registration form becomes mandatory. When an address is added / modified, the users will be asked to verify their address. If the email address is not verified, TeamCity will display a notification on the General tab of the [user account](#) settings.

Verified email addresses will be marked with a green check on the Administration | Users page.

To enable email verification in TeamCity:

1. Navigate to the Administration | Authentication page.

2. Select the Enable email verification option (off by default) .
3. Save your changes.

During the [project import](#) TeamCity will take verified emails into account. If there are two users with different usernames, but the same verified email, TeamCity will provide a possibility to merge these users.

See also:

[Concepts: User Account | Role and Permission](#)

[Administrator's Guide: Configuring Authentication Settings | Managing Users and User Groups](#)

## TeamCity Data Backup

TeamCity provides several ways to back up its data:

- **Backup from the Web UI:** an action in the web UI (can also be triggered via REST API) to create a backup while the server is running. It is recommended for regular maintenance backups. Some limitations on the backed up data apply (see the [related section](#) below). This option is also available on upgrade in the maintenance screen - on the first start of a newer version of the TeamCity server.
- **Backup via the `maintainDB` command-line tool:** Same as via the UI. To include all data, use the tool when the server is stopped.
- **Manual backup:** is suitable if you want to manage the backup procedure manually.
- You may need to back up the build agent's data only.

Restoring data from backup is performed using the `maintainDB` tool.



We strongly urge you to make the backup of TeamCity data before upgrading. Note that TeamCity server does not support downgrading.

### Backing up Data

TeamCity allows backing up the following data:

- **Data stored in the database**
- Server settings, settings of projects and builds configurations (everything stored in <TeamCity Data Directory>/config)
- Custom plugins (installed under <TeamCity Data Directory>/plugins) and database drivers (from <TeamCity Data Directory>/lib directory)
- Supplementary data: settings history, triggers states, plugins data, etc. (everything under <TeamCity Data Directory>/system/pluginData directory)
- Build logs
- Personal changes

The data to be backed up can be configured using the backup scope options when using the TeamCity Web UI or via additional parameters for the `maintainDB` command-line tool.

The following data is not included into backup:

- build artifacts (because of their size). These include explicit build artifacts and internal artifacts storing coverage report, finish build parameters, settings digest, etc. If you need the build artifacts, please also backup content of `artifacts` directories manually.
- for backup taken from UI: running builds and build queue state. If you want to backup these, use the command line `maintainDB` tool while the TeamCity server is not running.
- TeamCity application manual customizations under <TeamCity Home>, including used server port number which are stored in <TeamCity Home>/conf/server.xml file.
- TeamCity application logs (they also reside under <TeamCity Home>/logs)
- Any manually created files under <TeamCity Data Directory> that do not fall into previously mentioned items.

The recommended approach is either to perform the backup process described under [Manual Backup and Restore](#) or run a backup from the [web UI](#) regularly (e.g. automated via REST API) with the "Basic" level - this will ensure backing up all important data except build artifacts and build logs.

Build artifacts and logs (if necessary) can be backed up manually by copying files under `.BuildServer/system/artifacts` and `.BuildServer/system/messages`. See [TeamCity Data Directory#artifacts](#) for details. If logs are selected for backup, TeamCity will search for them in `all artifact` directories currently specified on the server.

Note that for large production TeamCity installations, exporting and importing of data from/to the database may not be an optimal solution and maintaining database backup via replication might be a better option; e.g. see the corresponding documentation for MySQL database.

See also:

[Installation and Upgrade: Upgrade](#)

## Creating Backup from TeamCity Web UI

TeamCity allows creating a backup of TeamCity data via the Web UI.

To create a backup file, navigate to the Administration | Backup page, specify backup parameters as described below, and start the backup process.

Option	Description
Backup file	<p>Specify the name for the backup file, the extension (.zip) will be added automatically. By default, TeamCity will store the backup file in the &lt;TeamCity Data Directory&gt;/backup folder. For security reasons you cannot explicitly change this path in the UI. To modify this setting, specify an absolute or relative path (the path should be relative to TeamCity Data Directory) in the &lt;TeamCity Data Directory&gt;/config/backup-config.xml file. For example:</p> <div style="border: 1px solid #ccc; padding: 10px;"><pre>&lt;backup-settings&gt; ... &lt;general&gt;   &lt;backup-dir path="C:/TC-Backups"/&gt; &lt;/general&gt; ... &lt;/backup-settings&gt;</pre></div>
add timestamp suffix	<p>Check this option to automatically add time stamp suffix to the specified filename. This may be useful to differentiate your backup files, if you don't clean up old backups.</p> <div style="border: 1px solid #ccc; padding: 10px;"><ul style="list-style-type: none"><li>• If the directory where backup files are stored already contains a file with the name specified above, TeamCity won't run backup - you will need either to specify another name, or enable time stamp suffix option, which allows to avoid this.</li><li>• Time stamp suffix has specific format: sorting backup files alphabetically will also sort them chronologically.</li></ul></div>
Backup scope	<p>Specify what kind of data you want to back up. The contents of the backup file depending on the scope is described right in the UI when you select a scope. Note that the size of the backup file and the time the backup process will take depends on the scope you select.</p> <p>To reduce the resulting file size and the time spent on the backup, select the "basic" scope, which includes server settings, projects and builds configurations, plugins and database. However, you'll be able to restore only the settings which were backed up.</p> <p>For the full backup suitable for most of the needs, it is recommended to use the Custom scope with all the items selected except for "build logs" and then backup the &lt;TeamCity Data Directory&gt;/system/artifacts location as a usual file system.</p> <p>Build artifacts are not included into the backup due to their size. It is recommended to either backup the <a href="#">artifacts directories</a> separately or use a redundant storage for the artifacts. Build logs are stored as a part of build artifacts, so there is no need to backup the build logs if you implement a separate backup of the artifacts locations.</p>

When you start backup, TeamCity will display its status and details of the current process including progress and estimates.



### Important notes

- Running and queued builds are not included into a backup created during server running. To include these builds, consider using the different [backup](#) approach when the server is not running.
- The backup process takes time that depends on how many builds there are in the system. During this process the system's state can change, e.g. some builds may finish, other builds that were waiting in the build queue may start, new builds may appear in the build queue, etc. Note, that these changes will not influence the backup. TeamCity will backup only the data actual by the time the backup process was started.

- The resulting backup file is a `*.zip` archive which has a specific structure that does not depend on the OS or database type you use. Thus, you can use the backup file to restore your data even on a different Operating System, or with a different database. If you change the contents of this file manually, TeamCity will not be able to restore your data.

## Backup History

The History tab of the Administration | Backup page allows reviewing the list of created backup files, their size and date when the files were created.

 Note that only backup files created from web UI are shown here. Backups created with the `maintainDB` utility are not displayed on the History tab.

See also:

[Installation and Upgrade: Upgrade](#)  
[Concepts: TeamCity Data Directory](#)  
[Administrator's Guide: TeamCity Data Backup](#)

## Creating Backup via maintainDB command-line tool

TeamCity `.tar.gz` and `.exe` distributions provide the `maintainDB.bat|sh` utility located in the `<TeamCity Home>/bin` directory. This command-line tool enables you to back up the server data, [restore it](#), and [migrate between different databases](#). You can also create data backup using web UI.

On this page:

- [Before You Backup](#)
- [Backup File Location and Format](#)
- [Performing TeamCity Data Backup with maintainDB Utility](#)
  - [Configuring backup scope](#)
  - [maintainDB Usage Examples for Data Backup](#)
- [maintainDB Startup Options](#)

### Before You Backup

Before backing up data, it is recommended to shut down the TeamCity server to include all builds into the backup. If the backup process is started when the TeamCity server is up, running and queued builds are not included into the backup.

### Backup File Location and Format

The default directory for backup files is the `<TeamCity Data Directory>\backup`.

 If not specified otherwise with the `-A` option, TeamCity will read the TeamCity Data Directory path from the `TEAMCITY_DATA_PATH` environment variable, or the default path (`$HOME\.Buildserver`) will be used.

The default format of the backup file name is `TeamCity_Backup_<timestamp>.zip`; the `<timestamp>` suffix is added in the '`YY YYMMDD_HHMMSS`' format.

### Performing TeamCity Data Backup with maintainDB Utility

This section describes some of the `maintainDB` options. For a complete list of all available options, run `maintainDB` from the command line with no parameters.

To create data backup file, from the command line start `maintainDB` utility with the `backup` command:

```
maintainDB.[cmd|sh] backup
```

TeamCity data backup has [some limitations](#). By default, if you run `maintainDB` utility with no optional parameters, only the database, server settings, projects and builds configurations, plugins and supplementary data (settings history, triggers states, plugins data, etc.) will be backed up, omitting build logs and personal changes.

#### Configuring backup scope

To configure the scope of data to include in the backup file, use the following options:

- `-C` or `--include-config` — includes build configurations settings
- `-D` or `--include-database` — includes database
- `-L` or `--include-build-logs` — includes build logs
- `-P` or `--include-personal-changes` — includes personal changes
- `-U` or `--include-supplementary-data` — includes supplementary (plugins') data

Specifying different combinations of the above options, you can control the content of the backup file. For example, to create backup with all supported types of data, run

```
maintainDB backup -C -D -L -P -U
```

#### maintainDB Usage Examples for Data Backup

To create backup file with custom name, run `maintainDB` with `-F` or `--backup-file` option and specify desired backup file name without extension:

```
maintainDB.cmd backup -F <backup file custom name>  
or  
maintainDB.cmd backup --backup-file <backup file custom name>
```

Executing the command above will create a new zip-file with the specified name in the default backup directory.

To add the timestamp suffix to a custom filename, add `-M` or `--timestamp` option:

```
maintainDB.cmd backup -F <backup file custom name> -M  
or  
maintainDB.cmd backup -F <backup file custom name> --timestamp
```

To create the backup file in a custom directory, run `maintainDB` with the `-F` option:

```
maintainDB backup -F <absolute path to the custom backup directory>  
or  
maintainDB backup --data-dir <absolute path to the custom backup directory>
```

#### maintainDB Startup Options

If you customize TeamCity server startup options via `TEAMCITY_SERVER_OPTS`/`TEAMCITY_SERVER_MEM_OPTS` environment variables or use custom JDK installation to run the server, you might need to run `maintainDB` script with related options added into `TEAMCITY_MAINTAINDB_OPTS`/`TEAMCITY_MAINTAINDB_MEM_OPTS` environment variables and run the script with all the same environment as the TeamCity server, so that the same JVM is used.

See also:

[Installation and Upgrade: Setting up an External Database | Migrating to an External Database](#)

## Manual Backup and Restore

On this page:

- [Server Manual Backup](#)
  - [TeamCity Data Directory](#)
  - [Database Data](#)
  - [Application Files](#)
  - [Log files](#)
- [Manual Restoration of Server Backup](#)
  - [TeamCity Data Directory Restoration](#)
  - [TeamCity Database Restoration](#)
  - [Restoration to New Server](#)
  - [Restoring Build Logs](#)

### Server Manual Backup

Other ways to create a backup are [available](#). You can use the instructions on this page if you want fine-grained control over the backup process or need to use a specific procedure for your TeamCity backups. Manual backups can also be the most performant way to create and restore backups as they can be tailored to the specific infrastructure in use.



Before performing the backup procedures, it is recommended to stop the TeamCity server. This way you ensure the backup taken is consistent. If you create a backup while the TeamCity server is running, currently updated data (mostly related to queued and running builds as well as settings updates) can appear in inconsistent state after restore. When performing backup while the server is running first create the database backup and then that of the files in the TeamCity Data Directory.

The following data needs to be backed up:

#### TeamCity Data Directory

TeamCity Data Directory stores:

- server settings, projects and build configurations with their settings (i.e. all that is configured via the Administration web UI)
- build artifacts by default, unless a different location is [configured](#). Build logs are also stored as a part of the build artifacts
- current operation files, internal data structure, etc.

For more details on the directory structure and data, refer to the [TeamCity Data Directory](#) section.

If necessary, you can exclude parts of the directory from the backup to save space — you will lose only the excluded data. You may safely exclude the "system/caches" directory from the backup — the necessary data will be rebuilt from scratch on TeamCity startup.

If you decide to skip the backup of data under `<TeamCity Data Directory>/system` directory, make sure you note the most recent files in each of the artifacts, messages and changes subdirectories and save this information. It will be needed if you decide to restore the database backup with the TeamCity Data Directory corresponding to a newer state than that of the database.

The `<TeamCity Data Directory>/system/buildserver.*` files store the internal database (HSQLDB) data. You need to back them up if you use HSQLDB (the default setting not suitable for production use).

#### Database Data

The database stores all information on the build results (build history and all the build-associated data except for artifacts and build logs), VCS changes, agents, build queue, user accounts and user permissions, etc.

- If you use the HSQLDB, the internal database (default setting, not recommended for production), the database is stored in the files residing directly in the `<TeamCity Data Directory>/system` folder. All files from the directory can be backed up. You may also refer to the [HSQLDB backup notes](#).
- If you use an external database, back up your database schema used by TeamCity with database-specific tools. For the external database connection settings used by TeamCity, refer to the `<TeamCity Data Directory>/config/database.properties` file. You can also see the [corresponding installation section](#).

#### Application Files

You do not need to back up TeamCity application directory (web server alone with the web application), provided you still have the original distribution package and you did not:

- place any custom libraries for TeamCity to use
- install any non-default TeamCity plugins directly into web application files
- make any startup script/configuration changes.

If you feel you need to back up the application files:

- If you use a non-war distribution: back up everything under `<TeamCity home directory>` except for the `temp` and `work` directories.
- If you use the war distribution, follow the backup procedure of the servlet container used.

#### Log files

If you need [TeamCity server log files](#) (which are mainly used for problem solving or debug purposes), back up the `<TeamCity Home>/logs` directory.



You may also want to back up TeamCity Windows Service settings, if they were modified.

#### Manual Restoration of Server Backup

If you need to restore backup created with the web UI or `maintainDB` utility, please refer to [Restoring TeamCity Data from Backup](#). This section describes restoration of a manually created backup.

You should always restore both the data in the `<TeamCity Data Directory>` and data in the database. Both the database and the directory should be backed up/restored in sync.

#### TeamCity Data Directory Restoration

You can simply put the previously backed up files back to their original places. However, it is important that no extra files are present when restoring the backup.

The simplest way to achieve this is to restore the backup over a clean installation of TeamCity. If this is not possible, make sure the files created after the backup was done are cleared. Especially the newly created files under the artifacts, messages, changes directories under `<TeamCity Data Directory>/system`.

#### TeamCity Database Restoration

The database must be restored using the database-specific tools and methods. You might also want to use database-specific methods to make the restore faster (like setting SQL Server "Recovery Model" to "Simple").

Prior to restoring, make sure there are no extra tables in the schema used by TeamCity.

#### Restoration to New Server

If you want to run a copy of the server, make sure the servers use distinct data directories and databases. For an external database, make sure you modify settings in the [TeamCity Data Directory](#)/config/database.properties file to point to a different database.

#### Restoring Build Logs

Build logs located in the /logs subdirectory of [/artifacts](#) from multiple artifact directories are backed up and restored into the single directory system/<project ID>/<build configuration name>/<internal\_build\_id>/.teamcity/logs.

## Backing up Build Agent's Data

To back up build agent's data:

1. Build Agent configuration

Back up the <Agent Home Directory>/conf/buildAgent.properties file.

You may also wish to back up any other configuration files changed (Build Agent configuration is specified in <Agent Home Directory>/conf and <Agent Home Directory>/launcher/conf directories).

2. Log files

If you need Build Agent log files (mainly used for problem solving or debug purposes), back up <Agent Home Directory>/logs directory.



You may also wish to back up Build Agent Windows Service settings, if they were modified.

## Restoring TeamCity Data from Backup

TeamCity administrators are able to restore [backed up data](#) using the `maintainDB` command-line utility.



#### Previous versions

Note that restoration of the backup created with TeamCity versions earlier than 6.0 can only be performed with the same TeamCity version as the one which created the backup.

Backups created with TeamCity 6.0+ can be restored using the same or more recent TeamCity versions.



#### Server Copying

If you are creating a copy of the server, make sure to go through [copied server checklist](#) after restoration.

On this page:

- [Performing restore](#)
- [Restoring database only](#)
- [Resuming restore after interruption](#)

You can restore backed up data into the same or a different database; from/to any of the [supported databases](#), e.g. you can restore data from a HSQL database to a PostgreSQL database, as well as restore a backup of a PostgreSQL database to a new PostgreSQL database.

During database restoration you might want to configure database-specific settings to make the bulk data changes faster (like setting SQL Server "Recovery Model" to "Simple").

**i** This document describes only some of the `maintainDB` options. For the complete list of all available options, run `maintainDB` from the command line with no parameters. See also [maintainDB startup options](#).

A TeamCity backup file does not contain build artifacts, so to get the server with all the same important data you need to restore from a backup file (at least settings and database) and copy the build logs and artifacts (located in `<TeamCity Data Directory>/system/artifacts` by [default](#)) from an old to the new data directory manually. The general compatibility rule of the data under `system/artifacts` is that files created by older TeamCity versions can be read by newer versions, but not necessarily vice versa.

When [external artifacts storage](#) is enabled, the [artifacts directory](#) of the TeamCity Data directory contains metadata about artifacts mappings, so make sure they are restored.

See also details on the directories in the [TeamCity Data Directory](#) description.

#### Performing restore

To perform restore from a backup file:

1. Install the TeamCity server from a `.tar.gz` or `.exe` installation package. Do not start the TeamCity server.
2. Create a new empty [TeamCity Data Directory](#).
3. Select one of the options:
  - a. To restore the backup into a new external database, [create and configure an empty database](#), configure a `database.properties` file with the database settings to be passed to the `restore` command later on and either place it into the `/config` subdirectory of the newly created [TeamCity Data Directory](#) or anywhere on your file system outside the TeamCity Data Directory.
  - b. To restore the data into the same database the backup was created from, proceed to the next step.
4. Place the required [database drivers](#) into the `lib/jdbc` subdirectory of the newly created [TeamCity Data Directory](#) directory.
5. Use the `maintainDB` utility located in the `<TeamCity Home>/bin` directory to run the `restore` command:
  - a. To restore the backup into a new external database
    - if the `database.properties` file is in the TeamCity Data Directory:

```
maintainDB.[cmd|sh] restore -A <absolute path to the newly created TeamCity Data Directory> -F <path to the TeamCity backup file> -T <config/database.properties>
```
    - If the `database.properties` file is outside the TeamCity Data Directory:

```
maintainDB.[cmd|sh] restore -A <absolute path to the newly created TeamCity Data Directory> -F <path to the TeamCity backup file> -T <absolute path to the database.properties file of the target database on the file system outside data dir>
```
  - b. To restore the data into the same database the backup was created from:

```
maintainDB.[cmd|sh] restore -A <absolute path to the newly created TeamCity Data Directory> -F <path to the TeamCity backup file>
```
  - c. To restore the backup into the internal database:

```
maintainDB.[cmd|sh] restore -A <absolute path to the newly created TeamCity Data Directory> -F <path to the TeamCity backup file>
```

- If the `database.properties` file is outside the TeamCity Data Directory:

```
maintainDB.[cmd|sh] restore -A <absolute path to the newly created TeamCity Data Directory> -F <path to the TeamCity backup file> -T <absolute path to the database.properties file of the target database on the file system outside data dir>
```

b. To restore the data into the same database the backup was created from:

```
maintainDB.[cmd|sh] restore -A <absolute path to the newly created TeamCity Data Directory> -F <path to the TeamCity backup file>
```

c. To restore the backup into the internal database:

```
maintainDB.[cmd|sh] restore -A <absolute path to the newly created TeamCity Data Directory> -I -F <path to the TeamCity backup file>
```

6. If the process completes successfully, copy over <TeamCity Data Directory>/system/artifacts from the old directory.

Notes on the `restore` command options:

- The `-A` argument can be omitted if you have the `TEAMCITY_DATA_PATH` environment variable set.
- The `-F` argument can be an absolute path or a path relative to the <TeamCity Data Directory>/backup directory.
- The `-T` argument must point to the `database.properties` file created in step 3.
- If the `-T` argument is not specified and the `database.properties` file is present in the newly created TeamCity Data Directory/config and the backup file, the database is restored using the properties file in TeamCity Data Directory/config.
- By default, if no other option except `-F` is specified, all of the backed up scopes will be restored from the backup file. To restore only specific scopes from the backup file, use the corresponding options of the `maintainDB` utility: `-D`, `-C`, `-U`, `-L`, and `-P`.

 To get the reference for the available options of `maintainDB`, run the utility without any command or option.

Restoring database only

Before restoring a TeamCity database to an existing server, make sure the TeamCity server is not running.

To restore a TeamCity database only from a backup file to an existing server:

1. Create and configure the database, placing the `database.properties` file into the config subdirectory of the TeamCity Data Directory.
2. Ensure that the required database drivers are present in the TeamCity Data Directory/lib/jdbc sub directory.
3. Use the `maintainDB` utility located in the <TeamCity Home>/bin directory (only available in TeamCity .tar.gz and .exe distributions).
4. Use the `restore` command (The `-T` argument must point to the `database.properties` file created in step 1):

```
maintainDB.[cmd|sh] restore -A <absolute path to TeamCity Data Directory> -F <path to the TeamCity backup file> -T <path to the database.properties file of the target database> -D
```

5. See the `maintainDB` utility console output. You may have to copy the `database.properties` file manually if requested.

Resuming restore after interruption

The restore may be interrupted due to the following reasons:

- Lack of space on the file system or in the database
- Insufficient permissions to the file system or the database.

The interruption occurs when one of tables or indexes failed to be restored, which is indicated in the `maintainDB` utility console output.

Before resuming the restore, manually delete the incorrectly restored object from the database.

To resume the backup restore after an interruption:

Run the `maintainDB` utility with the `restore` command with the required options and the additional `--continue` option:

```
maintainDB.[cmd|sh] restore <all previously used restore options> --continue
```

See also:

## Projects Import

TeamCity allows you to import projects with all their data and user accounts from a backup file to an existing TeamCity server. This approach should only be used when the task is to add projects from one server to another server when the target server is normally used. Otherwise, consider using entire [server move](#).

To import projects:

On the source TeamCity server:

- Create a usual backup file containing the projects to be imported (note that the [major](#) and [minor](#) versions of the source and target TeamCity servers have to be the same)

On the target TeamCity server:

- Go to the Server Administration area and select Project Import on the left. Upload your project settings and follow the wizard. When the import finishes, TeamCity will display the results.
- Important: to complete the import, you need to copy the artifacts from the old server to the new one by running the provided scripts: check details in the import log.



### External Artifacts storage

Projects import functionality does not support external artifacts storage. If you use external artifacts storage, you will need to move the externally stored artifacts manually to new locations using the build ids mapping generated during the import. Contact [TeamCity support](#) for details.

The following sections provide details on project import.

On this page:

- [Importing projects](#)
  - Defining import scope
  - Configuration files import
  - Importing users and groups
  - Conflicts
- [Data excluded from import](#)
- [Moving artifacts and logs](#)
- [Viewing Import Results](#)

### Importing projects

After selecting a backup file, you need to specify which projects will be imported.

TeamCity will analyze the selected projects to see if they will be imported, merged or skipped.

- The project will be imported if it is new for the target server. All its entities (Build Configurations, Templates, Builds, etc) and their data will be created on the target server.
- The project will be merged if the same project already exists on the target server (if the source and target project have the same [UUID](#) and [external ID](#)). During merging the existing entities will remain intact and only the entities new for the target will be imported with the related data. The data for the existing entities will not be imported or merged: new data will not be added to the existing entities (e.g. changes will not be added to an existing VCS root), the existing files will not be changed (e.g. if the same template exists on both servers with different settings, the target file will be preserved). It means, for example, that you cannot import missing builds to the existing build configuration. If you need to add the missing data to the existing entities, for example, import new builds into an already imported build configuration, then you should remove this build configuration using the UI and re-import its project.
- The project will be skipped if a [conflict](#) occurs: either the project's [UUID](#) is new but its [external ID](#) already exists on the target; or if the source and target projects have the same [UUID](#) but different [external IDs](#).

### Defining import scope

You can select the import scope: choose among project settings, builds and changes history, and user accounts or import all of

them. Since an imported project can also use settings from its parent, TeamCity will also import all the vcs roots, templates, meta-runners and other project-related settings for parent projects. If the same project already exists on the target server, the existing objects will not be overwritten.

## Configuration files import

For each imported or merged project, the configuration files are imported to the [data directory](#) on the target server, provided they are new for the target. The existing files will not be changed.

The following files are imported:

- Configuration xml files for the Project with its Build Configurations, Templates, and VCS Roots as well as its subprojects.
- All files from the [TeamCity Data Directory](#)/plugins directory.
- Build Numbers files for the newly added build configurations.

## Importing users and groups

When users are selected for import, TeamCity will analyze the usernames to see if users will be imported or merged.

TeamCity users must have unique usernames.

- A user account whose username is new for the target server will be imported. Such users appear on the target server in a separate group marked Imported <Import Date Time>. All the related data (personal builds, changes, test mutes and investigations) will be created on the target server. The [user account settings](#) (roles, permissions, VCS names, notification settings, etc.: system-wide settings as well as the settings related to the imported projects) are preserved during import.
- User accounts with the same username on the source and the target servers can be merged. During merging the existing data will remain intact and only the data new for the target will be added: all the new user-related data (personal builds, changes, test mutes and investigations) and the [user account settings](#) (roles, permissions, VCS names, notification settings, etc.: system-wide settings as well as the settings related to the imported projects) will be added to the user on the target server.

 Merging may cause a problem if the same username belongs to different users on the source and the target server: during import the user information will be merged anyway.

 Note that the scope of user permissions on the target may change after import, e.g.

- if a user has the system administrator role on the source, this role will be added to the user on the target after import,
- if a user has several roles in several projects on the source, only the new roles for the projects within the import scope will be added on the target.

The Project Import page | The Import scope section | Users will display the number of conflicts and you can view them and decide if you want to merge them.

TeamCity will show users with the same username and different emails on both servers, as well as the number of users with the same username and the same email.

Email verification [can be enabled](#) for the server, and the users with the same username and email are compared based on their email verification. You can view the conflicts information and choose whether to merge the users found. The options are active if users with verified emails are present either on the source or the target TeamCity server, or both.

Import of user groups works the same way: new groups are imported, while the existing groups can be merged.

If a [conflict](#) occurs (the groups exists on both the source and the target, but the group roles are different), after import the group on the target server may get additional roles. As a result, a member of this group on the target will get additional roles and permissions as well.

The Project Import page | The Import scope section | Groups will display how many conflicting groups are found. You can view all the groups that have the same group key and decide if you want to merge them. Note that "All Users" group is always listed as a conflicting one because it is a default group on all TeamCity servers.

## Conflicts

TeamCity does not import entities from the backup file if they conflict with some entity on the target server. Before import, TeamCity analyzes the backup file and displays all detected conflicts on the Import Scope configuration page.

It is highly recommended to resolve all conflicts before proceeding with the import, as unresolved conflicts may result in unpredictable behavior after the import, e.g.:

- Critical errors can be shown if, for example, some VCS Root was skipped, but a Build Configuration depending on it was imported.
- Imported Build Configurations may refer to the wrong Template if there was an unresolved conflict of [external IDs](#) betw

een the templates from the source and target servers.

## Data excluded from import

There is a number of limitations regarding the import of project-related data:

- audit records are imported only if users are selected in the scope;
- the backup files do not contain artifacts and logs (build logs are stored under build artifacts), so these are not imported automatically, but TeamCity provides scripts to move them [manually](#);
- running builds and the build queue are not included in the backup and not imported;
- global server settings (authentication schemes, custom roles, etc.) are not imported;
- internal ids (like ids of the builds) are not preserved during import. This means that URLs to the build results pages from the old server will appear broken even if redirected to the new server as build ids change on importing.



Importing projects may take significant time.

There can be only one import process per server.

## Moving artifacts and logs

Although artifacts and logs are not imported right from the backup file, you can copy/move them from the source to the target server using the .bat and .sh scripts from the `projectsImport-<date>` directory under TeamCity logs. These scripts accept the source and target data directories via the command line; the scripts accept the source and target [artifact directories](#). The rest is done automatically. The scripts can be executed while the server is running.

- It may take some time for TeamCity to display the imported build artifacts.

## Viewing Import Results

Each import process creates the `projectsImport-<date>` directory under the TeamCity logs allowing you to view the import results.

The directory contains the following:

- conflicting files folder, containing all data which has been merged
- mappings, containing mapping of the fields in the source and target databases
- scripts for copying artifacts and logs (see the section [below](#))
- import report, listing import results including the information on the data which has not been imported (if any)

## Project Export

Since TeamCity 10.0, it is possible to export settings of a project with its children to an archive to later import it to a different TeamCity server. Export includes settings (basically everything configured in the project administration area), but does not include builds or any other data visible in the user area. To export a project with all the related data, use [server backup](#).

On this page:

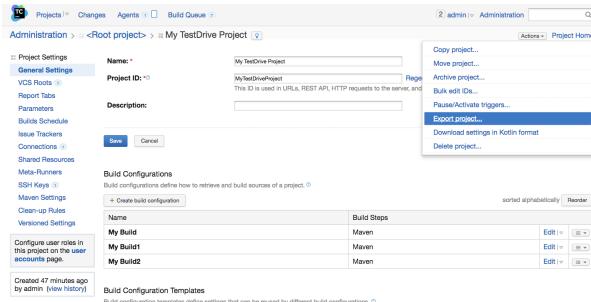
- [Exporting Project Settings](#)
- [Export Scope](#)

- Export Limitations

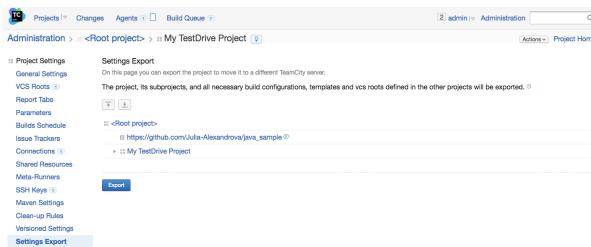
## Exporting Project Settings

To export the project settings, perform the following:

1. Go to the project settings, from the Actions menu in the top right of the project settings page select Export Project...:



2. The Settings Export page is displayed allowing exporting the project and viewing all its dependencies. Click Export to download a zip archive containing project settings.



The user exporting the project settings must have the 'view build configuration settings' permission granted to the project developer role by default. External dependencies are exported only if the user has the required permission there, otherwise a warning will be shown before export.

## Export Scope

Currently only the settings export is supported.

External dependencies for build configurations are exported as well. A build configuration defined in one project can depend on other projects in a number of ways: it can

- be associated with a template defined in parent projects,
- use vcs roots or an SSH key defined higher in the project hierarchy,
- use some external build configuration as a snapshot dependency.

The `report.log` file included in the archive details reasons for exporting external entities.

## Export Limitations

- Builds, changes, and other project-related data cannot be exported.
- External build configurations that are used in artifact dependencies only are not exported.

## TeamCity Startup Properties

Please see corresponding section:  
[Configuring TeamCity Server Startup Properties](#)  
[Configuring Build Agent Startup Properties](#)

## Configuring Server URL

The server URL configured in the Administration UI (on Administration | Global Settings page) is used by the server to generate links to the server when the URL cannot be derived from any other parameter. These cases include Notifications (email, Jabber, etc.) and some other actions performed not within a web request. All generated links will be prefixed by this URL.

Please make sure the server is accessible by the URL specified.

In most cases TeamCity correctly autodetects its own URL and sets it as the Server URL. However, sometimes autodetection is not possible/correct (for example, when the TeamCity server is running behind the Apache proxy). For such cases you can specify the server URL on the Administration | Global Settings page, or in the <TeamCity data directory>/config/main-config.xml file using the following format (no server restart is required after the change):

```
<server rootURL="http://some.host.com:port">  
</server>
```

## Configuring TeamCity Server Startup Properties

Various aspects of TeamCity behavior can be customized through a set options passed on a TeamCity server start. These options fall into two categories: affecting Java Virtual Machine (JVM) and affecting TeamCity behavior.

 You do not need to specify any of the options unless you are advised to do by the TeamCity support team or you know what you are doing.

In this section:

- [TeamCity internal properties](#)
- [JVM Options](#)
  - [Standard TeamCity Startup Scripts](#)

### TeamCity internal properties

TeamCity server has some configuration properties that are visible in the UI only to the Server Administrators. These are normally meant for debugging, changing internal constants or enabling an experimental behavior. Please do not change the internal properties unless asked by TeamCity support.

If you have internal properties customized, make sure to note this when you turn to the TeamCity support.

You can review and edit internal properties in the TeamCity web UI: go to the Administration | Server Administration | Diagnostics page, select the Internal Properties tab and click Edit internal properties. Many properties do not require the server restart, but some do.

The properties are stored in the <TeamCity Data Directory>/config/internal.properties file. The file is a Java properties file. Create the file and add a required property <property name>=<property value> on a separate line.

An alternative but obsolete way to add an internal property is to pass it as a -D<name>=<value> JVM option (see the section below).

### JVM Options

If you need to pass additional JVM options to a TeamCity server (e.g. -D options mentioned at [Reporting Issues](#) or any non--D options like -X...), the approach will depend on the way the server is run. If you are using the .war distribution, use the manual of your Web Application Server. In all other cases, please refer to the [#Standard TeamCity Startup Scripts](#) section below.

For general notes on the memory settings, please refer to [Setting Up Memory settings for TeamCity Server](#).

You will need to [restart](#) the server for the changes to take effect.

[Standard TeamCity Startup Scripts](#)

If you run the server using the `runAll` or `teamcity-server` scripts or as a Windows service, you need to set the options via the OS environment variables passed to the TeamCity server process:

- `TEAMCITY_SERVER_MEM_OPTS` — server JVM memory options (e.g. `-Xmx750m`)
- `TEAMCITY_SERVER_OPTS` — additional server JVM options (e.g. `-Dteamcity.git.fetch.separate.process=false`)

Please make sure the environment variables are set for the user whose account is used to run TeamCity or as global environment variables. You might need to reboot the machine after the environment change for the changes to have effect.

See also:

[Concepts: TeamCity Data Directory](#)

[Administrator's Guide: Configuring Build Agent Startup Properties](#)

## TeamCity Tweaks

This page lists some of the the [internal properties](#) which can be used to tweak certain aspects of TeamCity behavior. It is not recommended to use any of these unless you face an issue which you expect to address by using the properties. Please note that the support for any of these properties can be abandoned in the future versions of TeamCity without any notice. Thus, if you find a property useful in your environment please let us know about that: detail your case and the properties/values used in an email sent to our [support email address](#).

On this page:

- [Web Page Refresh Interval](#)

### Web Page Refresh Interval

You can configure different polling intervals (in seconds) for different pages in the TeamCity Web UI:

Property	Default	Description
<code>teamcity.ui.pollInterval</code>	6	How often the server is queried for common events (like build statuses, agents counter and so on). Since TeamCity 9.0 this property works only if WebSocket connection is not available and polling is used instead.

teamcity.ui.events.pollInterval	6	<p>the delay between an event (received via polling or WebSockets) and the ajax request to update the UI.</p> <ul style="list-style-type: none"> <li>With WebSocket, a client receives the event immediately, but reacts to it after the specified interval; as a result, e.g. a started build appears on the Overview page with a delay.</li> <li>With polling, a client receives the event during the polling request determined by <code>teamcity.ui.events.pollInterval</code> and reacts to it after the delay defined by <code>teamcity.ui.events.pollInterval</code>: e.g. a started build appears on the Overview page after <code>teamcity.ui.events.pollInterval + teamcity.ui.events.pollInterval</code> seconds</li> </ul>
teamcity.ui.systemProblems.pollInterval	20	
teamcity.ui.problemsSummary.pollInterval	8	
teamcity.ui.buildQueueEstimates.pollInterval	10	

## Setting up Google Mail and Google Talk as Notification Servers

This section covers how to set up the Google Mail and Google Talk as notification servers when configuring the TeamCity server.

### Google Mail

On the Administration | EMail Notifier page set the options as described below:

Property	Value
SMTP host	smtp.gmail.com
SMTP port	465
Send email messages from	E-mail address to send notifications from.
SMTP login	Full username with domain part if you use Google Apps for domain
SMTP password	User's GMail password
Secure connection	SSL

(see also [Google help](#))

### Google Talk

On the Administration | Jabber Notifier page set the options as described below:

Property	Value
Server	talk.google.com
Port	5222
Server user	Full username with domain part if you use Google Apps for domain
Server user password	User's GMail password
Use legacy SSL	no

## Using HTTPS to access TeamCity server

On this page:

- Authenticating with server certificate (HTTPS with no client certificate)
- Configuring JVM
  - Configuring JVM for authentication with the server certificate
  - Configuring JVM for authentication with client certificate
- Useful tools in analyzing certificates issues

This document describes how to configure various TeamCity server clients to use HTTPS for communicating with the server.



If you need to connect the TeamCity server to a service behind a self-signed certificate (e.g. Git) or if you need to connect an agent to the TeamCity server behind the self-signed certificate, see this section.

The [JVM configuration](#) instructions can also be used to configure TeamCity server JVM to connect to other HTTPS/SSL services.

We assume that you have already configured HTTPS in your TeamCity web server. The most common and recommended approach for this is to set up a middle proxying server like Nginx or Apache that will handle HTTPS with TeamCity server's Tomcat handling only HTTP requests. In the setup please make sure that the middle server/proxy has correct URL rewriting configuration, see also [Set Up TeamCity behind a Proxy Server](#) section.

For small servers you can also set up HTTPS by the internal [Tomcat means](#), but this is not recommended.

See also a feature request: [TW-12976](#).

### Authenticating with server certificate (HTTPS with no client certificate)

If your certificate is valid (i.e. it was signed by a well known Certificate Authority like Verisign), then TeamCity clients should work with HTTPS without any additional configuration. All you have to do is use `https://` links to the TeamCity server instead of `http://`.

If your certificate is not valid (is self-signed): (i.e. is not signed by a known Certificate Authority and likely to result in "PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path to requested target" error message)

- To enable HTTPS connections from the TeamCity [Visual Studio Addin](#) and [Windows Tray Notifier](#), point your Internet Explorer to the TeamCity server using `https://` URL and import the server certificate into the browser. After that, the Visual Studio Addin and Windows Tray Notifier should be able to connect by HTTPS.
- To enable HTTPS connections from Java clients (TeamCity Agents, IntelliJ IDEA, Eclipse, etc.), see the [section below](#) for configuring the JVM installation used by the connecting application.

### Configuring JVM

#### Configuring JVM for authentication with the server certificate

If your certificate is valid (i.e. it was issued and signed by a well known Certificate Authority like Verisign), then the Java clients should work with HTTPS without any additional configuration. To use Let's Encrypt-issued certificates, make sure to upgrade the JVM used by the client to the latest.

If your certificate is not valid (is self-signed):

For versions prior to TeamCity 2018.1, to enable HTTPS connections from Java clients, you need to install the server certificate (or your organization's certificate the server's certificate is signed by) into the JVM as a trusted certificate. These are generic Java application steps (not TeamCity-specific):

- save the CA Root certificate of the server's certificate to a file in one of the [supported formats](#) (the file is referred as `<cert file>` below). This can be done in a browser by inspecting certificate data and exporting it as Base64 encoded X.509 certificate.
- locate the JRE used by the process. The best way to get the path to the proper Java installation is to look up the command line of the running process.
  - If there is a JDK installed (like for IntelliJ IDEA), `<path to JRE installation>` should be `<path to used JDK>/jre`
  - For TeamCity agent or server installed under Windows, the default location for `<path to JRE installation>` is `<TeamCity installation path>/jre`
- import the server certificate into the default JRE installation keystore using JVM's `keytool` tool:

```
keytool -importcert -file <cert file> -keystore <path to JRE installation>/lib/security/cacerts
```

- By default, Java keystore is protected by password "changeit" which you need to type on prompt
- restart the JVM

#### Configuring JVM for authentication with client certificate

##### Importing client certificate

If you need to use a client certificate to access a server via https (e.g. from IntelliJ IDEA, Eclipse or the build agents), you will need to add the certificate to the Java keystore and supply the keystore to the JVM used by the connecting process.

1. If you have your certificate in a p12 file, you can use the following command to convert it to a Java keystore. Make sure you use keytool from JDK 1.6-1.8: earlier versions may not understand p12 format.

```
keytool -importkeystore -srckeystore <path to your .p12 certificate> -srcstoretype PKCS12  
-srcstorepass <password of your p12 certificate> -destkeystore <path to keystore file>  
-deststorepass <keystore password> -destkeypass <keystore password> -srcalias 1
```

This command extracts the certificate with the alias "1" from your .p12 file and adds it to Java keystore. You should know <path to your .p12 certificate> and <password of your p12 certificate> and you can provide new values for <path to keystore file> and <keystore password>.

Here, `keypass` should be equal to `storepass` because only `storepass` is supplied to the JVM, and if `keypass` is different, the following error may occur: "java.security.NoSuchAlgorithmException: Error constructing implementation (algorithm: Default, provider: SunJSSE, class: com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl)".

Importing root certificate to organize a chain of trust

If your certificate is not signed by a trusted authority, you will also need to add the root certificate from your certificate chain to a trusted keystore and supply this trusted keystore to JVM.

2. You should first extract the root certificate from your certificate. You can do this from a web browser if you have the certificate installed, or you can do this with the [OpenSSL](#) tool using the command:

```
openssl.exe pkcs12 -in <path to your .p12 certificate> -out <path to your certificate in .pem format>
```

You should know <path to your .p12 certificate> and its password (to enter it when prompted). You should specify new values for <path to your certificate in .pem format> and for the pem pass phrase when prompted.

3. Then you should extract the root certificate (the root certificate should have the same issuer and subject fields) from the pem file (it has text format) to a separate file. The file should look like:

```
-----BEGIN CERTIFICATE-----  
MIIGUjCCBDqgAwIBAgIEAKmKxzANBgkqhkiG9w0BAQQFADBwMRUwEwYDVQQDEwxK  
...  
-----END CERTIFICATE-----
```

Let's assume its name is <path to root certificate>.

4. Now import the root certificate to the trusted keystore with the command:

```
keytool -importcert -trustcacerts -file <path to root certificate> -keystore <path to trust keystore file>  
-storepass <trust keystore password>
```

Here you can use new values for <trust keystore path> and <trust keystore password> (or use existing trust keystore).

Starting the connecting application JVM

Now you need to pass the following parameters to the JVM when running the application:

```
-Djavax.net.ssl.keyStore=<path to keystore file>  
-Djavax.net.ssl.keyStorePassword=<keystore password>  
-Djavax.net.ssl.trustStore=<path to trust keystore file>  
-Djavax.net.ssl.trustStorePassword=<trust keystore password>
```

For IntelliJ IDEA, you can add the lines into the `bin\idea.exe.vmoptions` file (one option per line).

For the TeamCity build agent, see [agent startup properties](#).

Useful tools in analyzing certificates issues

Online HTTPS server configuration analysis: <https://www.ssllabs.com/ssltest/analyze.html>  
SSL Poke Java class

## TeamCity Disk Space Watcher

TeamCity server regularly checks for free disk space on the server machine (in the TeamCity Data Directory) and displays a warning on all the pages of the web UI if the free disk space falls below a certain threshold. If the space continues to decrease and reaches a certain limit, the build queue is paused.

The thresholds can be changed by modifying the following [internal properties](#) values in KB:

Property	Default	Description
teamcity.diskSpaceWatcher.threshold	512000 (500 MB)	displays a warning
teamcity.pauseBuildQueue.diskSpace.threshold	51200 (50 MB)	pauses the build queue

You can use [Disk Usage](#) report to evaluate what projects use the most of the disk space and adjust the [clean-up](#) rules if required.

See also:

[Administrator's Guide: Free disk space on agent | Disk Usage | Clean-up](#)

## TeamCity Server Logs

TeamCity Server keeps a log of internal activities that can be examined to investigate an issue with the server behavior or get internal error details.

The logs are stored in plain text files in a disk directory on the TeamCity server machine (usually in `<TeamCity Server home>/logs`).

The files are appended with messages when TeamCity is running.

While the server is running, the logs can be viewed in the web UI on the [Server Logs](#) tab of [Administration | Diagnostics](#) section.



### Enable Debug in Server Logs

In the web UI, go to [Administration | Diagnostics](#) page. On the Troubleshooting tab, choose a logging preset, view logs under Server Logs subsection.

If it is not possible to enable debug logging mode from the TeamCity web UI, refer to [Changing Logging Configuration](#) section to learn how to adjust logging options manually.

In this section:

- [General Logging Description](#)
- [Logging-related Diagnostics UI](#)
- [Changing Logging Configuration](#)
  - [Changing Logging Settings](#)
- [Reading Logs](#)
  - [General Logging Configuration](#)

### General Logging Description

TeamCity uses [log4j library](#) for the logging and its settings can be [customized](#).

By default, log files are located under the `<TeamCity Server home>/logs` directory.

The most important log files are:

<code>teamcity-server.log</code>	General server log
<code>teamcity-activities.log</code>	Log of user-initiated and main build-related events

teamcity-vcs.log	Log of VCS-related activity
teamcity-cleanup.log	contains clean-up-related log
teamcity-notifications.log	Notifications-related log
teamcity-clouds.log	(off by default) Cloud-integration-related log
teamcity-sql.log	(off by default) Log of SQL queries, see <a href="#">details</a>
teamcity-http-auth.log	(off by default) Log with messages related to <a href="#">NTLM</a> and other authentication for HTML requests
teamcity-xmlrpc.log	(off by default) Log of messages sent by the server to agents and IDE plugins via XML-RPC
vcs-content-cache.log	(off by default) Log related to individual file content requests from VCS
teamcity-rest.log	(off by default) REST-API related logging
teamcity-freemarker.log	(off by default) Notification templates processing-related logging
teamcity-agentPush.log	(off by default) Logging related to agent push operations
teamcity-remote-run.log	(off by default) Logging related to personal builds processing on the server
teamcity-svn.log	(off by default) SVN integration log
teamcity-tfs.log	(off by default) TFS integration log
teamcity-starteam.log	(off by default) StarTeam integration log
teamcity-clearcase.log	(off by default) ClearCase integration log
teamcity-ldap.log	LDAP-related log
teamcity-nuget.log	NuGet-related log
teamcity-maintenance.log	(off by default) logs of back-up/ restore/ migration performed with <a href="#">maintainDB tool</a>
teamcity-maintenance-truncation.log	(off by default) contains extended information on possible data truncation during back-up/ restore/ migration performed with <a href="#">maintainDB tool</a>
teamcity-versioned-settings.log	(off by default) contains information on synchronization of the project settings with the version control
teamcity-ws.log	logs related to communication between browsers and the TeamCity server using the <a href="#">WebSocket connection</a>
teamcity-issue-trackers.log	logs related to communication between TeamCity and configured issue trackers

Other files can also be created on [changing Logging Configuration](#).

Some of the files can have ".N" extensions - that are files with previous logging messages copied on main file rotation. See [maxBackupIndex](#) for preserving more files.

## Logging-related Diagnostics UI

Users with System Administrator role can view and download the server logs right from the TeamCity UI using Administration | Diagnostics | Server Logs.

The debug logging can be enabled via "Active logging preset" under the Administration | Diagnostics page, Troubleshooting, Debug logging subsection. Choosing a preset changes logging configuration immediately and the preset is preserved after a server restart, until changed on the page again. It is recommended to return to the "<Default>" once the necessary logs were collected.



Prior to TeamCity 9.1 the preset was reset to the default on the server restart, see [the related issue](#).

New presets can also be uploaded via Diagnostics | Logging Presets.

The available presets are configured by the files available under <TeamCity Data Directory>/config/\_logging directory with .xml extension. New files can be added into the directory and existing files can be modified (using [.dist convention](#)).

## Changing Logging Configuration

While TeamCity is running, logging configuration for the server can be switched to a logging preset.

If it is not possible to enable debug logging mode via logging presets (e.g. to get the logging during server initialization) or to make persistent changes to the logging, you can backup the `conf/teamcity-server-log4j.xml` file and copy/ rename the `<TeamCity Data Directory>/config/_logging/debug-general.xml` file over `conf/teamcity-server-log4j.xml` before the server start.

## Changing Logging Settings

If you want to fine-tune the log4j configuration, you can edit <TeamCity Server home>/conf/teamcity-server-log4j.xml file (for .war TeamCity distribution, see [the related section](#)). If the server is running, the log4j configuration file will be reloaded automatically and the logging configuration will be changed on the fly (some log4j restrictions still apply, so for a massive change consider restarting the server).

Most useful settings of log4j configuration:

To change the minimum log level to save in the file, tweak the "value" attribute of the "priority" element:

```
<category ...>
  <priority value="INFO"/>
  ...
  ...
```

The logs are rotated by default. When debug is enabled, it makes sense to increase the "value" attribute of "maxBackupIndex" element to affect the number of preserved log files. While doing so, please ensure there is sufficient free disk space available.

```
<appender ...>
  <param name="maxBackupIndex" value="10"/>
  ...
  ...
```

For detailed description of `maxBackupIndex` and other supported attributes, see <http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/RollingFileAppender.html>.

## Reading Logs

Each message has a timestamp and level (ERROR, WARN, INFO, DEBUG).

e.g.:

```
[2010-11-19 23:22:35,657] INFO - s.buildServer.SERVER - Agent vmWin2k3-3 has been registered with id 19, not running a build
```

ERROR means an operation failed and some data was lost or action not performed. Generally, there should be no ERRORS in the log.

WARNs generally means that an operation failed, but will be retried or the operation is considered not important. Some amount of WARNs is OK. But you can review the log for such warnings to better understand what is going OK and what is not.

INFO is an informational message that just reports on the current activities.

DEBUG is only useful for issue investigation. e.g. to be analyzed by TeamCity developers.

## General Logging Configuration

By default TeamCity searches for log4j configuration in the `../conf/teamcity-server-log4j.xml` file (this resolves to <TeamCity Server home>/conf/teamcity-server-log4j.xml for TeamCity .exe and .tar.gz distributions when run from "bin"). If no such file is present, the default log4j configuration is used.

The logs are saved to the `../logs` directory by default.

The configuration options values can be changed via the corresponding `log4j.configuration` and `teamcity_logs` JVM options or [internal properties](#).

For example: `log4j.configuration=...`/`conf/teamcity-server-log4j.xml` and `teamcity_logs=...`/`logs/`. Default values can be looked up in the `bin/teamcity-server` script available in the `.exe` and `tar.gz` distributions.

If you start TeamCity by the means other than the bundled `teamcity-server` or `runAll` scripts, please make sure to pass the above-mentioned options to the server JVM.

See also the [recommendations](#) on installing TeamCity into not bundled web server.

The default `teamcity-server-log4j.xml` file content can be found in the `.exe` and `tar.gz` distributions. The one with debug enabled can be found under `TeamCity Data Directory/config/_logging/debug-general.xml` name after server's first start. See also sample `teamcity-server-log4j.xml` file.

See also:

[Administrator's Guide: Viewing Build Agent Logs](#)  
[Troubleshooting: Reporting Issues](#)

## Build Agents Configuration and Maintenance

On this page:

- [Viewing TeamCity agents details](#)
  - [Connected / Disconnected](#)
    - [Enabling/Disabling Agents via UI](#)
  - [Pools](#)
  - [Parameters report](#)
  - [Matrix and Statistics](#)
  - [Cloud](#)
  - [Diff](#)

### Viewing TeamCity agents details

The Agents page of the TeamCity Web UI provides the comprehensive information on the TeamCity agents. The number of tabs on the page may differ depending on your agent setup.

#### Connected / Disconnected

The Connected and Disconnected tabs display the agents by [Agent pool](#) (default). To view the agents alphabetically, uncheck the Group by agent pool box.

For each pool TeamCity displays the status of its build agents. Clicking the arrow next to the pool displays the list of the pools agents with their statuses.

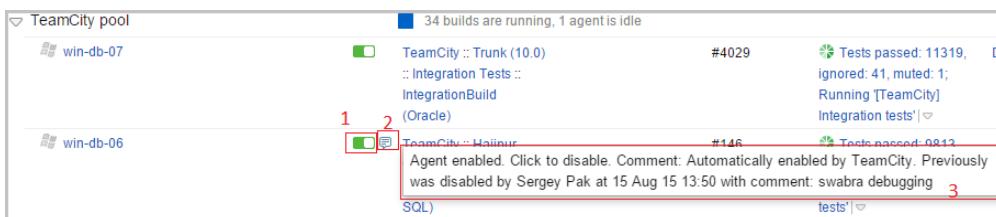
#### Enabling/Disabling Agents via UI

TeamCity distributes builds only among the enabled agents.

Agents can be manually enabled/disabled via the web UI by clicking the status icon (1) next to the agent's name.

Optionally, you can tell TeamCity to automatically disable/enable the agent after a period of time and enter your comment.

TeamCity will follow the instructions and show the comment icon (2). Hovering over the icons will display the related information (3).



#### Pools

Refer to a [separate page](#) for information on configuring agent pools in TeamCity.

#### Parameters report

Filter all available agents using a specified parameter.

## Matrix and Statistics

Refer to a separate page for information on [viewing the agents workload](#).

## Cloud

Lists all configured agent cloud profiles.

## Diff

Compare two agents and see their differences highlighted.

## Agent Pools

- [Concept](#)
- [Required Permissions](#)
- [Managing Agent Pools](#)

### Concept

Instead of having one common set of agents, you can break them into separate groups called agent pools. A pool is a named set of agents to which you can assign projects.

- An agent can belong to one pool only.
- A project can use several pools for its builds.

The number of agents authorized by the TeamCity server is limited by the number of [agent licenses](#). By default, all newly authorized agents are included into the Default pool.

With the help of agent pools you can bind specific agents to specific projects. Project builds can be run only on build agents from the pools assigned to the project. Agent pools can also help to calculate the required agents capacity.

You can find all agent pools configured in TeamCity at the [Agents | Pools](#) tab.

### Required Permissions

To be able to add\remove pools and set maximum number of agents in the pool, you need to have the "Manage agent pools" permission granted to the System administrator and Agent manager [roles](#) in the default TeamCity [per-project authorization mode](#).

Assigning and un-assigning projects and agents to/from pools is restricted by the "Change agent pools associated with project" permission, which by default is a part of the Project administrator role. A user can perform the operations on the pool only if he/she has "Change agent pools associated with project" for all projects associated with all pools affected by the operation.

See also [agent management permissions](#).

### Managing Agent Pools

The [Agents | Pools](#) tab allows managing pools in TeamCity.

To create a new agent pool, you need to specify its name.

By default, a pool can contain an unlimited number of agents. You can set a maximum number of agents in the pool (not applicable for the Default Pool). If the maximum number of agents is reached, TeamCity will not allow adding any new agents to this pool. This includes moving agents from other pools and automatic authorization of cloud agents. New cloud agents will not start if the target pool is full.

To populate a pool with agents, click [Assign agents...](#) and select them from a list. Since an agent can belong to one pool only, assigning it to a pool will remove it from its previous pool. If this may cause compatibility problems, TeamCity will give you a warning. Removing an agent from a custom pool will return it to the Default pool. You can also assign cloud agents to a specific pool when adding an image to a cloud profile.



Moving cloud agents

Only the cloud agent images configured in the the <Root> project can be moved using [Assign agents...](#) . To assign a pool to a cloud agent, please configure it using [Agent Cloud profile](#), in the cloud image details.

When you have configured agent pools, you can:

- Filter the build queue by pools.
- Use grouping by pool on the [Agent Matrix](#) and [Agent Statistics](#) pages.

See also:

[Concepts: Build Agent | Agent Requirements](#)  
[Administrator's Guide: Viewing Agents Workload](#)

## Configuring Build Agent Startup Properties

In TeamCity a [build agent](#) contains two processes:

- Agent Launcher — a Java process that launches the agent process
- Agent — the main process for a Build Agent; runs as a child process for the agent launcher

Whether you run a build agent via the `agent.bat|sh` script or as a Windows service, at first the agent launcher starts and then it starts the agent.



You do not need to specify any of the options unless you are advised to do so by the TeamCity support team or you know what you are doing.

In this section:

- [Agent Properties](#)
  - [Build Agent Is Run Via Script](#)
  - [Build Agent Is Run As Service](#)
- [Agent Launcher Properties](#)
  - [Build Agent Is Run Via Script](#)
  - [Build Agent Is Run As Service](#)

### Agent Properties

For both processes above you can customize the final agent behavior by specifying system properties and variables for the agent to run with.

#### Build Agent Is Run Via Script

Before you run the `<Agent Home>\bin\agent.bat|sh` script, set the following environment variables:

- `TEAMCITY_AGENT_MEM_OPTS` — Set agent memory options (JVM options)
- `TEAMCITY_AGENT_OPTS` — additional agent JVM options

#### Build Agent Is Run As Service

In the `<Agent Home>\launcher\conf\wrapper.conf` file, add the following lines (one per option):

```
wrapper.app.parameter.<N>
```

- You should add additional lines before the following line in the `wrapper.conf` file:

```
wrapper.app.parameter.N=jetbrains.buildServer.agent.AgentMain
```

- Please ensure to re-number all the lines after the inserted ones.

#### Agent Launcher Properties

It's rare that you would ever need these. Most probably you would need affecting main agent process properties described above.

#### Build Agent Is Run Via Script

Before you run the `<Agent Home>\bin\agent.bat|sh` script, set the `TEAMCITY_LAUNCHER_OPTS` environment variable.

#### Build Agent Is Run As Service

In the `<Agent Home>\launcher\conf\wrapper.conf` file, add the following lines (one per option, the `N` number should increase):

```
wrapper.java.additional.<N>
```

⚠ Make sure you re-number all the lines after the inserted ones.

#### See also:

Concepts: [Agent Home Directory](#)

Administrator's Guide: [Configuring TeamCity Server Startup Properties](#)

#### Viewing Agents Workload

TeamCity provides handy ways to estimate build agents efficiency and help you manage your system:

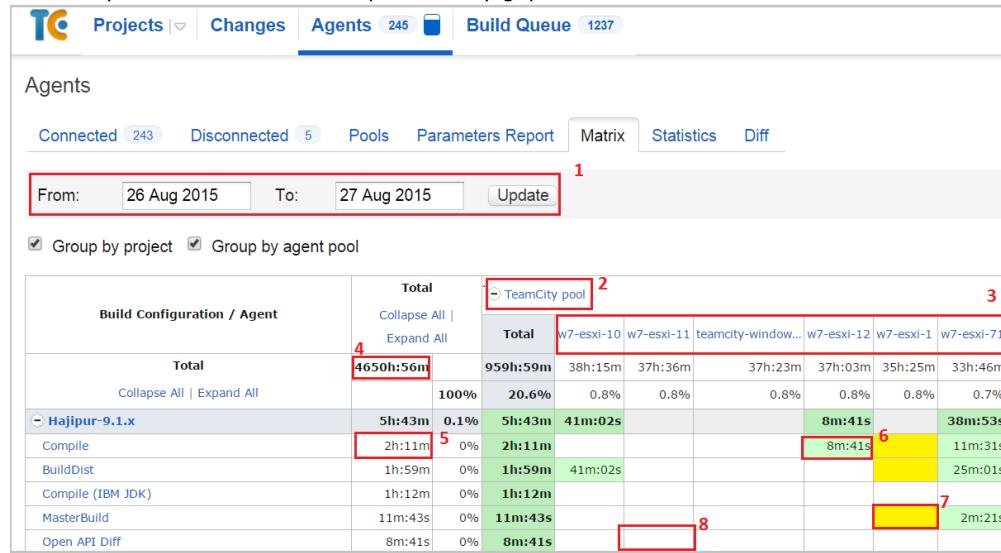
- [Load statistics matrix](#)
- [Build Agents' workload statistics](#)

#### Load Statistics Matrix

The Matrix available at the Matrix tab on the Agents page provides you with a bird's-eye view of the overall Build Agents workload for all finished builds during the time range you selected.

By taking a look at the build configurations compatible with a particular agent, you can [assign the build configuration to](#)

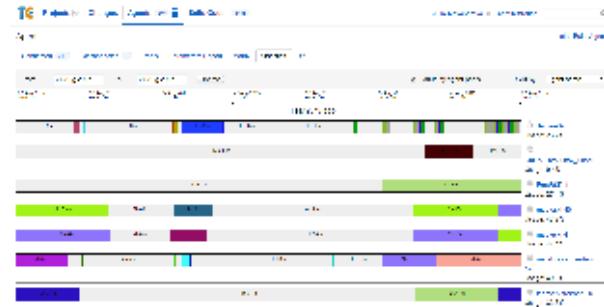
particular Build Agents and significantly lower the idle time. This helps you adjust the hardware resources usage more effectively and fill the discovered productivity gaps.



1. Specified time range. Click Update to reload the matrix.
2. [Agent pool](#). Clicking a link opens the pool's page.
3. Individual agents. Clicking a link opens the [agent's details](#) page.
4. Total build duration during the specified time.
5. Total build duration for a particular build configuration during the specified time range.
6. Total duration for a particular build configuration of builds that were run on the particular build agent during the specified time period.
7. The build agent is compatible with the build configuration, but no builds were run during the specified time range.
8. The build agent is incompatible with the configuration, or disconnected.

## Build Agents' Workload Statistics

TeamCity provides a number of visual metrics, namely, statistics of Build Agents' activity and their usage during a particular time period. These information is available at the Statistics tab on the Agents page.



You'll find this feature helpful in:

- your daily administration activities aimed at lowering your company's network workload
- locating and eliminating the gap between the most frequently used computers and those which are often idle
- reducing the cost of your hardware resources ownership

See also:

[Concepts: Build Agent](#)

[Installation and Upgrade: Installing and Running Additional Build Agents](#)

## Viewing Build Agent Details

To view the state and information about an agent, click its name or navigate to the Agents page, find the agent in the list of connected, disconnected or authorized agents and click its name.

For each connected agent TeamCity provides the following information:

- Agent Summary
  - Agent Reboot
- Build History
- Compatible Configurations
- Build runners
- Agent logs
- Agent Parameters

#### Agent Summary

- Status: [learn more about an agent's status](#).
- Details:
  - the agent name
  - the agent host IP address
  - the port used by the TeamCity server to connect to the agent
  - the communication [protocol](#) used for data transfers between the agent and the server
  - the agent operating system
  - CPU rank: the result of the bundled CPU benchmark. Note that the benchmark results can depend on the JVM version and options used by the agent. For example, '-server' JVM option has a significant influence on the results. CPU benchmark also affects the way how the server distributes builds among agents. If a build duration estimate cannot be calculated for an agent (there were no builds in the history ran on this agent), TeamCity chooses the fastest agent basing on the CPU benchmark value.
  - the [pool](#) the agent belongs to
  - the agent launcher version
- If there is a running build on the agent, the page displays information on the build with the link to the [build results](#).
- Miscellaneous: this section provides the options to
  - [Clean sources on the agent](#)
  - Open Remote Desktop: available for agents running on the Windows operating system if the RDP client is installed and registered on the agent.
  - Reboot Agent Machine: available to users with Reboot build agent machines permissions. Click the link and confirm the reboot action. By default, the TeamCity agent will wait until the current build finishes. Deselect the checkbox and click Reboot to restart the agent immediately.  
Additional configuration of the reboot command is possible.
    - ▼ [Click to view details](#).

#### Agent Reboot

Agent reboot is performed by executing an OS-specific command.

Under certain circumstances the command might need customization specific to the OS environment.  
Additional configuration might be required if the default reboot command fails.

To tweak the agent reboot, add the `teamcity.agent.reboot.command` agent configuration parameter to the `buildagent.properties` file with the command to execute when reboot is required.

Example configuration:

```
teamcity.agent.reboot.command=sudo shutdown -r 60
or
teamcity.agent.reboot.command=shutdown -r -t 60 -c "TeamCity Agent reboot
command" -f
```

- Dump threads on agent.

#### Build History

Shows the builds that were run on the agent

#### Compatible Configurations

Displays compatible and incompatible build configurations with the reason of incompatibility.

#### Build runners

Lists build runners supported by build agent.

#### Agent logs

The page allows viewing and downloading the logs.

## Agent Parameters

The tab lists system properties, environment variables, and configuration parameters. Refer to the [Configuring Build Parameters](#) page for more information on different types of parameters.

See also:

[Concepts: Build Agent | Run Configuration Policy](#)

[Installation and Upgrade: Installing and Running Additional Build Agents](#)

## Viewing Build Agent Logs

- [Log Files](#)
- [Generic Debug Logging](#)
- [VCS Debug Logging](#)
- [Specific Debug Logging](#)
- [Advanced Logging Configuration](#)

To analyze agent-specific cases, there are internal log files saved by the TeamCity agent process into <TeamCity agent home>/logs directory on the agent machine.

When the agent is connected to TeamCity server, you can browse and download the agent logs in TeamCity web UI, on [Agent Logs](#) tab for an agent.

If you need to customize the logging, see below.

### Log Files

TeamCity uses [Log4j](#) for internal logging of events. The default build agent Log4j configuration file is <agent home>/conf/teamcity-agent-log4j.xml

See the comments in the file for enabling the DEBUG mode: you will need to increase the value in "<param name="maxBackupIndex"...>" node and insert "<priority value="DEBUG"/>" nodes into "<category>" elements. Build agent logs are placed into <agent home>/logs directory.

Normally, you do not need to restart the agent for the updated logging configuration to be applied.

File name	Description
teamcity-agent.log	General build agent log
teamcity-build.log	stdout and stderr output of builds run by the agent
teamcity-vcs.log	VCS-related logging (for checkout mode "Automatically on agent")
upgrade.log	log of the build agent upgrade (logged by the upgrading process)
launcher.log	log of the agent's monitoring/launching process
wrapper.log	(only present when the agent is run as Windows service or by Java Service Wrapper) output of the process build agent launching process

### Generic Debug Logging

To enable general debug logging on agent, change the "jetbrains.buildServer" category logging priority in the <agent home>/conf/teamcity-agent-log4j.xml file:

```
<category name="jetbrains.buildServer">
  <priority value="DEBUG"/>
  <appender-ref ref="ROLL"/>
</category>
```

Then, see `teamcity-agent.log*` files

To turn the debug mode off, make the line "`<priority value="INFO" />`".

### VCS Debug Logging

To enable detailed VCS logging on agent, change the VCS category logging priority in the `<agent home>/conf/teamcity-agent-log4j.xml` file:

```
<category name="jetbrains.buildServer.VCS">
  <priority value="DEBUG"/>
  <appender-ref ref="ROLL.VCS"/>
</category>
```

Then, see `teamcity-vcs.log*` files

To turn debug mode off, make the line "`<priority value="INFO"/>`".

#### Specific Debug Logging

To get dump of the data sent from the agent to the server, enable agent XML-RPC log, by uncommenting the line below in the `<agent home>/conf/teamcity-agent-log4j.xml` file.

```
<category name="jetbrains.buildServer.XMLRPC">
  <!--<priority value="DEBUG"/>-->
  <appender-ref ref="ROLL.XMLRPC"/>
</category>
```

Then, see `teamcity-xmlrpc.log`

To turn it off, make the line "`<priority value="INFO"/>`".

#### Advanced Logging Configuration

You can configure location of the logs by altering the value of the `teamcity_logs` property (passed to JVM via `-D` option). You can also change the Log4j configuration file location by changing the value of the `log4j.configuration` property. See the corresponding documentation [section](#) on how to pass the options.

For additional options on tweaking logging, please consult the [TeamCity Server Logs#log4jConfiguration](#) section.

See also:

[Administrator's Guide: TeamCity Server Logs](#)  
[Troubleshooting: Reporting Issues](#)

## TeamCity Memory Monitor

TeamCity server checks available memory on a regular basis and warns you if the amount of the memory available is too low.

There are several warning types reported:

- Low pool memory
- Low total memory
- Heavy GC overload
- Customization

Low pool memory

Is reported when memory usage in a single memory pool exceeds 90% after garbage collection. High server activity may cause such memory usage.

Low total memory

Is reported when more than 90% of total memory has been in use during the last 5 minutes and more than 20% of CPU resources are being consumed by garbage collection. Lasting memory lack may result in performance degradation and server instability as well.

Heavy GC overload

Is reported when memory cleaning takes more than 50% of CPU resources on average. It usually means really serious

problems with memory resulting in high performance degradation.

#### Customization

Several [internal properties](#) can be used to customize the Monitor:

`teamCity.memoryUsageMonitor.poolNames` sets up pool names to track. Case-sensitive comma-separated string is accepted

`teamCity.memoryUsageMonitor.warningThreshold` allows setting up a minimal warning threshold. Affects all tracked memory pools except for PermGen ([replaced with metaspace](#) memory allocation in TeamCity 10)

`teamCity.memoryUsageMonitor[<Pool name>].warningThreshold` can be used to modify single memory pool threshold. Spaces should be escaped or changed to '\_' signs

`teamCity.memoryUsageMonitor.gcWarningThreshold` allows setting up the allowed percentage of resources to spent for cleaning the memory

See also:

[Reporting Issues: Out Of Memory Problems](#)

[Increasing Server Memory: Installing and Configuring the TeamCity Server](#)

## Disk Usage

TeamCity analyses disk space occupied by builds on the server machine and provides the Disk Usage report. To see the report, open Administration| Disk Usage.

The report contains information on the total free disk space and the amount of space taken by build artifacts and build logs. The default location for this directories is [TeamCity Data Directory/system](#). If build logs and artifacts directories are [located on different disks](#), free space is reported for each of the disks. The report also displays [pinned builds](#) if available, and the space taken by their logs and artifacts.

By default, the report displays data on the builds run from build configurations grouped by projects. You can choose to view the ungrouped list of build configurations or to show archived projects if required using the corresponding checkboxes. You can sort the information by clicking the column name.

The report allows you to see which projects consume most of disk space and configure the [clean-up rules](#); the link to the report is also available from the build history clean-up page. This page also displays disk usage information for the selected project or build configuration when managing [clean-up rules](#) in the Configure Clean-up section.

You can also see which configurations produce [large build logs](#) and adjust the settings accordingly.

The report is automatically updated when a new build is run or removed: only the data pertaining to this build is analyzed and the corresponding information is reflected in the report. The report is completely updated when TeamCity performs a full disk scan: by default, after a build history clean-up is run. You can force TeamCity to scan the disk on demand using the Rescan now button, but it may be time-consuming.

The Disk Usage report allows going deeper in the build history of a specific build configuration and showing some additional statistics for the builds of this configuration. The [clean-up rules](#) for this build configuration are also listed allowing you to adjust the settings based on the data displayed:

The screenshot shows the TeamCity Server Health report. At the top, there's a navigation bar with links like 'Dashboard', 'Builds', 'Tests', 'Logs', 'Artifacts', 'Metrics', 'Audit', and 'Help'. Below the navigation, there's a section titled 'Server Health' with a table of metrics:

Category	Value	Unit	Impact
Memory	288 877 MB	1%	Medium
Swap	234 876 MB	9%	Medium
Log Files	24 348 MB	2%	Medium
Logs	53 959 MB	>1%	Medium
VFS	5 554 MB	<1%	Medium
VDS	1 769 MB	<1%	Medium
Local Log Files	1 492 MB	<1%	Medium
Harver Cache Analysis	709 MB	<1%	Medium
Concurrent Line Reader Tests	247 MB	<1%	Medium
Process Resource Tests	345 MB	<1%	Medium
DB Tests	599 MB	<1%	Medium
Oracle Health Tests	22 MB	<1%	Medium

Below this, there's a detailed 'Configuration TeamCity :: Gaya Trunk :: Metrics/Details' table with many rows of data. Further down, there's a 'Cleanup rules for TeamCity :: Gaya Trunk :: Unresolved' section with a table of cleanup rules.

## Server Health

The Server Health report contains results of the server inspection for any configuration issues which impact or could potentially impact the performance. Such issues, the so called server health items, are collectively reported by TeamCity on the Server Health page in the Administration area.

The Project Administrator [permissions](#) at least are required to see the report.

### Scope and Severity

The report enables you to define the analysis scope: you can select to analyze the global configuration or report on project-related items. The scope available to you depends on the level of the View Project permission granted to you. Note that the report is not available for archived projects.

The Server Health analysis also employs the severity rating, depending on the issue impact on the configuration of the system on the whole or an individual project.

### Visibility Options

By default, the warning and error level results pertaining to the global configuration will be displayed on the report page as well as at the top of each page in TeamCity.

Besides those, some items will be displayed in-place: depending on the object causing the issue, the server health item will be reported on the Build Configuration, Template or VCS root settings page.

Only active items are displayed on the TeamCity pages. To remove an item from display, use the hide option next to an item on the report page. For global items, this option is available in every server health message.

Hidden items will be removed from the TeamCity pages, and will be displayed on the Server Health page below the active items. Their description will be greyed out.

To return an item to display, use the Show option.

The visibility changes will be listed on the Audit page in the Administration area.

### Issue Categories

Health items cover a wide range of server functionality to allow administrators to easily monitor the TeamCity overall status.

The main configuration issue categories are listed below:

### Global Configuration Items

TeamCity displays a notification on the availability of the new TeamCity version and a prompt to upgrade. A warning is displayed if any of the licenses are incompatible with this new version. The notification is visible to system administrators only and they can use the link in the "Some Licenses are incompatible" message to quickly navigate to the [Licen](#)

ses page, where all incompatible licenses will have a warning icon.

## Agent Configuration

TeamCity displays a notification if agents are not running the recommended Java 8: this report shows all of the agents running under Java earlier than version 1.8.

## WebSocket connection issues

The WebSocket protocol is used to get web UI updated for events, running builds updates and statistics counters.

In case of any problems preventing WebSocket connection from working, a warning will be displayed. TeamCity will automatically switch to the legacy update mode (usual HTTP request polling used by TeamCity before version 9.0) and you will be able to continue using TeamCity in this mode.

 However, it is recommended to make the following adjustments to benefit from faster web UI updates as well as reduced unnecessary network traffic and latency:

- [Proxy Server Configuration](#)
- [BIO Connector Adjustment](#)

### Proxy Server Configuration

If a reverse proxy is used in front of the TeamCity server, it needs to be [configured](#) to support the WebSocket protocol.

All URLs used by browsers that do not support the WebSocket connection are listed in the corresponding health report.

### BIO Connector Adjustment

If Tomcat is configured to use the [BIO connector](#), the WebSocket protocol is [automatically disabled](#). It is recommended to change the Tomcat Connector settings to use the [NIO connector](#).

## Critical Errors

This category shows the following errors:

- errors in project configuration files - occur if the server detects some inconsistency or a broken configuration while it loads configuration files from the disk
- errors raised by [the disk space watcher](#)
- warnings from the TeamCity Server Memory Monitor

## Database Related Problems

TeamCity will warn you if the server currently uses the internal database. [A standalone database is recommended as the storage engine](#).

As [TeamCity does not support Sybase as an external database](#), a warning message will be displayed if you are using Sybase.

## Build Configuration Items

### Dependency problems

TeamCity detects build configurations dependent on a missing build configuration (e.g. on a build configuration deleted after the dependency was configured).

### Configurations with Large Build Logs

Large [Build Logs](#) (more than 200 MB by default) can reduce the server performance as they could be too heavy to parse and are hard to view in the browser.

The build script can be tuned to print less output if this inspection fails frequently for some Build Configuration.

### Inefficient Artifacts Publishing

TeamCity detects a build publishing many small artifact files and suggests publishing them as a single .zip archive.

## VCS Root Related Problems

### Unused VCS Roots

TeamCity will show you the [VCS roots](#) defined in a project and will offer to delete the unused roots.

### Similar VCS roots

TeamCity qualifies [VCS roots](#) as identical when their major settings (e.g. URLs, branch settings) are the same even if some of their settings (e.g. user name, password) are different.

The report will show you identical roots and will leave it up to you whether to merge them or not.

### Similar VCS Root Usages (AKA instances)

You can define values for [VCS root](#) settings or use parameter references to various parameters defined at different levels.

When the referenced VCS roots parameters are resolved to the same values as the values defined, such cases will be reported as identical VCS root usages.

The general recommendation is to use parameter references for root settings, thus optimizing the amount of VCS roots.

### Trigger Rules for Unattached VCS roots

TeamCity displays a warning if a rule of a [VCS Trigger](#) or [Schedule Trigger](#) references a VCS root which is not attached to any build configuration.

### Redundant Trigger

The report will show cases when a build trigger is redundant, for example:

- There are two build configurations A and B
- A snapshot depends on B
- Both have VCS triggers, A with the [Trigger on changes in snapshot dependencies](#) option enabled.

In this case, the VCS trigger in B is redundant and causes builds of A to be put into the queue several times.

### Multiple identical build triggers

The warning is displayed if there are two or more enabled triggers of the same type with identical sets of parameter values. Disabled triggers are not taken into account.

### Effective Quiet Period is Bigger Than Specified

When a [VCS trigger](#) for a build configuration has a quiet period, TeamCity will wait the specified time after the last detected change before triggering the build. During this time, all VCS Roots which affect this build configuration are checked for changes. If other VCS Roots have checking for changes interval bigger than the quiet period, the effective quiet period will be equal to the maximum checking for changes interval of the involved VCS Roots (it could be a VCS Roots from the dependencies).



#### Commit Hooks

When a VCS Root uses commit hooks, its checking for changes interval does not affect the quiet period and won't delay build triggering, see [TW-44865](#).

Possible fix could be one of the following:

- Use commit hooks to trigger checking for changes operations
- Increase quiet period in the VCS trigger so it is larger than checking for changes interval of the related VCS Roots
- Reduce checking for changes interval for the problematic VCS Roots

### VCS checkout

## Possible Frequent Clean Checkout

This section of the report will show possible frequent [clean checkout](#), which may be caused by the following two reasons:

- [Custom Checkout Directory](#)
- [Build Files Cleaner \(Swabra\) Settings](#)

### Custom Checkout Directory

Build configurations having different [VCS settings](#) but the same [custom checkout directory](#) may lead to frequent clean checkouts, slowing down the performance and hindering the consistency of incremental sources updates.

### Build Files Cleaner (Swabra) Settings

Enabling the [Build files cleaner \(Swabra\)](#) build feature in several build configurations may cause extra clean checkouts. This may happen if builds from these configurations alternately run on the same agents and have identical Version Control Settings or the same custom checkout directory specified.

Possible frequent clean checkout (Swabra case) server health report shows such incorrectly set up configurations grouped by Swabra settings.

## Optimal recommended checkout

A report is shown for large server-side patches with a recommendation to switch to the [agent-side checkout](#).

### Default auto-checkout on agent

If the default agent-side checkout is not possible, TeamCity will display a corresponding health report item and will use the server-side checkout.

## Integrations-related Items

- If a project or build configuration has secured parameters and is configured to build GitHub pull requests, this report will raise the warning, because malicious code submitted via the pull request can obtain these secured parameters
- If a VCS root points to GitHub or Bitbucket, a suggestion to configure a corresponding issue tracker is displayed.

## Agents Health

### Some agents cannot upgrade

The report helps to find agents which failed to upgrade. [Build agent logs](#) should help identify the cause of the issue.

### Cloud Agents

If a user removes an image from a profile, a warning is displayed that the instances already started by TeamCity will not be automatically stopped.

### Unused Build Agents

The report is displayed for the agents not used for 3 days and more, if

- you have more than 3 agents in your environment
- your agents have been registered for more than 3 days
- if the builds were run on the server during these 3 days

### Incorrect proxying server configuration

The report displays detected misconfiguration of the proxy server that is used to access the TeamCity web interface.

See our [recommendations](#) how to set up a proxy server with TeamCity.

## Suggested Settings

TeamCity analyzes the current settings of a build configuration and suggests additional options, e.g. adding a VCS trigger, a build step, etc. Besides the server health report, the suggestions for a specific build configuration are shown right on the

configuration settings pages.

## Extensibility

The default Server Health report provided by TeamCity might cover either too many, or not all the items required by you. Depending on your infrastructure, configuration, performance aspects, etc. that you need to analyze, a custom Server Health report can be needed. TeamCity enables you to write a [plugin](#) which will report on specific items.

# Build Time Report

Since TeamCity 9.0, the Build Time report is available providing comparative statistics of the build time taken up by TeamCity projects and build configurations.

To see the report, open the TeamCity Administration area and select Build Time from Project-related Settings section.

## Period Settings

The report displays total build duration for the server allowing you to select the period to be analyzed:

- the last 24 hours
- last week
- last month.

## Grouping and Sorting

You can group the build configuration by project to display the duration of builds for individual projects with the percentage in relation to the parent project. The scope can be further drilled down to the build time of an individual build configuration.

Without grouping the report shows the build time for individual build configurations with the percentage in relation to the total server build time. It is possible to include archived projects in the report.

Clicking the column headers allows sorting the report results.

# TeamCity Monitoring and Diagnostics

TeamCity provides a variety of diagnostic tools and indicators to monitor and troubleshoot the server, which are accessible from the Administration | Diagnostics page.

The tools make it easier to identify and investigate problems and, if needed, [report issues](#) on your server.

The following options are available to you:

- Troubleshooting
  - CPU & Memory Usage
  - Troubleshooting
    - Debug Logging
    - Hangs and Thread Dumps
    - Server Restart
  - Java Configuration
- VCS Status
- Server Logs
- Internal Properties
- Logging Presets
- Caches
- Search
- Browse Data Directory

## Troubleshooting

This tab provides a number of indicators helping you to detect and address issues with the TeamCity server performance.

### CPU & Memory Usage

This section displays the data provided by the [TeamCity Memory Monitor](#), which regularly checks available memory and submits a warning if the [memory](#) or [CPU usage](#) grows too high. See also information on configuring [memory settings](#) for the TeamCity server.

Depending on your operating system and Java settings, the list of displayed properties below may vary.

#### CPU usage

- TeamCity process CPU usage shows the CPU time used by the main TeamCity process averaged over a period of time. Other processes consuming CPU resources (databases, Maven server, VCS) are not included.
- Garbage collection shows the time spent on cleaning the server memory averaged over a period of time. High numbers over long periods of time usually indicate insufficient server memory.
- System load displays CPU load information based upon the CPU queue length averaged over a period of time, i.e. the sum of the number of runnable entities queued to the available processors and the number of entities running on the available processors averaged over a period of time (1 minute)
- Overall CPU usage shows the recent CPU usage for the whole system with a value in the [0.0,1.0] interval. A value of 0.0 means that all CPUs were idle during the recent period of time, while a value of 1.0 means that all CPUs were actively running 100% of the time during the recent period

#### Memory usage

- Total heap displays the total amount of memory used by TeamCity to store all data, including temporary data about to be collected. Represents all heap memory pools (young and old generations) in terms of java. Garbage collection runs quickly and frequently.
- Data shows the amount of memory used by TeamCity to store persistent data. Represents old generations only in terms of java. Garbage collection runs slowly and infrequently.

## Troubleshooting

### Debug Logging

This allows changing the internal TeamCity server [logging settings](#).

### Hangs and Thread Dumps

The [server thread dump](#) can be viewed in the browser or saved to a file.

If you experience memory problems, this section provides an option to dump a memory snapshot.

### Server Restart

This section available since TeamCity 2017.2 allows restarting the server from the UI.

## Java Configuration

This section informs you on the Java installed on your server and the configured JVM options.

## VCS Status

This tab displays the information on the monitored VCS roots, including their checking for changes status and duration. You can filter the available VCS roots by the checking for changes duration.

## Server Logs

This tab allows you to view and download the available TeamCity server logs, as well as saved thread dumps and memory dumps.

## Internal Properties

This tab displays the internal properties affecting the TeamCity behavior and the JVM system properties. To fine-tune these aspects, see [this section](#).

## Logging Presets

TeamCity uses the `log4j` library for [logging](#) internal server activities. In this section you can view and download the available presets, as well as upload new presets, which can then be enabled on the [Diagnostics | Troubleshooting | Debug Logging](#).

It is also possible to change the logging configuration [manually](#).

## Caches

This tab shows you the caches of the TeamCity processes stored in [TeamCity Data Directory](#)/system/caches. Resetting some caches is performed by the server during the clean-up automatically, but sometimes you might need to clear caches manually using the [reset](#) link.

- `vcsContentCache` - TeamCity maintains `vcsContentCache` cache for the sources to optimize communications with the VCS server. The caches are reset during the cleanup time. To resolve problems with sources update, the caches may need to be reset manually.
- `search` - resetting this cache is required when enabling [search by build log](#).
- `git` - contains the bare clone of the remote [Git](#) repository used by TeamCity.
- `buildsMetadata` - resetting this cache is required to [reindex the TeamCity NuGet feed](#).

## Search

Displays information on the TeamCity data index related to the [search](#).

## Browse Data Directory

This tab shows the files in the [TeamCity Data Directory](#) and allows you to upload new files.

## Uploading SSL Certificates

It is possible to upload an SSL certificate which TeamCity considers trusted when establishing connection by HTTPS or SSL protocols. These can be self-signed certificates or certificates signed by a not well-known certificate authority (CA).

### Adding trusted certificates to TeamCity server

The trusted certificates storage is global for the whole server and affects all of the server projects.

To add a trusted certificate

1. Navigate to the Root project Administration area and select the SSL / HTTPS Certificates menu item in the sidebar

2. Click Upload certificate, specify the certificate name and choose a certificate file of one of the supported formats: PEM, DER or PKCS#7.
3. Save your changes.

## Delivering certificates to TeamCity agents

All uploaded certificates will be automatically delivered to all TeamCity agents.

However, sometimes automatically distributing certificates to all agents may not be needed or may be undesirable. Then you can manually add certificates to a required agent by placing them into the [<TeamCity Agent Home>/conf/trustedCertificates](#) folder.

This can be useful in the following cases:

- If the user is running the TeamCity server under a non-trusted certificate, you need to place the server certificate into this folder on an agent to establish agent-server connection
- If the user considers their network connection between the server and agents insecure and does not want to transfer sensitive information.

## Several Nodes Setup

TeamCity allows configuring and starting another server instance in addition to the main one. Both servers will share the TeamCity data directory and the database. Using several node setup, you can:

- improve the general TeamCity performance:  
configure the [Running Builds Node](#), which handles all of the data produced by running builds (build logs, artifacts, statistic values) while the main server will handle all other tasks
- set up a High-Availability TeamCity installation that will have zero read downtime:  
configure the [Read-Only Node](#) allowing users read operations during the downtime of the main server, e.g. during the upgrade.

## Configuring Read-Only Node

On this page:

- [Read-Only Node Overview](#)
- [Starting Read-Only Node](#)
  - [Proxy Configuration](#)
- [Startup & Shutdown](#)
- [Upgrade](#)

Since TeamCity 2018.1 TeamCity makes a step towards High Availability by introducing a read-only mode for the server. It is currently possible to start a second, read-only TeamCity server looking at the same database and data directory as the main one.

With read-only node it is possible to set up a High-Availability TeamCity installation that will have zero read downtime, i.e. when the main server is unavailable or is performing an upgrade, requests will be routed to the read-only node. Such setup requires installing both the main server and the read-only node behind a reverse proxy that should be configured to route requests to the main server while it's available and to the read-only one in other cases.

The sections below details the read-only node and proxy configuration.

### Read-Only Node Overview

The same TeamCity server distribution is used for the main and read-only nodes, the role is selected based on the startup settings. The primary setup is running the two server nodes on different machines.

An additional server in a High-Availability set-up uses the license from the main server and does not require a separate license.

When a TeamCity server is started as the read-only node, it allows users read operations: view the build information, download artifacts, etc. during the downtime of the main server, e.g. during upgrade. It is recommended that both the main TeamCity server and the read-only node should be of exactly the same version. However, the main server and the read-only one can be running different versions when the main server is upgraded. See the [Upgrade](#) section below.

The read-only node has a number of limitations compared with the main server:

- Not all the pages in the Administration area are available in the read-only mode. For example, the Projects Import tab is not shown as it's not useful when the server can't perform write operations.
- Currently, only bundled and a limited set of some other plugins are loaded by the read only server, so some functionality provided by external plugins can be missing.
- Currently, it's not possible to switch the read-only node to the write mode. So in case of the main server failure, a new main server should be started.
- Users may need to re-login when they are routed to the read-only node if they didn't use Remember Me.

## Starting Read-Only Node

A Read-Only node needs to access the same [TeamCity Data Directory](#) and the same external database as the main TeamCity Server.

The typical data directory mounting options are SMB and NFS. TeamCity uses the data directory as a regular file system so all the basic file system operations should be supported. The backing storage and way of mounting should not impose strict IO operations count or IO volume limits and should provide performance like that of a local drive.

Besides the data directory shared with the main server, the Read-Only node requires a 'local' data directory where it stores some caches, unpacked external plugins and some other configuration. By default, the 'local' data directory is automatically created under the <Main Data Directory>/nodes/read\_only\_node path during the first start. You can override its location using -Dteamcity.node.data.path property in the TeamCity [start-up scripts](#).

To start a Read-Only node along with the main TeamCity Server, do the following:

1. [Install](#) the TeamCity software as usual: download a distribution package, unpack it or follow the installation wizard.
2. Configure the location of the [TeamCity Data Directory](#) on the Read-Only Node, make sure it points to the shared data directory.
3. Add additional arguments to the [TEAMCITY\\_SERVER\\_OPTS](#) environment variable: TEAMCITY\_SERVER\_OPTS=-Dteamcity .server.role=read-only-server

## Proxy Configuration

The configurations examples below will proxy requests to the main server while it's available and to the read-only in other cases. If you did not setup TeamCity server behind reverse proxy before, make sure to review our [notes](#) on this topic.

For example, the following [NGINX](#) configuration will route requests to the read-only node only when the main server is not available or when the main server responds with 503 status code, which generally means that it is starting or is performing an upgrade.

```
http {
    upstream backend {
        server teamcity-main.local:8111 max_fails=0; # full internal address of the main server;
        server teamcity-ro.local:8111 backup; # full internal address of the read-only node;
    }

    server {
        location / {
            proxy_pass      http://backend;
            proxy_next_upstream error timeout http_503 non_idempotent;
        }
    }
}
```

Note that [NGINX Open Source](#) does not support active [health checks](#) (which is a better way to configure High Availability installation) and may experience DNS resolution issues. Consider using [NGinx Plus](#) or [HA Proxy](#).

[HA Proxy configuration example](#):

```

resolvers dns
  nameserver dns %IP%:%PORT%
  resolve_retries    3
  timeout resolve    1s
  timeout retry      1s
defaults
  mode http
frontend http-in
  bind *:80
  default_backend tc-ha
backend tc-ha
  option httpchk GET /login.html
  default-server inter 2s fall 2 rise 2
  server tc_main_node %TC_MAIN_NODE_URL:8111% check resolvers dns
  server tc_ro_node %TC_RO_NODE_URL:8111% backup check resolvers dns

```

## Startup & Shutdown

Both the main TeamCity server and the read-only node can be started / stopped using regular TeamCity scripts (teamcity-server.bat, teamcity-server.sh) or Windows services.

The read-only Node server uses the same approach to logging as the main server. You can check how the startup is doing in the <TeamCity installation directory>/logs/teamcity-server.log file. You can also open <read-only node URL> in your browser, there you should see the regular TeamCity startup screens.

## Upgrade

Both the main TeamCity server and the Read-Only Node should be of exactly the same version. However, the main server and the read-only one can be running different versions, e.g. when the main server has been upgraded. When the versions of the read-only node and the main server are different, the corresponding health report will be displayed on both nodes.

When upgrading to a minor version (a bugfix release), both servers should be running without issues as the TeamCity data has the same format.

When upgrading the main server to the major version, the TeamCity data format will be changed and data upgrade will be performed while the read-only node is running.

After the main server is upgraded to a new major version, the read-only node shows a health report when it detects that the data was upgraded and stops processing new events, i.e. will not show new builds. The read-only node must be upgraded after the main server upgrade to be able to process new events.

TeamCity agents will perform upgrade / automatically.

## Configuring Running Builds Node

It is possible to use a separate TeamCity server instance to process traffic coming from the TeamCity agents. This allows moving the related load to a separate machine from the main TeamCity server and improve general performance of the TeamCity server when handling hundreds of concurrent, actively logging running builds. In general, you do not need the two-node installation unless you have more than 400 agents connected to a single server. Using the running builds node allows you to significantly increase the amount of agents which the setup can handle.

On this page:

- Overview of two-node configuration
- Shared Data Directory
- Installation
- Startup & shutdown
- Enabling Running Builds Node
- Restarting Servers while Builds are Running
- Upgrade & Downgrade
- Cleanup
- Backup & Restore
- Current limitations

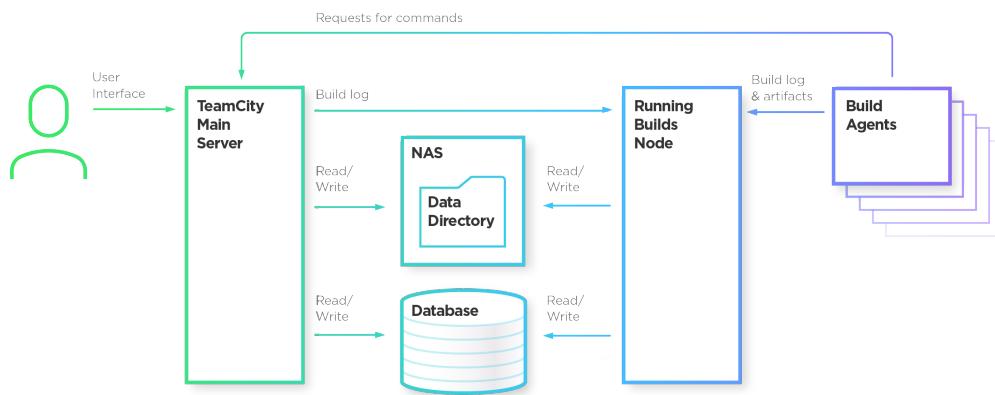
## Overview of two-node configuration

The same TeamCity server distribution is used for the main and running builds nodes, the role is selected based on the startup settings. When a TeamCity server is started as the running builds node, its functionality is reduced to only one task - process real time data coming from the running builds. The primary setup is running the two server nodes on different machines.

Several important notes about two-node configuration:

- the server started as the Running Builds Node preprocesses it and stores it in the database and on the disk in the TeamCity Data Directory
- in a two-node installation the main TeamCity server handles all other tasks: user interface, VCS-related activity, management of agents, scheduling builds on agents, etc.
- both the main TeamCity server and the Running Builds Node require access to the same [data directory](#) (which must be shared) and to the same database
- firewall settings should allow access to Running Builds Node from the agents and from the main TeamCity server (the main TeamCity server also communicates with the running builds node by HTTP)

## Two-Node Configuration



### Shared Data Directory

Ensure that both machines where TeamCity server nodes will be installed can access the same [TeamCity data directory](#) in the read / write mode. Using dedicated network file storage with good performance is recommended.

Note that Running builds node mostly writes data under TeamCity data directory, while main TeamCity server can do both read / write operations, and also scan directories during cleanup. So performance of file operations can be more important for the main server. As such to achieve better performance main server can either reside on the machine where data directory is located, or can use some kind of block level data storage (SAN). The same data directory can be then shared with Running builds node by SMB or NFS.

Tuning storage and network performance is recommended: make sure to review performance guidelines for your storage solution. For example, increasing MTU for the network connection between the server and the storage usually has positive effect on artifacts transfer speed.

### Installation

1. On both machines, [install](#) TeamCity software as usual: download a distribution package, unpack it or follow the installation wizard.
2. Configure the TEAMCITY\_DATA\_PATH environment variable on both machines, make sure it points to the shared data directory.
3. On the Running Builds Node machine, add additional arguments to the TEAMCITY\_SERVER\_OPTS environment variable:  

```
export TEAMCITY_SERVER_OPTS=-Dteamcity.server.role=running-builds-node -Dteamcity.server.rootURL=<node url> <your regular options if you have them>, where <node url> is the URL of the Running Builds node.
```

This URL must be accessible by both the agents and main server. If you do not have an HTTP proxy installed in front of the TeamCity servlet container and you did not change the port in the servlet container during the installation, then by default this URL will be: `http://<your host name>:8111/`

### Startup & shutdown

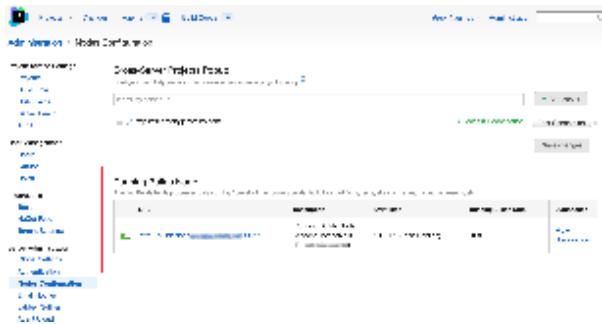
Both the main TeamCity server and Running Builds Node can be started / stopped using regular TeamCity scripts (`teamcity-se`

rver.bat, teamcity-server.sh) or Windows services.

The Running Builds Node server uses the same approach to logging as the main server. You can check how the startup is doing in the <TeamCity installation directory>/logs/teamcity-server.log file. You can also open <Running Builds Node URL> in your browser, there you should see regular TeamCity startup screens.

## Enabling Running Builds Node

When both the main server and Running Builds Node are up and running, the Administration | Nodes Configuration page on the main TeamCity server displays the Running Builds Node:



By default, the running builds node is disabled and all traffic produced by running builds is still handled by the main server. Once you enable the running builds node, all newly started builds will be routed to this node. The existing running builds will continue being executed on the main server.

And vice versa, when the running builds node has been disabled, only the newly started builds will be switched to the main server, the builds that were already running on the second node will continue running there.

The Running Builds Node on the Administration | Nodes Configuration page displays how many builds are currently assigned to this node. After a Running Builds Node is enabled, eventually all of the running builds will be switched to it.

## Restarting Servers while Builds are Running

The Running Builds Node as well as the main TeamCity server can be stopped or restarted while builds are running there. If agents cannot connect to the running builds node for some time, they will reroute their data to the main server. If the main server is unavailable either, agents will keep their data and resend it once the servers reappear.

## Upgrade & Downgrade

Both the main TeamCity server and the Running Builds Node must be of exactly the same version.

To upgrade:

1. stop the running builds node (if you forget to do that, you'll be warned during upgrade)
2. start [upgrade](#) on the main TeamCity server as usual
3. proceed with upgrade
4. check everything is OK, agents are connecting etc. (since the running builds node is not available anymore, the agents will reroute their data to the main server)
5. upgrade TeamCity installation on the running builds node to the same version
6. start the running builds node and check that it is connected using the Nodes Configuration page on the main server

To downgrade:

1. shutdown the main server and the running builds node
2. [restore data](#) from backup (only if the data format has been changed during upgrade)
3. downgrade TeamCity software on the main server
4. start the main TeamCity server and check that everything works
5. downgrade TeamCity software on the the running builds node to the same version as the main server
6. start the the running builds node

As usually with TeamCity, agents perform upgrade / downgrade automatically.

## Cleanup

The TeamCity [cleanup](#) task runs on the main TeamCity server only. In a two-node configuration as well as in a single node configuration, it can work at the same time while the servers are handling data from the running builds.

## Backup & Restore

[Backup](#) through TeamCity web interface can be done on the main TeamCity server only. Backup from the command line can be done on either the running builds node or on the main TeamCity server.

Restore can be done on either of these nodes, but only if all of the servers using the TeamCity database and data directory are stopped.

## Current limitations

- Only one running builds node can be configured for the main TeamCity server.
- If you use non-bundled TeamCity plugins, some fraction of them might be not compatible with the running builds node setup as few TeamCity API endpoints stop working in the setup (only the obsolete API related to build message processing listeners is affected). Also, there is no possibility to write plugins specific to the running builds node.

## Configuring Cross-Server Projects Popup

On this page:

- [Configuring Popup](#)
- [CORS Configuration](#)
- [Server Versions Compatibility](#)

TeamCity Projects popup allows browsing projects and build configurations on multiple servers, so that when a project or a build configuration is selected, the page is opened on the right server.

The projects popup uses separate REST API requests to get the list of projects to display. If peer servers are configured, REST API calls are made to them as well.

For these REST API calls to work, the servers [need to be configured](#) to allow CORS requests from all of their peers. Besides, the must be accessible from the user's browser and the user must be logged in on those servers.

### Configuring Popup

The dedicated UI on the Administration | Nodes configuration page allows configuring linked servers feeding the projects popup:

1. Specify the TeamCity server URL and click Add Server.
2. Click the test connection button. You have to be logged in to do that.

If the connection is successful, you will see the corresponding server node added to the projects popup.

### CORS Configuration

Provided a popup on the server A is to display projects from the server B, Server B is required to have CORS configured to trust all the URLs of server A which can be used by the users to access server A.

When configuring the popup on both servers, they need to have CORS configured to trust all the URLs of one another. If a third server is added, it has to be added to the other two.

### Server Versions Compatibility

It is possible to configure the cross-server projects popup between servers running different TeamCity versions.

## Managing Projects and Build Configurations

Projects and their entities in TeamCity can be configured:

- via the web UI
- using the [REST API](#)
- programmatically using DSL based on the [Kotlin language](#) if the [versioned settings](#) feature is enabled

### Configuring Visibility

The Configure Visible Projects pop-up on the Projects Overview page allows managing projects display.

An individual project can be added to / removed from the Projects Overview page using the "eye" button on the Project Home page:

The screenshot shows the TeamCity web interface. At the top, there's a navigation bar with links for 'Projects', 'Changes', 'Agents', 'Build Queue', 'Administration', and a search bar. Below the navigation is the project header for 'Sample Project' by 'Julia Alexandrova'. The main content area has tabs for 'Overview', 'Change Log', 'Statistics', 'Current Problems', 'Investigations', 'Muted Problems', and 'Flaky Tests'. A 'Run ...' button is on the right. The 'Build' tab is currently active. In the top right of the header, there's a button with an eye icon and the text 'Hide Successful Configurations'. A blue box highlights this button. Below it is a link 'Click to show project on the overview page'.

Note that if a project has [archived subprojects / paused build configurations](#), they will also be displayed on the overview page and will be marked correspondingly.

In this section:

- [Creating and Editing Projects](#)
- [Configuring VCS Settings](#)
- [Storing Project Settings in Version Control](#)
- [Creating and Editing Build Configurations](#)
- [Triggering a Custom Build](#)
- [Copy, Move, Delete Build Configuration](#)
- [Ordering Projects and Build Configurations](#)
- [Archiving Projects](#)
- [Ordering Build Queue](#)
- [Muting Test Failures](#)
- [Changing Build Status Manually](#)
- [Customizing Statistics Charts](#)
- [Configuring Artifacts Storage](#)

See also:

[Concepts: Project | Build Configuration](#)

## Creating and Editing Projects

This section details creating projects via the TeamCity web UI. Other options include the [REST API](#) and using TeamCity project configuration in [DSL based on the Kotlin language](#).

On this page:

- [Creating Project](#)
- [Managing Project](#)
  - [Copying Project](#)
  - [Moving Project](#)
  - [Archiving Project](#)
  - [Bulk Editing IDs](#)
  - [Pausing / Activating Triggers](#)
  - [Exporting Project](#)
  - [Deleting Project](#)

### Creating Project

To create a project, use the Administration link at the top right corner and click Create project. The Create project page is displayed.

There are several options to create a project:

- [From a repository URL](#)
- [From GitHub.com](#)
- [From Bitbucket Cloud](#)
- [From Visual Studio Team Services](#)
- [Manually](#)

To create a subproject, go to the parent project settings page and create a subproject using one of the available options.

### Creating project pointing to repository URL

1. On the Create project page, select to create project from a repository URL.
2. Specify the project settings:

Setting	Description
Parent Project	Select the parent project form the drop-down.
Repository URL	A VCS repository <a href="#">URL</a> . TeamCity recognizes URLs for Subversion, Git and Mercurial. TFS and Perforce are partially supported.
Username	Provide username if access to repository requires authentication
Password	Provide password if access to repository requires authentication

3. Click Proceed.

TeamCity will configure the rest of settings for you.

- it will determine the type of the VCS repository, auto-configure VCS repository settings, and suggest the project and build configuration names:
- the project, build configuration and VCS root will be created automatically
- TeamCity will add a VCS build trigger.
- TeamCity will attempt to auto-detect build steps: Ant, NAnt, Gradle, Maven, MSBuild, Visual Studio solution files, PowerShell, Xcode project files, Rake, and IntelliJ IDEA projects.

4. On the Auto-detected Build Steps page select the detected step(s) to use in your build configuration. Click Use selected.

If no steps found, you will have to [configure build steps manually](#).

5. Your project and a build configuration are configured. Click the Run button to start the build.

 Depending on the build configuration settings, TeamCity can suggest some additional configuration options. Review Suggestions at the end of the settings list and configure required ones.

### Creating project pointing to GitHub.com repository

1. On the Create project page, select to create project from GitHub.com.
    - If you do not have a GitHub connection configured, you will be redirected to the Connections page. Set up the connection as [described here](#), then follow the steps below.
    - If you have a GitHub connection configured, follow the steps below.
  2. Select a repository. TeamCity will verify the repository connection. If the Connection is verified, the new page opens.
  3. TeamCity will display the project and build configuration name. If required, modify the names and click Proceed.
  4. TeamCity will add a VCS build trigger and attempt to auto-detect build steps: Ant, NAnt, Gradle, Maven, MSBuild, Visual Studio solution files, PowerShell, Xcode project files, Rake, and IntelliJ IDEA projects.
- On the Auto-detected Build Steps page select the detected step(s) to use in your build configuration. Click Use selected.

If no steps found, you will have to [configure build steps manually](#).

5. Your project and a build configuration are configured. Click the Run button to start the build.

 Depending on the build configuration settings, TeamCity can suggest some additional configuration options. Review Suggestions at the end of the settings list and configure required ones.

### Creating project pointing to Bitbucket Cloud

1. On the Create project page, select to create project from Bitbucket Cloud.
    - If you do not have a Bitbucket connection configured, you will be redirected to the Connections page. Set up the connection as [described here](#), then follow the steps below.
    - If you have a Bitbucket connection configured, follow the steps below.
  2. Select a repository. TeamCity will verify the repository connection. If the Connection is verified, the new page opens.
  3. TeamCity will display the project and build configuration name. If required, modify the names and click Proceed.
  4. TeamCity will add a VCS build trigger and attempt to auto-detect build steps: Ant, NAnt, Gradle, Maven, MSBuild, Visual Studio solution files, PowerShell, Xcode project files, Rake, and IntelliJ IDEA projects.
- On the Auto-detected Build Steps page select the detected step(s) to use in your build configuration. Click Use selected.

If no steps found, you will have to [configure build steps manually](#).

5. Your project and a build configuration are configured. Click the Run button to start the build.

 Depending on the build configuration settings, TeamCity can suggest some additional configuration options. Review Suggestions at the end of the settings list and configure required ones.

## Creating project pointing to Visual Studio Team Services

1. On the Create project page, select to create project from Visual Studio Team Services.
  - If you do not have a Visual Studio Team Services connection configured, you will be redirected to the Connection s page. Set up the connection as [described here](#), then follow the steps below.
  - If you have a Visual Studio Team Services connection configured, follow the steps below.
2. Select a repository. TeamCity will verify the repository connection. If the Connection is verified, the new page opens.
3. TeamCity will display the project and build configuration name. If required, modify the names and click Proceed.
4. TeamCity will add a VCS build trigger and attempt to auto-detect build steps.  
On the Auto-detected Build Steps page select the detected step(s) to use in your build configuration. Click Use selected.  
If no steps found, you will have to [configure build steps manually](#).
5. Your project and a build configuration are configured. Click the Run button to start the build.  
 Depending on the build configuration settings, TeamCity can suggest some additional configuration options. Review Suggestions at the end of the settings list and configure required ones.

## Creating project manually

1. Click the Create project button and select Manually.
2. On the Create New Project page, specify the project settings:

Setting	Description
Parent Project	Select the parent project form the drop-down.
Name	The project name.
Project ID	the <a href="#">ID</a> of the project
Description	Optional description for the project.

3. Click Create. An empty project is created.

 To configure an existing project, select the desired project in the list, and click the Edit Project Settings link on the right.

4. [Create build configurations](#) (select build settings, [configure VCS settings](#), and choose [build runners](#)) for the project.
5. [Assign build configurations to specific build agents](#).

## Managing Project

You can view all available projects and subprojects on the Projects Overview page listed in the alphabetical order by default. Administrators can [customize the default order](#).

To copy, move, delete or [archive](#) a project, use the Actions menu in the top right of the project settings page or the more button  next to the project on the parent project settings page. These options are not available for the Root project.

## Copying Project

Use the corresponding item from the Actions menu in the top right of the project settings page or the more button  next to the project on the parent project settings page.

Projects can be copied and moved to another project by project administrators.

A copy duplicates all the settings, [subprojects](#), [build configurations](#) and templates of the original project, but no data related to builds is preserved. The copy is created with the empty [build history](#) and no [statistics](#).

You can copy a project into the same or another parent.

On copying, TeamCity automatically assigns a new name and [ID](#) to the copy. It is also possible to change the name and ID manually.

Selecting the Copy project-associated user, agent and other settings option makes sure that all the settings like notification rules or agent's compatibility are exactly the same for the copied and original projects for all the users and agents affected.

You can also opt to copy build configurations build numbers.

 When running TeamCity in the [Professional mode](#), the Copy option will not be displayed for a project if the number of build configurations on the server after copying will exceed the limit (100 build configurations since TeamCity 2017.2)

and 20 in earlier versions unless you purchased additional Build Agent licenses).

## Moving Project

 Before moving the project, consider the following:

- TeamCity assigns user roles on a [per-project](#) basis, which means that moving a project may result in changing the scope of user permissions in the new project (new permissions may be added or the existing permissions can be dropped)
- Connection to Git VCS Roots containing SSH keys may get unavailable after a project move

To move a project, use the corresponding item from the Actions menu in the top right of the project settings page or the more button  next to the project on the parent project settings page.

When moving a project, TeamCity preserves all its settings, [subprojects](#), [build configurations/templates](#) and associated data, as well as the [build history](#).

## Archiving Project

Use the corresponding item from the Actions menu in the top right of the project settings page or the more button  next to the project on the parent project settings page. Please refer to the dedicated [page](#).

## Bulk Editing IDs

 Care must be taken when performing this action. Modifying the ID will change all the URLs related to the project. It is highly recommended to update the ID in any of the URLs bookmarked or hard-coded in the scripts. The corresponding configuration and artifacts directory names on the disk will change too and it can take time.

1. Use the corresponding item from the Actions menu in the top right of the project settings page or the more button  next to the project on the parent project settings page.
2. The current project and build configuration [IDs](#) are displayed. You can modify or reset the IDs for all subproject, VCS roots, build configurations and templates. Click Regenerate to get new IDs automatically or edit them manually.
3. Click Submit.

## Pausing / Activating Triggers

You can [pause triggers](#) for all or selected build configurations of a project. Use the corresponding item from the Actions menu in the top right of the project settings page or the more button  next to the project on the parent project settings page.

## Exporting Project

You can [export configuration files](#) of a project with its children to move it to a different TeamCity server. Use the corresponding item from the Actions menu in the top right of the project settings page or the more button  next to the project on the parent project settings page.

## Deleting Project

Use the corresponding item from the Actions menu in the top right of the project settings page or the more button  next to the project on the parent project settings page.

When you delete a project, TeamCity will remove its .xml configuration files. After the deletion, the project is moved to the `<TeamCity Data Directory>/config/_trash/.ProjectID.projectN` directory. There is a [configurable](#) timeout (5 days by default) before all project-related data stored in the database (build history, artifacts, etc.) of the deleted project is completely removed during the next build history clean-up. You can [restore](#) a deleted project before the clean-up is run.

The `<TeamCity Data Directory>/config/_trash/` directory is not cleaned automatically and can be emptied manually if you are sure you do not need the deleted projects.

 If you attempt to delete a project with [dependent build configurations](#) from other projects, TeamCity will warn you about it. If you proceed with the deletion, the dependencies will no longer function.

See also:

[Administrator's Guide: Creating and Editing Build Configurations](#)

## Configuring VCS Settings

On this page:

- [VCS Settings Overview](#)
- [Attach VCS Root](#)
  - [Configure Checkout Rules](#)
- [Configuring Checkout Options for Build Configuration](#)
  - [Checkout Settings](#)
  - [Changes Calculation Settings](#)
- [Disable Building in Default Branch](#)
- [Other VCS-Related Settings](#)

### VCS Settings Overview

A Version Control System (VCS) is a system for tracking the revisions of the project source files. It is also known as SCM (source code management) or a revision control system. The following VCSs are supported by TeamCity out-of-the-box: [Git](#), [Svntools](#), [Mercurial](#), [Perforce](#), [Team Foundation Server](#), [CVS](#), [StarTeam](#), [ClearCase](#), [SourceGear Vault](#), [Visual SourceSafe](#).

Connection to a version control system is defined by a TeamCity [VCS root](#). A project or a [build configuration](#) in TeamCity can have one or more VCS roots attached; a build configuration and also defines other checkout options like [Checkout Rules](#) - these define the workspace for the build.

TeamCity always monitors the repositories from the server-side to detect changes and display them in the UI. Depending on the specified [VCS Checkout Mode](#) the actual repository checkout can also happen on the agent-side.

TeamCity performs VCS-related operations per each VCS root separately, thus it is advised to reuse VCS roots with same settings.

When [parameter references](#) are used in a VCS root, TeamCity performs VCS-related operations per each "VCS root instance", where "instance" is a unique set of VCS root parameters after references resolution. Adding parameters to the VCS roots does not reduce the number of VCS operations performed, it just allows sharing settings more effectively.

### Attach VCS Root

VCS settings are configured on the Version Control Settings page for a project or a build configuration: you can attach an

existing [VCS root](#) to your project/build configuration, or create a new one to be attached. This is the main part of VCS parameters setup; a VCS Root is a description of a version control system where project sources are located. Learn more about VCS Roots and configuration details [here](#).

## Configure Checkout Rules

When several VCS roots are attached or you need to checkout only a portion of the repository, specify the [checkout rules](#) for the VCS root to provide advanced possibilities to control sources checkout. With the rules you can exclude and/or map paths to a different location on the Build Agent during checkout.

## Configuring Checkout Options for Build Configuration

### Checkout Settings

Setting	Description
VCS Checkout Mode	To define how project sources reach an agent, use the <a href="#">VCS Checkout Mode</a> options.
Checkout Directory	The <a href="#">build checkout directory</a> is a directory on the TeamCity agent machine where all of the sources of all builds are checked out into.
Clean build	Define whether you want to clean all files in the checkout directory before the build. See <a href="#">Clean Checkout</a> for details.

### Changes Calculation Settings

 This section, formerly called the Display settings, was renamed into Changes calculation settings since TeamCity 2017.1.

Setting	Description
Show changes from snapshot dependencies	Configure whether TeamCity will show changes from snapshot dependencies. This also affects treatment of pending changes in schedule trigger.
Exclude default branch changes from other branches (since TeamCity 2017.1)	<p>By default, when displaying pending changes in a feature branch, TeamCity includes changes in the default branch as well. This allows tracking the cases when a commit that broke a build was fixed in the default branch, but not in a feature branch.</p> <p>However, for large projects with multiple teams simultaneously working on lots of different branches this means that all the project committers (regardless of the branch they are committing to) will be notified when, for example, a commit in the default branch broke the build or if a force push was performed.</p> <p>If you want to see the changes in a feature branch only, check the box to exclude changes in the default branch from being displayed in other branches.</p>

### Disable Building in Default Branch

By default, TeamCity will run builds in the default branch: the Default Branch Settings section, available since TeamCity 2017.1, has the option Allow builds in the default branch enabled.

If this behavior is undesirable, you can uncheck this box and the default branch will not be shown in the UI. This can be useful if you want to build pull requests only.

 Note that unchecking the option will affect all the aspects concerning the default branch, including:

- build chains: if a chain is triggered in a branch, and this branch does not exist in one of the configurations that is a part of the chain, previously TeamCity fell back on the default branch. If the default branch in this configuration is not allowed, building the dependency will fail
- the behavior of the Run dialog: the run custom build dialog will be displayed asking to select a branch
- triggers and notifications.

## Other VCS-Related Settings

- Configure [VCS trigger](#) if you want the build to be started on new changes detection.
- Additionally, you can add a label into the version control system for the sources used for a particular build by means of [VCS Labeling](#) build feature.

See also:

[Administrator's Guide: Configuring VCS Roots | VCS Checkout Rules | VCS Checkout Mode | VCS Labeling](#)

## Configuring VCS Roots

### VCS Roots in TeamCity

A VCS root defines a [connection to a version control system](#) and consists of a set of settings (paths to sources, username, password, and other settings) that defines how TeamCity communicates with a version control (SCM) system to [monitor changes](#) and get sources for a build. A VCS root can be attached to a build configuration or template. You can add several VCS Roots to a build configuration or a template and specify portions of the repository to checkout and target paths via [VCS Checkout Rules](#).

VCS roots are created in a project and are available to all the Build Configurations defined in that project or its [subprojects](#).

You can view all VCS roots configured within the project and create/edit/delete/detach them using the VCS Roots page under the project settings in the Administration UI.

If someone attempts to modify a VCS root that is used in more than one project or build configuration, TeamCity will issue a warning that the changes to the VCS root could potentially affect other projects or build configurations. The user is then prompted to either save the changes and apply them to all the affected projects and build configurations, or to make a copy of the VCS root to be used by either a specific build configuration or project.

On an attempt to create a new VCS root, TeamCity checks whether there are other VCS roots accessible in this project with similar settings. If such VCS roots exist, TeamCity suggests using them.

Once a VCS root is configured, TeamCity regularly queries the version control system for new changes and displays the changes in the [Build Configurations](#) that have the root attached. You can set up your build configuration to trigger a new build each time TeamCity detects changes in any of the build configuration's VCS roots, which suits most cases. When a build starts, TeamCity gets the changed files from the version control and applies the changes to the [Build Checkout Directory](#).

### Common VCS Root Properties

Property	Description
Type of VCS	Type of Version control system supported by TeamCity, for example, Perforce, Subversion, etc
VCS root name	Unique name of VCS root across all VCS roots of the project.
VCS root ID	Unique <a href="#">ID</a> of VCS root across all VCS roots in the system. VCS root ID can be used in parameter references to VCS root parameters and <a href="#">REST API</a> . If not specified, will be generated automatically from VCS root parameters.
Repository URL	URL to VCS repository. Supports URLs in <a href="#">different formats</a> , like: <code>http(s)://</code> , <code>svn://</code> , <code>ssh://git@</code> , <code>git://</code> and others as well as URLs in Maven format.

Minimum checking interval	<p>Specifies how often TeamCity polls the VCS repository for VCS changes. By default, the global predefined server setting is used that can be modified on the Administration   Global Settings page. The interval time starts as soon as the last poll is finished on the per-VCS root basis. Here you can specify a custom interval for the current VCS root.</p> <p> Some public servers may block access if polled too frequently.</p> <p>If TeamCity detects that a <a href="#">VCS commit hook</a> is used to trigger checking for changes, this interval is automatically increased up to the predefined value (4 hours). If the periodical check finds changes undetected via the commit hook, the checking interval is reset to the specified minimum.</p>
Belongs to project	Each VCS root belongs to some project, and in this section the name of this project is displayed. A VCS root can be moved to the common parent project of all subprojects, build configurations and templates where the root is currently used.

Please refer to the following pages for VCS-specific configuration details:

- [Guess Settings from Repository URL](#)
- [ClearCase](#)
- [CVS](#)
- [Git](#)
- [Mercurial](#)
- [Perforce](#)
- [StarTeam](#)
- [Subversion](#)
- [Team Foundation Server](#)
- [SourceGear Vault](#)
- [Visual SourceSafe](#)
- [Configuring VCS Post-Commit Hooks for TeamCity](#)

 Make sure to synchronize the system time at the VCS server, TeamCity server and TeamCity agent (if agent-side checkout is used) if you use the following version controls:

- CVS
- StarTeam (if the audit is disabled or the server version is older than 9.0).
- Subversion repositories connected through externals to the main repository defined in the VCS root.
- VSS (all VSS [clients](#) and TeamCity server should have synchronized clocks)

## Guess Settings from Repository URL

TeamCity can automatically discover the VCS type and settings from the repository URL.

When configuring a [VCS root](#), select the Guess from Repository URL option from the drop-down and specify the URL. TeamCity will recognize the URL for a supported version control and will create a VCS root automatically. After the VCS Root is created, you can [modify its settings](#) using the Project Settings or Build Configuration Settings page.

## VCS URL Formats

VCS	URL Formats
Git	<ul style="list-style-type: none"> <li>• http(s) urls</li> <li>• git://</li> <li>• Maven-like urls: <a href="http://maven.apache.org/scm/git.html">http://maven.apache.org/scm/git.html</a></li> </ul> <p>For SSH authentication, create a VCS Root from a URL first and then open its settings to specify the SSH key to be used. Alternatively, create a Git VCS Root manually.</p>
Mercurial	<ul style="list-style-type: none"> <li>• http(s) urls</li> <li>• Maven-like urls: <a href="http://maven.apache.org/scm/mercurial.html">http://maven.apache.org/scm/mercurial.html</a></li> </ul>
Subversion	<ul style="list-style-type: none"> <li>• http(s) urls +Maven-like urls: <a href="http://maven.apache.org/scm/subversion.html">http://maven.apache.org/scm/subversion.html</a></li> <li>• svn://</li> </ul>

TFS	<p>Recommended URL formats:</p> <ul style="list-style-type: none"> <li>• <code>http[s]://&lt;tfsserver&gt;:&lt;port&gt;/&lt;collection name&gt;\$/&lt;project path&gt;</code>  <code>http[s]://&lt;tfsserver&gt;:&lt;port&gt;/tfs/&lt;collection name&gt;/&lt;project name&gt;</code>          for example: <a href="http://tfshost:8080/tfs/DefaultCollection\$/Project/root">http://tfshost:8080/tfs/DefaultCollection\$/Project/root</a></li> <li>• for Visual Studio Team Services:  <code>https://&lt;url to visualstudio.com&gt;/&lt;project name&gt;</code> or  <code>https://&lt;url to visualstudio.com&gt;/\$/&lt;project path&gt;</code>          for example: <a href="https://username.visualstudio.com/Project">https://username.visualstudio.com/Project</a></li> </ul> <p>See the <a href="#">related blog post</a> as well.</p>
Perforce	<ul style="list-style-type: none"> <li>• Maven-like urls: <a href="http://maven.apache.org/scm/perforce.html">http://maven.apache.org/scm/perforce.html</a></li> <li>• same as Maven but without the 'scm' prefix, for example: <code>perforce://main:1666/myproject/</code></li> </ul>
Vault	<p>http(s) urls from Vault SourceCode Control web which contain "repid" parameter, e.g.</p> <ul style="list-style-type: none"> <li>• <a href="http://vault-server.example.net/VaultService/VaultWeb/Default.aspx?repid=1709&amp;path=\$">http://vault-server.example.net/VaultService/VaultWeb/Default.aspx?repid=1709&amp;path=\$</a></li> </ul>
CVS	no support yet
ClearCase	no support yet

## ClearCase

### Initial Setup

An installed ClearCase client on the TeamCity server is required to make TeamCity ClearCase integration work. You also need to create a ClearCase view on the TeamCity server machine (regardless of whether you plan to use the server-side or agent-side checkout). This view will be used for collecting changes in ClearCase VCS roots and for checkout in case of the server-side checkout mode. In case of the agent-side checkout mode, the config spec of this view will be also used to automatically create views on agents.

 If you plan to use the agent-side checkout mode, make sure the ClearCase client (cleartool) is also installed on the agents and is properly configured, i.e. the tool must be in system paths and have access to the ClearCase Registry.

### ClearCase Settings

Common VCS Root properties are described here. The section below contains description of the fields and options specific to the ClearCase Version Control System.

Option	Description
ClearCase view path	<p>A local path on the TeamCity server machine to the ClearCase view created during the <a href="#">initial setup</a>. The snapshot view is preferred as there is no benefit in using the dynamic view with TeamCity. Also, dynamic views are not supported for the agent-side checkout (<a href="#">TW-21545</a>).</p> <div style="border: 1px solid #ffcc00; padding: 5px; margin-top: 10px;">  Do not use the view you are currently working with. TeamCity calls the update procedure before checking for changes and building patch, and thus it might lead to problems with checking in changes.       </div>
Relative path within the view	<p>the path relative to the "ClearCase view path" that limits the sources to watch and checkout for the build.</p>
Branches	<p>Branches are used in the "-branch" parameter for the "Ishistory" command to significantly improve its performance.</p> <p>You can either let TeamCity detect the needed branches automatically (via the view's config spec) or specify your own list of needed branches. Press the Detect now button to see the branches automatically detected by TeamCity.</p> <p>You can also choose the "custom" option and leave the text field blank - then the "-branch" parameter will not be used for the "Ishistory" command at all.</p> <p>If you specify or TeamCity detects several branches, then "Ishistory" will be called for every branch and all results will be merged.</p> <div style="border: 1px solid #0072bc; padding: 2px; border-radius: 15px; display: inline-block;">  "Ishistory" options can be customized, see <a href="#">TW-12390</a> </div>

Use ClearCase	Use the drop-down menu to select either UCM or BASE.
Global labeling	Check this option if you want to use global labels
Global labels VOB	Pathname of the VOB tag (whether or not the VOB is mounted) or of any file system object within the VOB (if the VOB is mounted)

 Make sure that the user that runs the TeamCity server process is also a ClearCase view owner.

## See also:

[Administrator's Guide: VCS Checkout Mode](#)

## CVS

Common VCS Root properties are described [here](#).

This page contains descriptions of CVS-specific fields and options available when setting up a VCS root. Depending on the selected access method, the page shows different fields that help you to easily define the [CVS settings](#):

- [CVS Root](#)
- [Checkout Options](#)
- [PServer Protocol Settings](#)
- [Ext Protocol Settings](#)
- [SSH Protocol Settings \(internal implementation\)](#)
- [Local CVS Settings](#)
- [Proxy Settings](#)
- [Advanced Options](#)

## CVS Root

Option	Description
Module name	Specify the name of the module managed by CVS.
CVS Root	<p>Use these fields to select the access method, point to the user name, CVS server host and repository. For example: ':pserver:user@host.name.org:/repository/path'. For a <a href="#">local</a> connection only the path to the CVS repository should be used. TeamCity supports the following connection methods (described below):</p> <ul style="list-style-type: none"> <li>• <a href="#">pserver</a></li> <li>• <a href="#">ext</a></li> <li>• <a href="#">ssh</a></li> <li>• <a href="#">local</a></li> </ul>

## Checkout Options

Option	Description
<ul style="list-style-type: none"> <li>• <a href="#">Checkout HEAD revision</a></li> <li>• <a href="#">Checkout from branch</a></li> <li>• <a href="#">Checkout by Tag</a></li> </ul>	Define the way CVS will fill in and update the working directory
Quiet period	Since there are no atomic commits in CVS, using this setting you can instruct TeamCity not to take (detect) a change until the specified period of time passed since the previous change was detected. It helps avoid situations when one commit is shown as two different changes in the TeamCity web UI.

## PServer Protocol Settings

Option	Description
CVS Password	Click this radio button if you want to access the CVS repository by entering a password.
Password File Path	Click this radio button to specify the path to the <code>.cvspass</code> file.
Connection Timeout	Specify the connection timeout.

#### Ext Protocol Settings

Option	Description
Path to external rsh	Specify the path to the rsh program used to connect to the repository.
Path to private key file	Specify the path to the file that contains user authentication information.
Additional parameters	Enter any required rsh parameters that will be passed to the program. Multiple parameters are separated by spaces.

#### SSH Protocol Settings (internal implementation)

Option	Description
SSH version	Select a supported version of SSH.
SSH port	Specify SSH port number.
SSH Password	Click this radio button if you want to authenticate via an SSH password.
Private Key	Click this radio button to use private key authentication. In this case specify the path to the private key file.
SSH Proxy Settings	See <a href="#">proxy options below..</a>

#### Local CVS Settings

Option	Description
Path to the CVS Client	Specify the path to the local CVS client.
Server Command	Type the server command. The server command is the command which makes the CVS client to work in server mode. Default value is 'server'

#### Proxy Settings

Option	Description
Use proxy	Check this option if you want to use a proxy, and select the desired protocol from the drop-down list.
Proxy Host	Specify the name of the proxy.
Proxy Port	Specify the port number.
Login	Specify the login.
Password	Specify the password.

#### Advanced Options

Option	Description

Use GZIP compression	Check this option to apply gzip compression to the data passed between the CVS server and the CVS client.
Send all environment variables to the CVS server	Check this option to send environment variables to the server for compatibility with certain servers.
History Command Supported	Check this option to use the history file on the server to search for newly committed changes. Enable this option to improve the CVS support performance. By default, the option is not checked because not every CVS server supports keeping a history log file.

## Git

TeamCity supports Git out of the box. Git source control with Visual Studio Team Services is supported (see authentication notes [below](#)).

This page contains description of the Git-specific fields of the VCS root settings.  
For common VCS Root properties, see [this section](#).



Git command line client needs to be installed on the agents if [the agent-side checkout is used](#).



### Important Notes

- [Remote Run](#) and [Pre-Tested Commit](#) are supported in the [IntelliJ IDEA](#) and [Eclipse](#) plugins; with the [Visual Studio Addin](#) use the [Branch Remote Run Trigger](#).
- Initial Git [checkout](#) may take significant time (sometimes hours), depending on the size of your project history, because the whole project history is downloaded during the initial checkout.

On this page:

- General Settings
  - Branch Matching Rules
  - Supported Git Protocols
- Authentication Settings
- Authenticating to Visual Studio Team Services
  - Personal Access Tokens
    - Required Access Scope
    - Alternate Authentication Credentials
- Server Settings
- Agent Settings
  - Git executable on the agent
- Configuring Git Garbage Collection on Server
- Git LFS
- Internal Properties
- Limitations
- Known Issues
- Development Links

## General Settings

Option	Description
Fetch URL	The URL of the remote Git repository used for fetching data from the repository.
Push URL	The URL of the target remote Git repository used for pushing annotated tags created via <a href="#">VCS labeling</a> build feature to the remote repository. If blank, the fetch URL is used.
Default branch	Configures <a href="#">default branch</a> . Parameter references are supported here. Default value is <code>refs/heads/master</code> <div style="border: 1px solid #ffcc00; padding: 5px; margin-top: 10px;">  You can configure Git-plugin to fetch all heads by adding a build configuration parameter <code>teamcity.git.fetchAllHeads=true</code> </div>

Branch specification	Lists the patterns for branch names, required for <a href="#">feature branches</a> support. The matched branches are monitored for changes in addition to the default branch. The syntax is similar to checkout rules: <code>+ -:branch_name</code> , where <code>branch_name</code> is specific to the VCS, i.e. <code>refs/heads/</code> in Git (with the optional <code>*</code> placeholder).
Use tags as branches	Allows monitoring / checking out git <a href="#">tags</a> as branches making branch specification match tag names as well as branches (e.g. <code>+ -:refs/tags/&lt;tag_name&gt;</code> ). By default, tags are ignored.
Username style	Defines a way TeamCity reports username for a VCS change. Changing the username style will affect only newly collected changes. Old changes will continue to be stored with the style that was active at the time of collecting changes.
Submodules	Select whether you want to ignore the submodules, or treat them as a part of the source tree. Submodule repositories should either not require authentication or use the same protocol and accept the same authentication as configured in the VCS root.
Username for tags/merge	A custom username used for <a href="#">labeling</a> .

## Branch Matching Rules

- If the branch matches a line without patterns, the line is used.
- If the branch matches several lines with patterns, the best matching line is used.
- If there are several lines with equal matching, the one below takes precedence.

Everything that is matched by the wildcard will be shown as a branch name in the TeamCity interface. For example, `+*:refs/heads/*` will match `refs/heads/feature1` branch, but in the TeamCity interface you'll see only `feature1` as a branch name.

The short name of the branch is determined as follows:

- if the line contains no brackets, then full line is used, if there are no patterns or part of line starting with the first pattern-matched character to the last pattern-matched character.
- if the line contains brackets, then part of the line within brackets is used.

When branches are specified here, and if your build configuration has a VCS trigger and a change is found in some branch, TeamCity will trigger a build in this branch.

## Supported Git Protocols

The following protocols are supported for Git repository URL:

- ssh: (e.g. `ssh://git.somewhere.org/repos/test.git`, `ssh://git@git.somwhereElse.org/repos/test.git`, scp-like syntax: `git@git.somwhere.org:repos/test.git`)



### Be Careful

The scp-like syntax requires a colon after the hostname, while the usual ssh url does not. This is a common source of errors.

- git: (e.g. `git://git.kernel.org/pub/scm/git/git.git`)
- http: (e.g. <http://git.somewhere.org/projects/test.git>)
- file: (e.g. `file:///c:/projects/myproject/.git`)



### Be Careful

When you run TeamCity as a Windows service, it cannot access mapped network drives and repositories located on them.

## Authentication Settings

Authentication Method	Description
Anonymous	Select this option to clone a repository with anonymous read access.
Password	Specify a valid username (if there is no username in the clone URL; the username specified here overrides the username from the URL) and a password to be used to clone the repository. For the <a href="#">agent-side checkout</a> , it is supported only if git 1.7.3+ client is installed on the agent. See <a href="#">TW-18711</a> . For Git hosted from Team Foundation Server 2013, specify NTLM credentials here.

<b>Private Key</b>	<p>Valid only for SSH protocol. A private key must be in the OpenSSH format. Select one of the options from the Private Key list and specify a valid username (if there is no username in the clone URL; the username specified here overrides the username from the URL).</p> <p>Available Private Key options:</p> <ul style="list-style-type: none"> <li>• Uploaded Key: uses the key(s) uploaded to the project. See <a href="#">SSH Keys Management</a> for details.</li> <li>• Default Private key - Uses the keys available on the file system in the default locations used by common ssh tools: the mapping specified in <code>&lt;USER_HOME&gt;/ .ssh/config</code> if the file exists or the private key file <code>&lt;USER_HOME&gt;/ .ssh/id_rsa</code> (the files are required to be present on the server and also on the agent if the <a href="#">agent-side checkout</a> is used).</li> <li>• Custom Private Key - Supported only for <a href="#">server-side checkout</a>, see <a href="#">TW-18449</a>. When this method is used, specify an absolute path to the private key in the Private Key Path field. If required, specify the passphrase to access your SSH key in the corresponding field.</li> </ul>
--------------------	---

For all the available options to connect to GitHub, please see the [comment](#).

#### Authenticating to Visual Studio Team Services

If you use Git source control with Visual Studio Team Services, the following options are available to you:

#### Personal Access Tokens

To use access tokens, you need to create a [personal access token](#) in your Visual Studio Team Services account, where you have to set some Code [access scope](#) in your repositories and use it when configuring a VCS root.

Option	Description
Username	Leave blank for TFVC, any value for Git, e.g. username
Password	Enter your personal access token created earlier

#### Required Access Scope

TFS subsystem	Scopes
TFVC	All scopes
Git	Code (read) / Code (read and write) for versioned settings
Work Items	Work items (read)
Commit Status	Code (status)

#### Alternate Authentication Credentials

To use the login/password pair authentication, you have to enable [alternate credentials](#) in your Visual Studio Team Services account, where you can set a secondary username and password to use when configuring a VCS root.

#### Server Settings

These are the settings used in case of the server-side [checkout](#).

Option	Description
Convert line-endings to CRLF	Convert line-endings of all text files to CRLF (works as setting <code>core.autocrlf=true</code> in a repository config). When not selected, no line-endings conversion is performed (works as setting <code>core.autocrlf=false</code> ). Affects the server-side checkout only. A change to this property causes a clean checkout.
Custom clone directory on server	To interact with the remote git repository, the its bare clone is created on the TeamCity server machine. By default, the cloned repository is placed under <code>&lt;TeamCity Data Directory&gt;/system/caches/git</code> and <code>&lt;TeamCity Data Directory&gt;/system/caches/git/map</code> . The field specifies the mapping between repository url and its directory on the TeamCity server. Leave this field blank to use the default location.

#### Agent Settings

These are the settings used in case of the agent-side [checkout](#).

Note that the agent-side checkout has limited support for SSH. The only supported authentication methods are "Default Private Key" and "Uploaded Private Key".

If you plan to use the [agent-side checkout](#), you need to have Git 1.6.4+ installed on the agents.

Option	Description
Path to git	Provide the path to a git executable to be used on the agent. When set to %env.TEAMCITY_GIT_PATH%, the automatically detected git will be used, see <a href="#">Git executable on the agent</a> for details
Clean Policy/Clean Files Policy	Specify here when the "git clean" command is to run on the agent, and which files are to be removed.
Use mirrors	When enabled (default), TeamCity clones the repository under the agent's system\git directory and uses the mirror as an alternate repository when updating the checkout directory for the build. As a result, this speeds-up clean checkout (because only the working directory is cleaned), and saves disk space (as there is only one clone of the given git repository on an agent).

 To configure a connection from a TeamCity server running behind a proxy to a remote Git repository, see [this section](#).

## Git executable on the agent

TeamCity needs Git command line client version 1.6.4+ on the agent in order to use the agent-side checkout.

The recommended approach is to ensure that the git client is available in PATH of the TeamCity agent and leave the "Path to git" setting in the VCS root blank.

If you only have the git command line on some machines, set "Path to git" setting in the VCS root to the %env.TEAMCITY\_GIT\_PATH% value.

Instead of adding Git to the agent's PATH, you can set the TEAMCITY\_GIT\_PATH environment variable (or env.TEAMCITY\_GIT\_PATH property in the agent's `buildAgent.properties` file) to the full path to the git executable.

If TEAMCITY\_GIT\_PATH is not defined, the Git agent plugin tries to detect the installed git on the launch of the agent. It first tries to run git from the following locations:

- for Windows - it tries to run `git.exe` at:
  - C:\Program Files\Git\bin
  - C:\Program Files (x86)\Git\bin
  - C:\cygwin\bin
- for \*nix - it tries to run `git` at:
  - /usr/local/bin
  - /usr/bin
  - /opt/local/bin
  - /opt/bin

If git is not found in any of these locations, it tries to run the git accessible via the PATH environment variable.

If a compatible git (1.6.4+) is found, it is reported in the TEAMCITY\_GIT\_PATH environment variable. This variable can be used in the Path to git field in the [VCS root](#) settings. As a result, the configuration with such a VCS root will run only on the agents where git was detected or specified in the agent properties.

## Configuring Git Garbage Collection on Server

TeamCity server maintains a clone for every Git repository it works with, so the process which collects changes in the large Git repository may cause memory problems on the TeamCity server if the Git garbage collection for the repository was not run for a long time.

TeamCity can automatically run git gc periodically when native Git client can be found on the server.

When TeamCity runs Git garbage collection, the details are logged into the `teamcity-cleanup.log`. If git garbage collection fails, a corresponding warning is displayed. To fix the warning / meet automatic git gc requirements, perform the following:

1. Install a native Git client manually on the TeamCity server.
2. Specify the directory to the Git executable:
  - a. either add it to the Path environment variable and restart the server,
  - b. or set it in the `teamcity.server.git.executable.path` internal property without the server restart.

TeamCity executes Git garbage collection until the total time doesn't exceed 60 minutes quota; the quota can be changed using the `teamcity.server.git.gc.quota.minutes` internal property.

Git garbage collection is executed every night at 2 a.m., this can be changed by specifying the internal property with a cron expression like this: "teamcity.git.cleanupCron=0 0 2 \* \* ?" (restart the server for the property to take effect)

## Git LFS

TeamCity supports Git LFS for agent-side checkout. To use it, install git 1.8.5+ and Git LFS on the build agent machine. Git LFS should be enabled using the 'git lfs install' command (on Windows an elevated command prompt may be needed). More information is available in [Git LFS documentation](#).

## Internal Properties

For Git VCS it is possible to configure the following [internal properties](#):

Property	Default	Description
teamcity.git.idle.timeout.seconds	1800	The idle timeout for communication with the remote repository. If no data were sent or received during this timeout, the plugin throws a timeout error to prevent hanging of the process forever.
teamcity.git.fetch.timeout	1800	(deprecated) Override of "teamcity.git.idle.timeout.seconds" for git fetch operation
teamcity.git.fetch.separate.process	true	Defines whether TeamCity runs git fetch in a separate process
teamcity.git.fetch.process.max.memory	512M	The value of the JVM -Xmx parameter for a separate fetch process. You also need to ensure the server machine has enough memory as the memory configured will be used in addition to the main server process and there can be several child processes doing git fetch and each using the configured amount of the memory.   For large repositories requiring heap memory greater than -Xmx1024m for Git fetch, <a href="#">switching to 64-bit Java</a> may be needed.
teamcity.git.monitoring.expiration.timeout.hours	24	
teamcity.server.git.gc.enabled	false	Whether TeamCity should run <code>git gc</code> during the server cleanup (native git is used)
teamcity.server.git.executable.path	git	The path to the native git executable on the server
teamcity.server.git.gc.quota.minutes	60	Maximum amount of time to run <code>git gc</code>
teamcity.git.cleanupCron	0 0 2 * * ?	<a href="#">Cron expression</a> for the time of a cleanup in git-plugin, by default - daily at 2a.m.
teamcity.git.stream.file.threshold.mb	128	Threshold in megabytes after which JGit uses streams to inflate objects. Increase it if you have large binary files in the repository and see symptoms described in <a href="#">TW-14947</a>
teamcity.git.buildPatchInSeparateProcess	true	Git-plugin builds patches in a separate process, set it to false to build patch in the server process. To build patch git-plugin has to read repository files into memory. To not run out of memory git-plugin reads only objects of size smaller than the threshold, for larger objects streams are used and they can be slow ( <a href="#">TW-14947</a> ). With patch building in a separate process all objects are read into memory. Patch process uses the memory settings of the separate fetch process.
teamcity.git.mirror.expiration.timeout.days	7	The number of days after which an unused clone of the repository will be removed from the server machine. The repository is considered unused if there were no TeamCity operations on this repository, like checking for changes or getting the current version. These operations are quite frequent, so 7 days is a reasonably high value.

teamcity.git.commit.debug.info	false	Defines whether to log additional debug info on each found commit
teamcity.git.sshProxyType		Type of ssh proxy, supported values: http, socks4, socks5. Please keep in mind that socks4 proxy cannot resolve remote host names, so if you get an UnknownHostException, either switch to socks5 or add an entry for your git server into the hosts file on the TeamCity server machine.
teamcity.git.sshProxyHost		Ssh proxy host
teamcity.git.sshProxyPort		Ssh proxy port
teamcity.git.connectionRetryAttempts	3	Number of attempts to establish connection to the remote host for testing connection and getting a current repository state before admitting a failure
teamcity.git.connectionRetryIntervalSeconds	4	Interval in seconds between connection attempts

#### Agent configuration for Git:

Property	Default	Description
teamcity.git.use.native.ssh	false	When checkout on agent: whether TeamCity should use native SSH implementation.
teamcity.git.idle.timeout.seconds	1800	The idle timeout for the git fetch operation when the agent-side checkout is used. The fetch is terminated if there is no output from the fetch process during this time. Prior to 8.0.4 the default was 600.

#### Limitations

When using checkout on an agent, a limited subset of [checkout rules](#) is supported. Git-plugin translates some of the checkout rules to the sparse checkout patterns. Only the rules which do not remap files are supported:

```
+:some/dir
-:some/dir/subDir
```

An unsupported rule example is `+:some/dir=>some/otherDir`.

#### Known Issues

- `java.lang.OutOfMemoryError` while fetch repository. Usually occurs when there are large files in the repository. By default, TeamCity runs fetch in a separate process. To increase memory available to this process, change the `teamcity.git.fetch.process.max.memory` internal property (see description of this property [above](#)).
- Teamcity run as a Windows service cannot access a network mapped drives, so you cannot work with git repositories located on such drives. To make this work, run TeamCity using `teamcity-server.bat`.
- inflation using streams in JGit prevents `OutOfMemoryError`, but can be time-consuming (see the related thread at [git-dev](#) for details and the [TW-14947](#) issue related to the problem). If you meet conditions similar to those described in the issue, try to increase `teamcity.git.stream.file.threshold.mb`. Additionally, it is recommended to increase the overall amount of memory dedicated for TeamCity to prevent `OutOfMemoryError`.

#### Development Links

Git support is implemented as an open-source plugin. For development links, refer to the [plugin's page](#).

#### See also:

[Administrator's Guide: Branch Remote Run Trigger](#)

#### SSH Keys Management

You can upload an SSH private key into a project via the TeamCity web interface and then use it in VCS roots configuration or in [SSH Agent](#) build feature. Supported Key Format

TeamCity supports keys in the OpenSSH format only. If your private key uses a different format, it has to be converted to the OpenSSH.

 For example, the Putty private key format (\*.ppk) not supported by TeamCity can be converted to the OpenSSH format using PuTTY Key Generator: use the menu `Conversions -> Export OpenSSH key`.

### Uploading SSH Key to TeamCity Server

1. Go to the Administration | <ProjectName> page.
2. On the left of the page, in Project Settings, click SSH Keys.
3. On the page that opens, click Upload SSH Key.
4. In the dialog that opens, select a private key usually stored in `<USER_HOME>/ .ssh/id_rsa` or `<USER_HOME>/ .ssh/id_dsa`.

When you upload an SSH key for the project, it is stored in `<TeamCity Data Directory>/config/projects/<project>/plugins/ssh_keys`. TeamCity tracks this folder and is able to pick up new keys on the fly. The key will be available in the current project and its subprojects.

 The access to the [TeamCity Data Directory](#) must be kept secure, as the keys are stored in an unmodified/unencrypted form on the file system.

Once the key is uploaded, a VCS root can be configured to use this uploaded key.

### SSH Key Usage

See [SSH Agent](#) for usage from within the build scripts.

The uploaded key can be used in a VCS root. SSH key is used on the server and is also passed to the agent in case agent-side checkout is configured.

During the build with agent-side checkout, the Git plugin downloads the key from the server to the agent. It temporarily saves the key on the agent's file system and removes it after `git fetch/clone` is completed.

 The key is removed for security reasons: e.g. the tests executed by the build can leave some malicious code that will access the build agent file system and acquire the key. However, tests cannot get the key directly since it is removed by the time they are running. It makes it harder but not impossible to steal the key. Therefore, the agent must also be secure.

To transfer the key from the server to the agent, TeamCity encrypts it with a DES symmetric cipher. For a more secure way, configure an [https connection between agents and the server](#).

### Mercurial

TeamCity uses the typical Mercurial command line client: `hg` command. Mercurial 1.5.2+ is supported.



 Mercurial is to be installed on the server machine and, if the [agent-side checkout](#) is used, on the agents.

Note that:

- Remote Run from IDE is not supported. Please use [Branch Remote Run Trigger](#) instead.
- Checkout rules for agent-side checkout are not supported except for the `.=><target_dir>` rule.

For common VCS Root properties, see [this section](#). The section below contains the description of Mercurial-specific fields and options.

TeamCity supports Mercurial out of the box.

On this page:

- [General Settings](#)
  - [Path to hg executable detection](#)
- [Agent Settings](#)
- [Internal Properties](#)

General Settings

Option	Description
Pull changes from	The URL of your hosting.
Default branch	Set to the default branch which used in the absence of branch specification or when the branch of the branch specification cannot be found. Note that parameter references are supported here.
Branch specification	In this area list all the branches you want to be monitored for changes. The syntax is similar to checkout rules: <code>+   - :branch_name</code> , where <code>branch_name</code> is specific to the VCS (with the optional <code>*</code> placeholder). Note that only one asterisk is allowed and each rule has to start with a new line. Bookmarks can also be used in the branch and branch specification fields. If a bookmark has the same name as a regular branch, a regular branch wins. More in the related <a href="#">TeamCity blogpost</a> .
	<p> Bookmarks support requires Mercurial 2.4 installed on the TeamCity server.</p>
Use tags as branches	Allows you to use tags in branch specification. By default, tags are ignored.
Detect subrepo changes	By default, subrepositories are not monitored for changes.
Username for tags/merge	A custom username used for labeling
Use uncompressed transfer	Uncompressed transfer is faster for repositories in the LAN.
HG command path	The path to the hg executable. Used on TeamCity server only if included into whitelist. See more <a href="#">below</a> .

Path to hg executable detection

When an agent starts, the hg-plugin detects Mercurial installed on the agent machine.

The plugin tries to run the `hg version` command using the path specified by `teamcity.hg.agent.path` parameter. You can change this parameter in `<Agent Home Directory>\conf\buildAgent.properties`.

If this parameter is not set, the plugin uses `hg` as a path to the command, assuming it is somewhere in the `$PATH`. If the command is executed successfully and mercurial has an appropriate version (1.5.2+), then the hg-plugin reports the path to hg in the `teamcity.hg.agent.path` parameter.

During the build, the plugin uses the hg specified in the HG command path field of a VCS root settings. To use the detected hg, put %teamcity.hg.agent.path% in this field. Configurations with such settings will be run only on agents which report the path to hg.

The server side of the plugin checks the value of the teamcity.hg.customServerHgPathWhitelist [internal property](#). The property contains the ;-separated list of allowed hg paths to use on the server. If the path specified in VCS root is in whitelist, then it is used on the server. If not, the path specified in the teamcity.hg.server.path [internal property](#) is used. If this property is not set, TeamCity server uses hg from the \$PATH.

## Agent Settings

These are the settings used in case of the agent-side checkout ([default mode](#)), which requires Mercurial installed on all agents.

Option	Description
Mercurial config	Specify the Mercurial configuration options to be applied to the repository during agent-side checkout, e.g. enter the following to enable the largefiles extension: <pre>[extensions] largefiles =</pre> The configuration format is described <a href="#">here</a> . Before 2017.2.2 this option was also used on TeamCity server. This was disabled for security reasons.
Purge settings	Defines whether to <a href="#">purge files</a> and directories not being tracked by Mercurial in the current repository. You can choose to remove only unknown files and empty directories, or to remove ignored files as well. Added files and (unmodified or modified) tracked files are preserved.
Use mirrors	When enabled, TeamCity creates a local agent mirror first (under agent's system/mercurial directory) and then clones to the working directory from this local mirror. This option speeds up clean checkout, because only the build working directory is cleaned. Also, if a single root is used in several build configurations, a clone will be faster.

## Internal Properties

This section describes hg-related [internal properties](#). You can modify the defaults to adjust the Mercurial settings as needed.

Server-side [internal properties](#):

Property	Default	Description
teamcity.hg.pull.timeout.seconds	3600	Maximum time in seconds for pull operation to run
teamcity.hg.server.path	hg	Path to the hg executable on the server (see <a href="#">Path to hg executable detection</a> for the details).
teamcity.hg.customServerHgPathWhitelist		;-separated list of allowed paths to hg executable to use on TeamCity server machine

Agent configuration for Mercurial:

Property	Default	Description
teamcity.hg.pull.timeout.seconds	3600	Maximum time in seconds for pull operation to run
teamcity.hg.agent.path	hg	Path to hg executable on the agent (see <a href="#">Path to hg executable detection</a> for the details).

See also:

[Administrator's Guide: Branch Remote Run Trigger](#)

Perforce

This page contains descriptions of the fields and options available when setting up VCS roots using Perforce. Common VCS Root properties are [described here](#).

 A Perforce client must be installed on the TeamCity server and it should be present in PATH. Alternatively, a full path

to p4 could be set via internal property `teamcity.perforce.customP4Path`.

If you plan to use the agent-side **checkout mode**, note that a Perforce client must be installed on the agents, and the path to the p4 executable must be added to the PATH environment variable.

Also check [TeamCity and Perforce compatibility](#).

- [P4 Connection Settings](#)
- [Case-Insensitivity in Checkout Rules](#)
- [Checkout On Agent Settings](#)
  - [Perforce Workspace Parameters](#)
  - [Perforce Proxy Settings](#)
- [Other Settings](#)
- [Perforce Jobs Support](#)
- [Logging](#)
- [Perforce Workspace Handling in TeamCity](#)
- [Perforce VCS Compatibility](#)
- [Perforce Streams as feature branches](#)

#### P4 Connection Settings

Option	Description
Port	<p>Specify the Perforce server address. The format is <code>host:port</code>.</p> <p><b>!</b> For specific environments, P4Host can be specified in the <a href="#">Workspace options below</a> for any type of checkout.</p>
Stream	<p>Click this radio button to specify an existing Perforce stream. TeamCity will use this stream to prepare the stream-based workspace, and will use the client mapping from such a workspace.</p> <p>TeamCity supports deeper directory structure within the root depot: depots with a depth of <code>//DEPOTNAME/1/2/n</code> can be specified in this field.</p> <p>Prior to TeamCity 2017.2, TeamCity supports streams stored one level below the depot name: the format is <code>//streamdepot/streamname</code>.</p> <p>Parameters are supported. For the <code>StreamAtChange</code> option, use the <a href="#">Label to checkout</a> field.</p> <div style="border: 1px solid red; padding: 5px; margin-top: 10px;"><p><b>!</b> <b>Performance impact</b> When this option is used with the <a href="#">Checkout on the server</a> mode, the internal TeamCity source caching on the server side is disabled, which may worsen the performance of <a href="#">clean checkpoints</a>. Also, with this option, snapshot dependencies builds are not reused. (<a href="#">TW-41898</a> - fixed in TeamCity 2017.1)</p></div> <div style="border: 1px solid orange; padding: 5px; margin-top: 10px;"><p><b>!</b> <b>Checkout rules limitations</b> When Perforce Streams are used with the <a href="#">Checkout on the agent</a>, simple checkout rules like <code>. =&gt; sub/directory</code> are supported. Exclude checkout rules, multiple include rules, or rules like <code>aaa=&gt;bbb</code> are not supported.</p></div> <div style="border: 1px solid orange; padding: 5px; margin-top: 10px;"><p>Enable feature branches support (experimental) - select this check box to specify branch streams you want to be monitored for changes in addition to the default one. Enter / Edit the branch specification as a newline-delimited set of rules. The syntax is <code>+ -:stream_name</code> (with the optional <code>*</code> placeholder).</p></div> <div style="border: 1px solid orange; padding: 5px; margin-top: 10px;"><p><b>!</b> <b>Task stream limitations</b> When task streams are used for feature branches, TeamCity may miss some changes in task streams until a modifying commit is made, which means that merge commits from the parent stream are not detected until a 'real' commit to the task stream is made (<a href="#">TW-44765</a>).</p></div>

Client	Click this radio button to directly specify the client workspace name. The workspace must be already created by a Perforce client application like P4V or P4Win. Only the mapping rules from the configured client workspace are used. The client name is ignored.
	<p> <b>Performance impact</b> When this option is used with the <a href="#">checkout on the server</a> mode, the internal TeamCity source caching on the server side is disabled, which may worsen the performance of <a href="#">clean checkouts</a>. Also, with this option, snapshot dependencies builds are not <a href="#">reused</a>. (<a href="#">TW-41898</a> - fixed in TeamCity 2017.1)</p>
Client Mapping	Click this radio button to specify the mapping of the depot to the client computer. If you have Client mapping selected, TeamCity handles file separators according to the OS/platform of the build agent where a build is run. To enforce specific line separator for all build agents, use Client or Stream with the <code>LineEnd</code> option specified in Perforce instead of Client mapping. Alternatively, you can add an <a href="#">agent requirement</a> to run builds only on a specific platform.
	<p> <b>Tip</b> Use <code>team-city-agent</code> instead of the client name in the mapping.</p>
	<p>Example:</p> <pre>//depot/MPS/... //team-city-agent/... //depot/MPS/lib/tools/... //team-city-agent/tools/...</pre>
	<p> Prior to TeamCity 10.0 editing the client mapping for a Perforce VCS root resulted in a <a href="#">Clean Checkout</a> before the next build. A <a href="#">workaround</a> was provided.</p> <p>Now <a href="#">Clean Checkout</a> on a client mapping change is not enforced for the agent-side checkout in the following cases:</p> <ul style="list-style-type: none"> <li>• when a Perforce client name is used. Changing the Perforce client mapping for the client will not result in a clean checkout</li> <li>• when a Perforce stream is used, changing the stream name while keeping the same stream root will not result in a clean checkout</li> </ul> <p> If the direct client mapping is changed, a clean checkout will be forced unless the <code>teamcity.perforce.enable-no-clean-checkout</code> <a href="#">internal property</a> is set on the server.</p>
Username	Specify the user login name.
Password or Ticket	Specify the password or ticket.
Ticket-based authentication	Check this option to enable ticket-based authentication.  This option is enabled by default and not displayed.

#### Case-Insensitivity in Checkout Rules



Note that Perforce support in TeamCity treats checkout rules as case-sensitive. Case-insensitivity for Perforce-based build configurations can be enabled on the VCS settings page by adding the following comment in the Edit Checkout Rules field:

```
##teamcity ignore-case
```

#### Checkout On Agent Settings

When the [agent-side checkout](#) is used, TeamCity creates a Perforce workspace for each [checkout directory/VCS root](#). These workspaces are automatically created when necessary and are automatically deleted after some idle time. It is possible to customize the name generated by TeamCity: add the `teamcity.perforce.workspace.prefix` configuration parameter at the [Parameters](#) page with the prefix in the value.

Option	Description
Workspace options	If needed, you can set here the following options for the <code>p4 client</code> command: <code>Options</code> , <code>SubmitOptions</code> , and <code>LineEnd</code> .  ! For specific environments, P4Host can be specified here for any type of checkout.
Run 'p4 clean' for cleanup	Enable this option to clean up your workspace from extra files before a build (since p4 2014.1) When enabled, the <code>p4 clean</code> command will be run before <code>p4 sync</code> command, unless <code>p4 sync -f</code> or <code>p4 sync -p</code> is used. See the <a href="#">p4 sync command reference</a> .
Skip the have list update:	Enable this option not to track files on the Perforce server on sync (always transfer all files to the agent, <code>p4 sync -p</code> )
Extra sync options	Specify additional 'p4 sync' options, like <code>--parallel</code> . See <a href="#">p4 sync command reference</a> .

#### Perforce Workspace Parameters

With checkout on agent, TeamCity provides environment variables describing the Perforce workspace created during the checkout process.

If several Perforce VCS Roots are used for the checkout, the variables are created for the first VCS root.  
The variables are:

- `P4USER` - same as `vcsroot.<VCS root ID>.user` parameter
- `P4PORT` - same as `vcsroot.<VCS root ID>.port` parameter
- `P4CLIENT` - name of the generated P4 workspace on the agent

These variables can be used to perform custom p4 commands after the checkout.

More information: [Perforce Workspace Handling in TeamCity](#)

#### Perforce Proxy Settings

To allow using Perforce proxy with the [agent-side checkout](#), specify the `env.TEAMCITY_P4PORT` environment variable [on the build agent](#) and the agent will take this value as the `P4PORT` value.

#### Other Settings

P4 path on the build agent	<p>Specify the path to the Perforce command-line client: <code>p4.exe</code> file.</p> <p>This field works only on the agent side for agent-side checkout (prior to TeamCity 2017.2.2 this path was be used for both the server-side checkout and the agent-side checkout). For the server, the <code>p4</code> binary should be present in the <code>PATH</code> of the TeamCity server or can be specified via the <code>teamcity.perforce.customP4Path</code> <a href="#">internal property</a>).</p> <p>To restore old behavior, the <code>teamcity.perforce.p4PathOnServerWhitelist</code> <a href="#">internal property</a> can be used to specify a semi-colon-separated list of allowed p4 paths.</p>
Label/changelist to sync	<p>If you need to check out sources not with the latest revision, but with a specific Perforce label (with selective changes), you can specify this label here. For instance, this can be useful to produce a milestone/release build, or a reproduce build. If the field is left blank, the latest changelist will be used for sync.</p> <div style="border: 2px solid red; padding: 5px; margin-top: 10px;"> <span style="color: red;">!</span> It is recommended to use the <a href="#">agent-side checkout</a> if you use symbolic labels. With the server-side checkout on label, TeamCity will perform full checkout.       </div>
Charset	Select the character set used on the client computer.
Support UTF-16 encoding	<p>Enable this option if you have <code>UTF-16</code> files stored as <code>utf16</code> <a href="#">Perforce file type</a> in your project.</p> <p>You may want to enable this option if you use server-side checkout and have files of the <code>utf16</code> <a href="#">Perforce file type</a> in your depot. Enable this flag for the checked out files to be in the <code>UTF-16</code> encoding. Otherwise, such files may be converted to <code>UTF-8</code> upon checkout.</p> <p>If you store <code>UTF-16</code> files as the binary <a href="#">Perforce file type</a>, they will always be checked out "as is", no conversion will be performed.</p>

## Perforce Jobs Support

For a changelist which was checked in with one or several associated jobs, TeamCity shows a wrench icon  which allows you to view details of the jobs when clicked or hovered over.

## Logging

All Perforce plugin operations are logged into teamcity-vcs.log files with category jetbrains.buildServer.VCS.P4 (on an agent or on a server, depending on the operation context). The detailed logging can be enabled with [debug-vcs preset](#).

## Perforce Workspace Handling in TeamCity

Please refer to a [separate page](#).

## Perforce VCS Compatibility

Please refer to a [separate page](#).

## Perforce Streams as feature branches

Please refer to a [separate page](#).

See also:

[Administrator's Guide: VCS Checkout Mode](#)

## Perforce VCS Compatibility

Perforce server version	TeamCity version	Comment
2009.1 .. 2016.1	10.0+	
Feature branches work with 2014.2+		
2009.1 .. 2015.1	9.1.1+	

2010.2 .. 2015.1	9.1.0	TW-41876
2009.1 .. 2015.1	8.1.x..9.0.x	
2009.1 .. 2013.2	8.0.5	
2009.1 .. 2014.1 (might work with later versions, not tested)	8.0.6+	TW-34128
2009.1 .. 2012.1	7.1.2	TW-24046
2009.1 .. 2013.2	7.1.3..8.0.5	

## Perforce Workspace Handling in TeamCity

### Overview

To perform Perforce-related operations, TeamCity usually operates in a "no-workspace" mode, i.e. it executes Perforce commands without workspace context. For instance, checking for changes operations and creating server-side patches do not require Perforce workspaces creation.

The cases when a workspace is created are:

- [Agent-side checkout](#), the default mode. In this case TeamCity creates a Perforce workspace to checkout the sources
- Using [versioned settings](#) with Perforce VCS.

### Perforce Workspace Name

The names of the created workspaces start with the TC\_p4\_ prefix. It is possible to provide an additional prefix for the workspace name using the `teamcity.perforce.workspace.prefix` configuration parameter.

The name of the workspace also includes the build agent name and a hash value built from the checkout directory and (optionally) checkout rules.

### Perforce Workspace Parameters

With [agent-side checkout](#), TeamCity provides environment variables describing the Perforce workspace created during the checkout process.

If several Perforce VCS Roots are used for the checkout, the variables are created for the first VCS root in the list of the build's VCS Roots.

The variables are:

- `P4USER` - same as `vcsroot.<VCS root ID>.user` parameter
- `P4PORT` - same as `vcsroot.<VCS root ID>.port` parameter
- `P4CLIENT` - the name of the generated P4 workspace on the agent

These variables can be used to perform custom p4 commands after the checkout.

### Workspace Deletion

TeamCity deletes the Perforce workspaces it created in different situations:

- Immediately after a versioned settings commit (a workspace is created for each commit)
- For the agent-side checkout - when a [clean checkout](#) is performed (TeamCity will also run `p4 sync -f` in this case, see details [below](#))
- In the background of the agent process (between builds), when it detects a non-existing workspace directory for a workspace associated with the current agent. A TeamCity agent performs a cleanup of unused [checkout directories](#) (the default timeout is 8 days, can be changed with the `system.teamcity.build.checkoutDir.expireHours` system property). When a checkout directory is deleted, and this directory is associated with a Perforce workspace, this workspace is deleted as well. Cleaning Perforce workspaces can be disabled via the `teamcity.perforce.workspace.cleanup=false` setting, either in the `buildAgent.properties` file or globally at the server level as a Root project [configuration](#) parameter.

When deleting a Perforce workspace which contains pending changes or opened files, TeamCity tries to revert the changes and remove pending changelists, and after that repeats the operation. If the second attempt fails as well, TeamCity tries to run the `p4 client -d -f` operation (forced). All those actions are logged to `teamcity-vcs.log`.

Also, TeamCity does not force workspace deletion when a Perforce edge/replica server is used.

### Perforce sync -f and workspace reuse

When [agent-side checkout](#) is used, the TeamCity Perforce plugin creates a workspace which is bound to the checkout directory on an agent. The checkout is performed with an incremental `p4 sync` command (both for personal and non-personal builds).

When a VCS Root is configured to use `p4 sync -p`, the Perforce plugin always runs this command to checkout the sources.

Usually, each [clean checkout](#) build results in `p4 sync -f` command with cleaning the sources. But for the Perforce agent checkout there are exceptions described below.

### Errors during checkout

When an error occurs during the checkout, or a build is interrupted/stopped during the checkout, or a timeout occurs, no [clean checkout](#) will occur for the subsequent builds on the same build agent. Instead, TeamCity will rely on the Perforce ability to recover from the state.

#### VCS Root client mapping modification

Usually, when a project administrator modifies a VCS Root client mapping specified in the VCS Root, this is considered as a change in VCS Root settings and results in a [clean checkout](#). This clean checkout behaviour can be disabled using the `teamcity.perforce.enable-no-clean-checkout=true` internal property.



Changing `teamcity.perforce.enable-no-clean-checkout` internal property results in one-time clean checkout for all affected build configurations.

When a VCS Root is configured to use Client Name, or Stream, no clean checkout will occur when the client mapping of the corresponding client/stream is edited in Perforce. The corresponding TeamCity issue is [TW-25344](#).

#### Forced protection against clean checkout

There is a [configuration parameter](#) which protects a build configuration from the TeamCity-initiated clean checkout, `teamcity.agent.failBuildOnCleanCheckout`. When this parameter is set to `true`, TeamCity will fail a build instead of running a clean checkout. It will never clean the workspace, unless it is explicitly requested via the [Enforce clean checkout](#) action or if the "Clean all files in the checkout directory before the build" option is enabled in the checkout options of the version control settings for a build configuration.

When using a custom checkout path, TeamCity will not clean the checkout directory when VCS settings are changed, and it will fail a build instead. To ignore clean checkout and proceed with incremental checkout, use the `teamcity.agent.failBuildOnCleanCheckout=ignoreAndContinue` parameter for a project or build configuration. Do this only if you're absolutely sure that the sources in the checkout directory are in the correct state.

The same applies to the broken personal builds. When the sources become dirty and the option is set, TeamCity will fail the build instead of running a clean checkout. You can clean the working copy via '`p4 clean`', for instance, and try to continue with the `ignoreAndContinue` value after this (you can run a custom build with the specified [configuration parameter](#)).

The corresponding TeamCity issue is [TW-33168](#).

## Perforce Streams as feature branches

### How to enable

On the VCS Root page, select checkbox "Enable feature branches support" after the parent stream name.

After that, all streams which have the specified main stream as a parent, will be included into the feature branches.

It is possible to specify some mapping to include only specific streams into the feature branches set, like

```
+://stream-depot/*
```

In this case, only streams under depot stream-depot will be included for changes collection/build triggering.

### Task streams

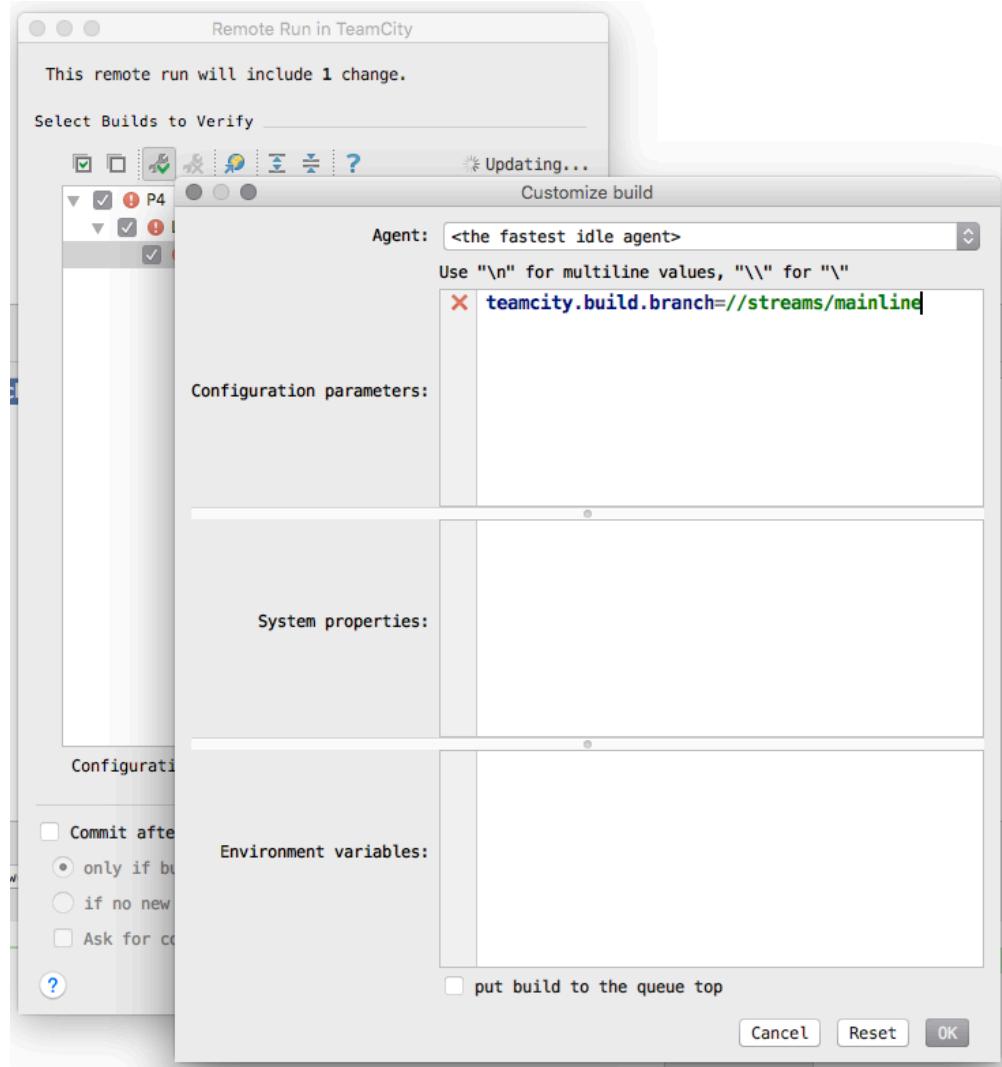
Task streams are supported, but new task streams are not detected by TeamCity until there is a non-merge commit into this stream.

### Remote run

Remote run from IDEA is possible only in a stream which was already detected by TeamCity. TeamCity remote run plugin tries to deduce the correct stream according to the depot paths of the files in the IDE working copy.

For instance, if a file path in the working copy starts with `//depot/stream1/some/path`, TeamCity will try finding `//depot/stream1` stream and start remote run there.

But if you modified a file from another stream (imported into the working copy) and want to enforce build in a particular stream, you should specify a configuration parameter `teamcity.build.branch` when triggering the remote run.



## StarTeam

This page describes the fields and options available when setting up VCS roots using StarTeam. Common VCS Root properties are described [here](#).

- [StarTeam Connection Settings](#)
- [Notes on Directory Naming Convention](#)

### StarTeam Connection Settings

Option	Description
URL	Specify the StarTeam URL that points to the required sources
Username	Enter the login for the StarTeam Server
Password	Enter the corresponding password for the user specified in the field above

EOL Conversion	Define the EOL (End Of Line) symbol conversion rules for text files. Select one of the following: <ul style="list-style-type: none"> <li>As stored in repository — EOL symbols are preserved as they are stored in the repository. No conversion is done.</li> <li>Agent's platform default — Whatever EOL symbol is used in a particular file in the repository, it will be converted to the platform-specific line separator when passed to the build agent. The resulting EOL symbol exclusively depends on the agent's platform.</li> </ul>
Directory naming	Define the mode for naming the local directories. Select one of the following: <ul style="list-style-type: none"> <li>Using working folders — StarTeam folders have the "working folder" property. It defines which local path corresponds to the StarTeam folder (by default, the working folder equals the folder's name). In this mode TeamCity gives the local directories the names stored in the "working folder" properties. Please note that even though StarTeam allows using absolute paths as working folders, TeamCity supports relative paths only and doesn't detect the presence of absolute paths.</li> <li>Using StarTeam folder names — In this mode the local directories are named after corresponding StarTeam folders' names. This mode can suit users who keep the working directory structure the same as the project structure in the repository and don't want to rely on "working folder" properties because they can be uncontrollably modified.</li> </ul>
Checkout mode	Files can be retrieved by their current (tip) revision, by label, or by promotion state. Note that if you select checkout by label or promotion state, change detection won't be possible for this VCS root. As a result, your builds cannot be triggered if the label is moved or the promotion state is switched to another label. The only way of keeping the build up-to-date is using the schedule-based triggering. To make it possible, a full checkout is always performed when you select checkout by label or promotion state options.

#### Notes on Directory Naming Convention

When checking out sources TeamCity (just as StarTeam native client) forms local directory structure using working folder name s instead of just a folder name. By default, the working folder name for a particular StarTeam folder equals the folder's name. For example, your project has a folder named "A" with a subfolder named "B". By default, their working folders are "A" and "B" correspondingly, and the local directory structure will look like <checkout dir>/A/B. But if the working folder for folder "A" is set to something different (for example, "Foo"), the directory structure will also be different: <checkout dir>/Foo/B.

StarTeam allows specifying absolute paths as working folders. However, TeamCity supports only relative working folders. This is done by design; all files retrieved from the source control must reside under the checkout directory. The build will fail if TeamCity detects the presence of absolute working folders.

You need to ensure that all the folders under the VCS root have relative working folder names.

#### Subversion

This page contains descriptions of Subversion-specific fields and options available when setting up a VCS root. Common VCS Root properties are described [here](#).

- SVN Connection Settings
- SSH settings
- Checkout on agent settings
- Labeling settings
- Authentication for SVN externals
- Timeouts
  - Connection timeout
  - Read timeout
    - Subversion server access via HTTP/HTTPS (both server/agent)
    - Subversion server access via svn:// or svn+ssh://
- Miscellaneous

You do not need Subversion client to be installed on the TeamCity server or agents. TeamCity bundles the Java implementation of SVN client ([SVNKit](#)).

#### SVN Connection Settings

Option	Description
URL	Specify the SVN URL that points to the project sources.
Username	Specify the SVN user name.
Password	Specify the SVN password.

Configuration Directory	You can specify an alternative subversion configuration directory, or use the default one (recommended). This setting also applies to agent-side checkout. TeamCity does not store authentication in SVN configuration directory, but can read settings stored there.
Use default configuration directory	Check this option to make this the default configuration directory for the SVN connection.
Externals Support	<p>Check one of the following options to control the SVN externals processing.</p> <ul style="list-style-type: none"> <li>Full support (load changes and checkout) - when the option is selected, TeamCity will check out all configuration's sources (including the sources from the externals) and will gather and display information about externals' changes on the <a href="#">Changes tab</a>.</li> <li>Checkout, but ignore changes - when the option is selected, TeamCity will check out the sources from externals but any changes in externals' source files will not be gathered and displayed in the <a href="#">Changes tab</a>. You can use this option if you have several SVN externals and do not want to get information about any changes made in the externals' source files.</li> </ul> <div style="border: 1px solid #f0e68c; padding: 10px; margin-top: 10px;"> <p> <b>Build revision number impact</b> If you use the "Checkout, but ignore changes" option, TeamCity will always use the latest repository revision as the revision for checkout (the same revision will be used for the <code>build.vcs.number</code> parameter). For other two options, TeamCity takes the revision of the latest detected change as the revision for checkout.</p> <p> <b>Subversion Repository UUID</b> TeamCity relies on Subversion repository UUID as an unique identifier of a repository. If you have 2 different repositories with the same UUID (due to repository copy) TeamCity may function incorrectly, for instance, wrong HEAD revision of an external repository can be checked out.</p> </div>
HTTPS Connections: Accept non-trusted SSL certificates  (Enable non-trusted SSL certificate in 10.0)	When this option is enabled, TeamCity is able to connect to SVN servers without properly signed SSL certificate.

 Note that if you have anonymous access for some path within SVN, the entered username will never be used to authenticate when accessing any of its subfolders. Anonymous access will be used instead. This rule only applies for `svn://` and `http(s)://` protocols; i.e. if you have a build configuration which uses a combination of this VCS Root + [VCS Checkout Rules](#) referencing a non-restricted path above the restricted one for another build configuration, changes under the restricted path will be ignored even if you specify correct username/password for the VCS Root itself.

## SSH settings

Option	Description
Private Key File Path	Specify the full path to the file that contains the OpenSSH-formatted private key. You can also specify an <a href="#">SSH key uploaded to TeamCity</a>
Private Key File Password	Enter the password to the SSH private key.
SSH Port	Specify the port that SSH is using.

 Only the OpenSSH format is supported for the key. The key in a format unsupported by TeamCity has to be converted to the OpenSSH format (e.g. a Putty private key (\*.ppk) can be converted using PuTTYgen.exe: see [Conversions -> Export](#)

OpenSSH key).

#### Checkout on agent settings

Option	Description
Working copy format	Select the format of the working copy. Available values for this option are 1.4 through 1.8 (current default) This option defines the format version of Subversion files located in <code>.svn</code> directories, when the <a href="#">checkout on agent mode</a> is used. The specified format is important in two cases: <ul style="list-style-type: none"><li>If you run command-line <code>svn</code> commands on the files checked out by TeamCity. For example, if your working copy has version 1.5, you will not be able to use Subversion 1.4 binaries to work with it.</li><li>If you use new Subversion features; for example, file-based externals which were added in Subversion 1.6. Thus, unless you set the working copy format to 1.6, the file-based externals will not be available in the check out on agent mode.</li></ul>
Revert before update	If the option is selected, then TeamCity always runs the " <code>svn revert</code> " command before updating sources; that is, it will revert all changes in versioned files located in the checkout directory. When the option is disabled and local modifications are detected during the update process, TeamCity runs the " <code>svn revert</code> " after the update. TeamCity does not delete non-versioned files in the working directory during the revert. For deleting non-versioned files, consider using <a href="#">Swabra</a>

#### Labeling settings

Option	Description
Labeling rules	Specify a newline-delimited set of rules that defines the structure of the repository. See the <a href="#">detailed format description</a> for more details.

#### Authentication for SVN externals

TeamCity does not allow specifying SVN externals authentication parameters explicitly, in user interface. To authenticate on the SVN externals server, the following approaches are used:

- authenticate using the same credentials (username/password) as for the main repository
- authenticate without explicit username/password. In this case, the credentials should be already available to the `svn` process (usually, they stored in subversion configuration directory). So, this require setting correct "Configuration Directory" or "Default Config Directory" option under [SVN Connection Settings](#)

When TeamCity has to connect to a SVN external, it uses the following sequence:

- if the SVN external URL has the same prefix as the main repository (there is a match > 20 characters), TeamCity tries the main repository credentials first, and in case of a failure tries to connect without the username/password (so they picked up from SVN configuration directory)
- if the SVN external URL noticeably differs from the main repository, TeamCity tries to connect without the username/password, and in case of a failure, tries using the credentials from the main repository

#### Timeouts

Sometimes, the SVN checkout operation for remote SVN servers may fail with a error like `svn: E175002: timed out waiting for server`.

Usually this can happen due to network slowness or the SVN server overload.

The timeout values for the connection and for read operations can be configured.

#### Connection timeout

Connection timeout is applied when TeamCity creates a connection to the SVN server. The default timeout for this operation is 60 seconds, and can be specified via the TeamCity internal property `teamcity.svn.connect.timeout`, in seconds. The value of the property is set differently for server-side checkout and agent-side checkout:

- Server-side operations - [configure internal property](#)
- Agent-side checkout - [add start-up property](#)

#### Read timeout

The read timeout is used when a connection with the SVN server is established, and TeamCity is waiting for the data from the server. The value of the timeout depends on the SVN server access protocol.

Subversion server access via HTTP/HTTPS (both server/agent)

For HTTP read timeout TeamCity uses the `http-timeout` setting specified in the `servers` file in the Subversion configuration directory. On Win32 systems, this directory is typically located the Application Data area of the user's profile directory. On Unix/Mac, this directory is usually named `$HOME/.subversion` for the user account who runs the TeamCity server/agent.

If not specified, the default value for the timeout is 1 hour.

Subversion server access via `svn://` or `svn+ssh://`

In this case the read timeout can be specified in seconds via the TeamCity internal property `teamcity.svn.read.timeout`. The default value is 30 minutes. The value of the property is set differently for server-side checkout and agent-side checkout:

- Server-side operations - configure internal property
- Agent-side checkout - add start-up property

#### Miscellaneous

Directories are not considered changed when they have the "svn:mergeinfo" Subversion property changes only. See [details](#).

#### See also:

[Administrator's Guide: Configuring VCS Settings | VCS Checkout Mode](#)

#### Team Foundation Server

This page contains descriptions of the fields and options available when setting up a VCS root to connect to Microsoft Team Foundation Server Version Control.

Common VCS Root properties are described here.

When connecting to a Visual Studio Team Services Git repository, select **Git** as Type of VCS.



TeamCity can also automatically configure the project / VCS root from a repository URL.

If you have a TFVC root configured, TeamCity will suggest configuring the Team Foundation Work Items as well.

#### On this page:

- [Cross-Platform TFS Integration](#)
- [TFS Settings](#)
- [Agent-Side Checkout](#)
- [Authentication Notes](#)
  - [Personal Access Tokens](#)
    - [Required Access Scope](#)
    - [Alternate Authentication Credentials](#)
    - [NTLM/Kerberos on Linux and macOS](#)
- [Team Foundation Proxy Configuration](#)
- [HTTP Proxy Server Configuration](#)
  - [Default .NET Working Mode](#)
  - [Cross-Platform Working Mode](#)

#### Cross-Platform TFS Integration

TeamCity features the [cross-platform TFS integration](#), which works on Linux, macOS, and Windows platforms. Without installing additional software, TeamCity servers and build agents can interact with Team Foundation Servers 2010 - 2018 and Visual Studio Team Services.

The built-in TFS plugin can work in two modes: the default and cross-platform. The working mode is based on the availability of Team Explorer (default mode): if it is not present, the plugin falls back from the default to cross-platform mode.

When detecting the Team Explorer version, TeamCity checks `.NET GAC` and the following paths:

- Windows x86: `%CommonProgramFiles%\Microsoft Shared\Team Foundation Server\%version_number%`
- Windows x64: `%CommonProgramFiles(x86)%\Microsoft Shared\Team Foundation Server\%version_number%`

To enforce the cross-platform mode on TeamCity, set the `teamcity.tfs.mode=java` [internal property](#) or [build configuration parameter](#).

#### TFS Settings

Option	Description
--------	-------------

URL	Team Foundation Server URL in the following format:  TFS 2010+: http[s]://<TFS Server>:<Port>/tfs/<Project Collection Name> TFS 2005/2008: http[s]://<TFS Server>:<Port> TFS 2005/2008: http[s]://<TFS Server>:<Port> Visual Studio Team Services: https://<accountname>.visualstudio.com
Root	Specify the root using the following format: \$<project name><project catalogue>
Username	Specify a user to access Team Foundation Server. This can be a user name or DOMAIN\UserName string. Use blank to let TFS select a user account that is used to run the TeamCity Server (or Agent for the agent-side checkout).
Password	Enter the password of the user entered above

Learn more about authentication in [Visual Studio Team Services](#).

#### Agent-Side Checkout

The [agent-side](#) checkout is supported on Windows, as well as Linux and Mac agent machines.

TeamCity automatically creates a TFS workspace for each [checkout directory](#) used. The workspace is created on behalf of the user account specified in the VCS root settings.

By default, the created TFS workspace uses the location defined in the TFS server settings. You can force TeamCity to use a specific workspace location via the [build configuration parameter](#) teamcity.tfs.workspace.location set to local or server.

The created TFS workspaces are automatically removed based on the timeout configured via the teamcity.tfs.workspace.idleTime build agent property, set to the default value of [1209600](#) sec (2 weeks).

Option	Description
Enforce overwrite all files	When the option is enabled, TeamCity will call TFS to update workspace rewriting all files.

 Normally, there is no need to do a forced update for every build. But, if you suspect that TeamCity is not getting the latest version from the repository, you can use this option.

 TFS does not allow several workspaces on a machine mapped to the same directory. If it happens, the TeamCity TFS agent-side checkout will attempt to remove intersecting workspaces to create a new workspace that matches the specified VCS root and checkout rules.

Note that it can fail to remove workspaces created by another user, and in this case you need to remove such workspaces manually.

It is recommended to use checkout rules of the format below to differentiate local mappings:

```
$/root1 => /root1
$/root2 => /root2
```

#### Authentication Notes

The following authentication options are available in Visual Studio Team Services.

#### Personal Access Tokens

To use access tokens, you need to create a [personal access token](#) in your Visual Studio Team Services account, where you have to set some Code [access scope](#) in your repositories and use it when configuring a VCS root.

Option	Description
Username	Leave blank for TFVC, any value for Git, e.g. username
Password	Enter your personal access token created earlier

## Required Access Scope

TFS subsystem	Scopes
TFVC	All scopes
Git	Code (read) / Code (read and write) for versioned settings
Work Items	Work items (read)
Commit Status	Code (status)

### Alternate Authentication Credentials

To use the login/password pair authentication, you have to enable [alternate credentials](#) in your Visual Studio Team Services account, where you can set a secondary username and password to use when configuring a VCS root.

## NTLM/Kerberos on Linux and macOS

To use this authentication method, check that your machine includes Kerberos libraries and that the authentication is properly configured. If you encounter any issues, please check the steps described in the [Microsoft documentation](#).

### Team Foundation Proxy Configuration

To enable usage of [Team Foundation Proxy](#), define the `TFSPROXY` environment variable for the user account which runs the TeamCity server or agent and restart them to apply changes.

#### Example

```
TFSPROXY=https://tfss-proxy:8081
```

### HTTP Proxy Server Configuration

#### Default .NET Working Mode

To interact with the TFS server, [the proxy server settings](#) specified for the user account which runs the TeamCity server or agent will be used.

#### Cross-Platform Working Mode

The default Java proxy server settings specified for the TeamCity server or agent will be used in the TFS integration. On the TeamCity server, [internal properties or Java options](#) can be used. On the TeamCity agent, [build agent configuration or Java options](#) can be used. The TeamCity-TFS integration supports the following options:

```
http.proxyHost  
http.proxyPort  
http.nonProxyHosts  
https.proxyHost  
https.proxyPort  
http.proxyUser  
http.proxyPassword
```

### SourceGear Vault

SourceGear Vault Version Control System support is implemented as a plugin. Please, refer to the [plugin's page](#) for the configuration details.

### Visual SourceSafe



Since TeamCity 2018.1 The Visual SourceSafe plugin is no longer bundled with TeamCity and is available as a [separate download](#).

## Notes and Limitations

- TeamCity supports Microsoft Visual SourceSafe 6.0 and 2005 (English versions only!).
- Microsoft Visual SourceSafe only works if the TeamCity server is installed on a computer running a Windows® operating system.
- Make sure the TeamCity server process is run by a user that has permission to access the VSS databases.

TeamCity has the following limitations with Visual SourceSafe:

- Shared (not branched) files cannot be checked out.
- Comments for add and delete operations for file and directories are not shown in VSS 6.0. All such operations will have "No Comment" instead of a real VSS comment. (This limitation is imposed by the VSS API, which makes it impossible to retrieve comments with acceptable performance).
- The timestamps on VSS check in are driven by local time on client computer. Therefore if the time on clients and build server are not synchronized, TeamCity cannot properly order check-ins. There is also problem with timezone, however it was already addressed in VSS client version 2005, when configured properly. To avoid these problems follow the recommendations:
  1. Sync client machines via timeserver: <http://support.microsoft.com/kb/131715/EN-US>.
  2. Setup timezone for VSS database: Start VSS admin-> Tools-> Options-> TimeZone and pick one.
  3. Use VSS 2005.

## VSS Settings

### Configuring VCS Post-Commit Hooks for TeamCity

- Overview
- Post-commit generic script
- Setting up post-receive hook on Git server
- Setting up hook on Mercurial server
- Setting up post-commit hook on Subversion server
- Setting up post-commit trigger on Perforce server
- Setting up service hook on Team Foundation Server for TFVC and Git
  - TFVC Repository
  - Git Repository
- Troubleshooting

## Overview

By default TeamCity uses a polling approach to detect changes in a VCS repository, i.e. for each VCS Root, it periodically sends requests to the version control repository server to find out whether there are new revisions. For large installations with hundreds of VCS roots, this may create a noticeable load on the VCS server and on TeamCity itself.

To avoid background polling, it is possible to set up a post-commit hook on the VCS server, which will notify TeamCity to start checking for changes procedure. This way TeamCity will make background requests for changes detection only when such changes are available.

Even with commit hooks configured and working properly TeamCity still makes requests for changes on the server start and on each build queuing (or starting) to ensure the latest changes are used even if commit hooks stopped to function.

When a commit hook call comes in, TeamCity automatically increases the **VCS polling interval** (the minimum after the increase is 15 minutes, maximum is 4 hours, increased by 2 times on each successful check). If the commit hook stops working (e.g. TeamCity finds a change in a VCS root which it did not receive a commit hook call for), the **VCS polling interval** value is reset to default.

Commit hooks are received via TeamCity REST API requests which should typically be configured to in the post-commit repository triggers:

```
POST .../app/rest/vcs-root-instances/commitHookNotification?locator=<vcsRootInstancesLocator>
```

The request returns textual details as to the performed operation or an error message.

It is important to find the "<vcsRootInstancesLocator>" for the request to match only the affected VCS roots from those configured in the TeamCity instance. If too many VCS roots are matched by the request configured in the commit hook, it will lead to more requests and more overload on the VCS repository and TeamCity than using default polling approach. Some examples of the "locator" are provided below.

The request should be performed by a user who has "View build configuration settings" permission for all the projects where

VCS root is defined.

**!** Note that by default only the first 100 matched "VCS root instances" will be matched by the request. To match more, "count:9999" can be added as below.

The most common form of the "<vcsRootInstancesLocator>" is "vcsRoot:(type:<TYPE>, count:99999),property:(name:<URL\_PROPERTY\_NAME>, value:<VCS\_REPOSITORY\_URL\_PART>, matchType:contains, ignoreCase:true),count:99999". However, for VCS roots without parameter %-references, a more performant variance can be used: "vcsRoot:(type:<TYPE>, property:(name:<URL\_PROPERTY\_NAME>, value:<VCS\_REPOSITORY\_URL\_PART>, matchType:contains, ignoreCase:true),count:99999),count:99999".

Commit hooks examples for UNIX-based VCS servers are described below.

Post-commit generic script

Save the script below on a VCS server as `teamcity-trigger.sh`:

```
SERVER=https://buildserver-url
USER=buildserver-user
PASS=<password>

LOCATOR=$1

# The following is one-line:
(sleep 10; curl --user $USER:$PASS -X POST
"$SERVER/app/rest/vcs-root-instances/commitHookNotification?locator=$LOCATOR" -o /dev/null)
>/dev/null 2>&1 <&1 &

exit 0
```

Set the variables according to your TeamCity server. The user must have View build configuration settings permission for projects where VCS root is defined. This permission is included in the Project developer role by default.

**!** If your TeamCity server uses a custom SSL certificate, you'll need to pass `-k` or `--cacert /path/to/correct/internal/CACertificate` parameter to the `curl` command above.

Setting up post-receive hook on Git server

1. Locate the Git repository root on the target VCS server. It should contain the `.git/hooks` directory with some templates.
2. Create the `.git/hooks/post-receive` file with a line:

```
/path/to/teamcity-trigger.sh
'vcsRoot:(type:jetbrains.git,count:99999),property:(name:url,value:<VCS root repository URL>,matchType:contains,ignoreCase:true),count:99999'
```

where `<VCS root repository URL>` must be replaced with the repository URL specified in the corresponding TeamCity VCS root and the value should be URL-escaped. Note that locator has `matchType:contains` in it, which means you can specify some part of URL too.

3. Make sure that both `teamcity-trigger.sh` and `hooks/post-receive` scripts can be read and executed by Git user(s). You may need to execute the following command:

```
chmod 755 /path/to/teamcity-trigger.sh /path/to/git_root/.git/hooks/post-receive
```

Setting up hook on Mercurial server

1. Locate the Mercurial repository root on the target VCS server.
2. Create or edit the `.hg/hgrc` config and add the following snippet:

```
[hooks]
changegroup = /path/to/teamcity-trigger.sh
'vcsRoot:(type:mercurial,count:99999),property:(name:repositoryPath,value:<VCS root
repository url>,matchType:contains,ignoreCase:true),count:99999'
```

where `<VCS root repository URL>` must be replaced with the repository URL specified in the corresponding TeamCity VCS root and the value should be URL-escaped. Note that the locator has `matchType:contains` in it, which means you can specify some part of the URL too.

3. Make sure that `teamcity-trigger.sh` is executable. You may need to execute the following command:

```
chmod 755 /path/to/teamcity-trigger.sh
```

#### Setting up post-commit hook on Subversion server

1. Locate the Subversion repository root containing the `db`, `hooks`, `locks`, and other directories. We need the `hooks` directory.
2. Create the `hooks/post-commit` file with a line:

```
/path/to/teamcity-trigger.sh 'vcsRoot:(type:svn,count:99999),property:(name:url,value:<VCS
root repository url>,matchType:contains,ignoreCase:true),count:99999'
```

where `<VCS root repository URL>` must be replaced with the repository URL specified in the corresponding TeamCity VCS root and the value should be URL-escaped. Note that the locator has `matchType:contains` in it, which means you can specify some part of URL too.

3. Make sure that both `teamcity-trigger.sh` and `hooks/post-commit` script can be read and executed by the process of the Subversion server. You may need to execute the following command:

```
chmod 755 /path/to/teamcity-trigger.sh /path/to/svn_repository_root/hooks/post-commit
```

#### Setting up post-commit trigger on Perforce server

Set up a `change-commit` trigger by adding one or several lines when [editing p4 triggers specification](#) (the text below must be placed in one line, one line per a VCS Root):

```
check-for-changes-teamcity change-commit //depot/project1/... "/path/teamcity-trigger.sh '<VCS
Root locator>'"
```

Where `<VCS Root locator>` can be one of the following:

- for Stream-based VCS roots: `vcsRoot:(type:perforce,count:99999),property:(name:stream,value://streamdepot/streamname,matchType:contains,ignoreCase:true),count:99999`
- for Client-based VCS roots: `vcsRoot:(type:perforce,count:99999),property:(name:client,value:<client
name>,matchType:contains,ignoreCase:true),count:99999`
- for Client-mapping VCS roots: `vcsRoot:(type:perforce,count:99999),property:(name:client-mapping,value:<so
me unique part of client mapping>,matchType:contains,ignoreCase:true),count:99999`

## Setting up service hook on Team Foundation Server for TFVC and Git

The latest TFS 2015 and Visual Studio Team Services provides service hooks for code commit events. To create a hook, perform the following steps:

1. Open the admin page for a team project in web access.
2. Create a subscription by running the wizard.
3. Select the "Web Hooks" service to integrate with.
4. Select the "Code checked in" event and specify a filter.
5. Fill in the TeamCity username, password, and server URL in the format:

```
"$SERVER/app/rest/vcs-root-instances/commitHookNotification?locator=$LOCATOR",
```

Where the \$LOCATOR value depends on the TFS repository type as described in the sections below.

### TFVC Repository

```
vcsRoot:(type:tfss, count:99999),property:(name:tfss-url,value:<TFS server url>,matchType:contains,ignoreCase:true),property:(name:tfss-root,value:<TFS project>,matchType:contains,ignoreCase:true),count:99999
```

where <TFS server url> must be replaced with the value specified in the TFS VCS root URL and path properties.  
Example:

```
http://teamcity/app/rest/vcs-root-instances/commitHookNotification?locator=vcsRoot:(type:tfss, count:99999),property:(name:tfss-url,value:http%3A%2F%2Ftfss%3Aport%2Ftfss%2Fcollection, matchType:contains,ignoreCase:true),property:(name:tfss-root,value:Project,matchType:contains,ignoreCase:true),count:99999
```

### Git Repository

```
vcsRoot:(type:jetbrains.git,count:99999),property:(name:url,value:<VCS root repository URL>,matchType:contains,ignoreCase:true),count:99999
```

where <VCS root repository URL> must be replaced with the repository URL specified in the corresponding TeamCity VCS root and the value should be URL-escaped. Example:

```
http://teamcity/app/rest/vcs-root-instances/commitHookNotification?locator=vcsRoot:(type:jetbrains.git,count:99999),property:(name:url,value:https%3A%2F%2Faccount.visualstudio.com%2FDefaultCollection%2FProject%2F_git%2FRepository,matchType:contains,ignoreCase:true),count:99999
```

6. Press Finish to create the service hook.

### Troubleshooting

It is recommended to try executing the following command from the command line before configuring the actual hook:

```
curl --user $USER:$PASS -X POST  
"$SERVER/app/rest/vcs-root-instances/commitHookNotification?locator=$LOCATOR"
```

If the commit hook matches the VCS root on the server correctly, you should see the output similar to this:

Scheduled checking for changes for 1 VCS roots. (Server time: 20160719T192540.787+0300)

If the commit hook has not found any VCS roots, it will report an error:

No VCS roots are found ...

Possible reasons for this output:

- the specified locator is incorrect, it does not match any VCS root on the server
- the specified user does not have enough permission for at least one of the matched VCS roots.

To check what roots are actually matched, use the request (see also [details](#)):

```
curl --user $USER:$PASS -X POST "$SERVER/app/rest/vcs-root-instances?locator=$LOCATOR"
```

 A commit hook supports matching more than one VCS root, but it is highly recommended to limit the matched VCS roots only to those affected by the change generating the event.

It is recommended to set up a commit hook per a VCS repository. In this case on a check-in to some repository, TeamCity will not spend resources trying to find commits in other non-related VCS roots which were also matched by the commit hook.

By default, the number of VCS roots matched by a commit hook in TeamCity is limited by 100. If you want to match more than 100 VCS roots, add the count parameter: <locator>,count:1000 Check [corresponding request description](#).

## Working with Feature Branches

Feature Branches in distributed version control systems (DVCS) allow you to work on a feature independently from the main development and commit all the changes for the feature onto the branch, merging the changes into the main branch when your feature is complete. This approach brings a number of advantages to software development teams; however, in continuous integration servers that do not have dedicated support for it, it also causes a number of problems, like constant build configurations duplication, poor visibility, and, in the end, loss of control over the process.

TeamCity support for feature branches is continuously increasing and, among other features, includes [Branch Remote Run Trigger](#) starting a new personal build each time TeamCity detects changes in a particular branches of the VCS roots of the build configuration and [Automatic Merge](#) to merge a branch into another after a successful build.

- Supported version control systems
- Configuring branches
- Default branch
- Logical branch name
- Builds
- Changes
- Active branches
- Tests
- Failure Conditions
- Triggers
- Dependencies
- Notifications
- Build configuration status
- Multiple VCS roots
- Build parameters
- Clean-up
- Manual branch merging

Supported version control systems

[Git](#) and [Mercurial](#) feature branches are supported as well as Perforce [branch streams support](#).

Configuring branches

To start working with DVCS branches, you need to configure which branches need to be watched for changes. This is done in the General Settings section of a [Git](#) or [Mercurial](#) VCS root via the Branch Specification field. With Perforce, check the corresponding box to enable feature branches support, which will display the branch specification field. The field accepts a list of branch names or patterns. TeamCity monitors the branches matched by the branch specification in addition to the [default branch](#).

Once you've configured the branch specification, TeamCity will start to monitor these branches for changes. If your build configuration has a [VCS trigger](#) and a [change is found in some branch](#), TeamCity will trigger a build in this branch. From the build configuration home page you'll also be able to filter the history, change log, pending changes and issue log by the branch name. Branch names will also appear in the custom build dialog, so you'll be able to manually trigger a custom build on a branch too.

The syntax of the branch specification field is newline-delimited list of "+|-:branch\_name" rules. "+" rules include the matching branches into the list, the "-" rules exclude the branches from the list.

Each rule can have one optional "\*" placeholder which matches one or more characters. e.g. "+:refs/heads/teamcity\*" matches all branches starting with `refs/heads/teamcity` and at least one additional character. The branch with `refs/heads/teamcity` will not be matched.

The "branch\_name" is VCS-specific, i.e. `refs/heads/master` in Git:



Branch specification: Edit branch specification:  
+:refs/heads/\*

Branches to monitor besides the default one as a newline-delimited set of rules in the form of +|-:branch name (with th

The part of the branch name matched by the asterisk (\*) wildcard becomes the short branch name to be displayed in the TeamCity user-level interface (also known as the [logical branch name](#)). The line can also contain optional parentheses which, when present, denote the part of the pattern to be used as the logical name instead of just \*-matched symbols.

You can use parameters in the branch specification.

When a single VCS branch is matched by several lines of the branch specification, the most specific (least characters matched by pattern) last rule applies.

That is:

If the specification contains an exact pattern matching the branch (i.e. a pattern without the \* wildcard), then the last such pattern is used. So if you have a specification like this:

```
+:refs/heads/release-v1  
-:refs/heads/release-v1
```

then the last pattern will win and the branch will be excluded.

If a branch specification has several patterns with the \* wildcard, then TeamCity selects the pattern producing the shortest logical name. The branch specification

```
+:refs/heads/*/hotfix  
-:refs/heads/v1/*
```

will include the `refs/heads/v1/hotfix` branch (because `v1` is shorter than `hotfix`). If 2 patterns with \* wildcard produce logical names of the same length, then the last pattern wins.

The branch specification supports comments as lines beginning with "#".

There is also a special escaping syntax defined via "#! escape: CHARACTER" syntax:

e.g. to use round brackets in a branch name, you need to escape them. Let's say you want to track the `release-(7.1)` branch: to do that, specify an escaping symbol as the first line in the specification.

For Mercurial, the following branch specification does that:

```
#! escape: \  
+:release-\(7.1\)
```

## Default branch

When configuring a VCS root for DVCS, you need to specify the branch name to be used as the default one in case a branch name was not specified. For example, if someone clicks on a Run button, TeamCity will create a build in the default branch.

The default branch is always implicitly included into the branch specification. In the TeamCity UI the default branch is marked with darker background of the branch marker.

It is possible to [disable building in the default branch](#) if you want to build pull requests only.

## Logical branch name

A logical branch name is a branch name shown in the user interface for the builds and on build configuration level. A logical branch name is regularly a part of the full VCS-specific branch name. It is calculated by applying a [branch specification](#) to the branch name from the version control.

For example, if the branch specification is defined like this:

```
+refs/heads/*
```

then the part matched by \* (e.g. `master`) is a logical branch name.

If the branch specification pattern uses parentheses, the logical name then is made up of the part of the name within the parentheses; to see the `v8.1/feature1` logical name displayed in the UI for the VCS branch `refs/heads/v8.1/feature1`, use this:

```
+refs/heads/(v8.1/*)
```

You do not need to include the default branch into the branch specification as it is already included there implicitly. But, if you want to have some short logical branch name for the default branch in the UI, e.g. `master`, you can include it in the branch specification and use the parentheses:

```
+refs/heads/(master)
```

## Builds

Builds from branches are easily recognizable in the TeamCity UI, because they are marked with a special label:

## PyCharm Educational Edition

<Active branches>

|▽

Branches

Overview

History

Change Log

Issue Log

Statistics

Compatible Agents

24

Pending Changes

507

There are 5 active branches in this configuration. 23 inactive branches are not displayed.



2018.2

#1457

Success |▽

Changes (16) |▽

13 hours ago (15m:2)



2018.1

Pending (71) |▽

#1423

Success |▽

Changes (16) |▽

one month ago (8m:1)



2018.3

Pending (25) |▽



Inspections-rework

Pending (100+) |▽



update\_tests

Pending (50) |▽

You can also filter history by a branch name if you're interested in a particular branch.

TeamCity assigns a branch label to the builds from the default branch too.

### Changes

For each build TeamCity shows changes included in it. For builds from branches the changes calculation process takes the branch into account and presents you with the changes relevant to the build branch. The change log with its graph of commits will help you understand what is going on in the monitored branches.

## ✓ #1457 (30 Jun 18 01:04) ▾

[Overview](#)[Changes 16](#)[Build Log](#)[Parameters](#)[Dependencies](#)[Issues](#)[« ✓ #1456](#)[All history](#)[Edit Config](#)[Artifacts](#)

VCS Root	Revision	Links
(jetbrains.git) Help Sources v2	refs/heads/2018.2 1e56eced6ba2732531068a9730a07b2a3d8d8e94	...

Show changes by: [<All users>](#)[Filter](#)[Advanced search](#)

Results per page:

 Show graph [?](#)  Show files


more feedback comments	Alla Redko	1 file   ▾	1e56eced6ba2	29 Jun 18 2
Fixed TXP00010.	Artem Pronichev	1 file   ▾	6e8b98df2367	29 Jun 18 1
Merge remote-tracking branch 'origin/2018.2' into 2018.2	Artem Pronichev	0 files   ▾	314eb8026974	29 Jun 18 1
Restored anchors.	Artem Pronichev	1 file   ▾	3ff176ab10da	29 Jun 18 1
Fixing TXP0001	Alla Redko	1 file   ▾	da2794648630	29 Jun 18 1
Fixed the lost anchor error	Artem Pronichev	1 file   ▾	79f740ca3e26	29 Jun 18 1
Merge remote-tracking branch 'origin/2018.2' into 2018.2	Artem Pronichev	0 files   ▾	7977c7dfda4e	29 Jun 18 1
Fixed <<<< error	Artem Pronichev	1 file   ▾	e8ab5cf78558	29 Jun 18 1
PY-29885   ▾ : review comments	Alla Redko	5 files   ▾	ef3abf211f7e	29 Jun 18 1
Updated the whole file about managing data sources and linked chunks.	Artem Pronichev	2 files   ▾	16bb6b890ded	29 Jun 18 1
WI-42648   ▾ : Customize 'Rework documentation on generating code'	Anton Monakov	18 files   ▾	0461689e3ec4	29 Jun 18 1
CPP-13474   ▾ : 'Configuring CLion on macOS' - more minor corrections	Marina Kalashina	1 file   ▾	69c1d4b95d97	29 Jun 18 1

With the Show graph option enabled by default TeamCity displays build markers on the graph.

## Active branches

In a build configuration with configured branches, the Overview page shows active branches.

A number of parameters define whether a branch is active. The parameters can be changed either in a build configuration (this will affect one build configuration only), project, or in the [internal properties](#) (this defines defaults for the entire server). A parameter in the configuration overrides a parameter in the [internal properties](#).

A branch is considered active if:

- it is present in the VCS repository and has recent commits (i.e. commits with the age less than the value of `teamcity.activeVcsBranch.age.days` parameter, 7 days by default).
- or it has recent builds (i.e. builds with the age less than the value of `teamcity.activeBuildBranch.age.hours`

parameter, 24 hours by default).

 A closed VCS branch with builds will still be displayed as active during 24 hours after last build. To remove closed branches from display, set `teamcity.activeBuildBranch.age.hours=0`.

## Tests

TeamCity tries to detect new failing tests in a build, and for those tests which are not new, you can see in which build the test started to fail. This functionality is aware of branches too, i.e. when the first build is calculated, TeamCity traverses builds from the same branch.

Additionally, a [branch filter](#) is available on the test details page and you can see a history of test passes or failures in a single branch.

## Failure Conditions

If build fail condition is configured as follows: build metric has changed comparing to a last successful/finished/pinned build, then the build from the same branch will be used. If there is no suitable build on the same branch, then build from default branch is used and the corresponding message is added to the build log.

## Triggers

The VCS trigger is fully aware of branches and will trigger a build once a check-in is detected in a branch. All VCS trigger options like per-checkin triggering, quiet period, and triggering rules are directly available for builds from branches. By default, the Schedule and Finish build trigger will watch for builds in the default branch only.

Additionally, a [branch filter](#) can be specified for the VCS, Schedule and Finish build triggers.

## Dependencies

If a build configuration with branches has snapshot dependencies on other build configurations with branches, then when a build in a branch is triggered, the other builds in the chain will also get the branch associated, if the branches in the VCS roots of the builds have the same [logical name](#) and this branch is not excluded by the branch specification. The VCS roots of the builds can point to different repositories, but the logical branch name must be the same.

If this condition is met, the branches with this name will be checked out and all the builds down the chain (which the build triggered depends on) and all the builds up the chain (depending on the triggered build) will be marked with the same branch. Otherwise the default branch will be checked out.

It is possible to configure artifact dependencies to retrieve artifacts from a build from a specific branch: artifact dependencies will use builds from the branch specified. The same applies to the [Schedule](#) and [Finish Build](#) triggers.

## Notifications

All notification rules except "My changes" will only notify you on builds from the default branch. At the same time, the "My changes" rule will work for builds from all available branches.

## Build configuration status

The Build Configuration status is calculated based on the builds from the default branch only. Consequently, per-configuration investigation works for builds from the default branch. For example, a successful build from a non-default branch will not remove a per-configuration investigation, but a successful build from the default branch will.

## Multiple VCS roots

If your build configuration uses more than one VCS root and you specified branches to monitor in both VCS roots, the way the builds are triggered is more complicated.

The VCS trigger groups branches from several VCS roots by [logical branch names](#). When some root does not have a branch from the other root, its default branch is used. For example, you have 2 VCS roots, both have the default branch `refs/heads/master`, the first root has the branch specification `refs/heads/7.1/*` and changes in branches `refs/heads/7.1/feature1` and `refs/heads/7.1/feature2`, the second root has the specification `refs/heads/devel/*` and changes in branch `refs/heads/dev1/feature1`. In this case VCS trigger runs 3 builds with revisions from following branches combinations:

root1	root2
<code>refs/heads/master</code>	<code>refs/heads/master</code>
<code>refs/heads/7.1/feature1</code>	<code>refs/heads/devel/feature1</code>
<code>refs/heads/7.1/feature2</code>	<code>refs/heads/master</code>

## Build parameters

If you need to get the branch name in the build script or use it in other build configuration settings as a parameter, please refer to [Predefined Build Parameters#Branch-Related Parameters](#).

## Clean-up

Clean-up rules are applied [independently](#) to each [active branch](#).

## Manual branch merging

You can merge branches in TeamCity manually, e.g. if you want to merge branches only after a code review / approval, or if you want to perform the merge despite the tests failure in a branch.

To merge sources manually:

Open the [build results page](#), click the Actions menu in the top-right corner and select "Merge this build sources...". The dialog that appears enables you to select the destination branch and add a commit message (required).

It is also possible to merge branches [automatically](#).

## See also:

[Administrator's Guide: Git | Mercurial](#)

## VCS Checkout Rules

VCS Checkout Rules allow you to check out a part of the configured VCS root and to map directories from the version control to subdirectories in the [build checkout directory](#) on a Build Agent. Thus, you can define a VCS root for the entire repository and make each build configuration checkout only the relevant part of it.

The Checkout Rules affect the changes displayed in the TeamCity for the build and the files checked out for the build on agent. To display changes, but not to trigger a build for a change, use [VCS Trigger Rules](#).

The general recommendation is to have a small number of VCS roots (pointing to the root of the repository) and define what is checked out by a specific build configuration via checkout rules.

To add a checkout rule, go to the build configuration's Version Control Settings page, locate the VCS root in the list, and click

the Edit checkout rules link. A pop-up window will appear where you can enter the rule. Use the VCS repository browser  to select a directory to check out.



Note that Perforce support in TeamCity treats checkout rules as case-sensitive. Case-insensitivity for Perforce-based build configurations can be enabled on the VCS settings page by adding the following comment in the Edit Checkout Rules field:

```
##teamcity ignore-case
```

## Syntax

In the examples below the paths in the repository (VCSPPath) are relative to the configured VCS Root, the paths on the agent (AgentPath) are relative to the build checkout directory:

The general syntax of a single checkout rule is as follows:

+|-: VCSPath [=> AgentPath]

 When entering rules, please note that as soon as you enter any "+:" rule, TeamCity will remove the default "include all" setting. To include all the files, use "+: ." rule.

 Please note that exclude checkout rules (in the form of "-:") will generally only speed up server-side checkouts, unless you use [Perforce](#) and [TFS](#) agent-side checkout, where exclude rules are processed in an effective manner. With other VCSs, agent-side checkouts may emulate the exclude checkout rules by checking out all the root directories mentioned as include rules and deleting the excluded directories. So with these VCS, exclude checkout rules should generally be avoided for the agent-side checkout. Please refer to the [VCS Checkout Mode](#) page for more information.

With Git agent-side checkout, TeamCity translates some of the checkout rules to the sparse checkout patterns. See the [details](#).

When entering rules please note the following:

- To enter multiple rules, each rule should be entered on a separate line.
- For each file the most specific rule will apply if the file is included, regardless of what order the rules are listed in.
- If you don't enter an operator, it will default to +:

Rules can be used to perform the following operations:

Syntax	Explanation
+:.=>AgentPath	Checks out the root into Path directory on an Agent
-:VCSPath	Excludes VCSPath (note: the path must be a directory and not a filename)
+:VCSPath=>.	Maps the VCSPath from the VCS to the <a href="#">Build Agent's default work directory</a>
VCSPath=>NewAgentPath	Maps the VCSPath from the VCS to the NewAgentPath directory on the Build Agent
+:VCSPath	Maps the VCSPath from the VCS to the same-named directory (VCSPath) on the Build Agent

An example with three VCS checkout rules:

```
-:src/help  
+:src=>production/sources  
+:src/samples=>./samples
```

In the above example, the first rule excludes the `src/help` directory and its contents from checkout. The third rule is more specific than the second rule and maps the `src/samples` path to the `samples` path in the Build Agent's default work directory. The second rule maps the contents of the `src` path to the `production/sources` on the build agent, except `src/help` which was excluded by the first rule and `src/samples` which was mapped to a different location by the third rule.

See also:

[Administrator's Guide: VCS Checkout Mode](#)

## VCS Checkout Mode

The Version Control Settings page for a build configuration allows configuring how project source code is retrieved from VCS: you can [attach a VCS Root here](#) and configure checkout options.

The VCS Checkout mode is a setting that affects how project sources reach an agent. This mode affects only sources checkout. The current revision and changes data retrieving logic is executed by the TeamCity server and thus TeamCity server needs to access the VCS server in any mode.

Depending on the version control used, agents can require command line clients installed and available in PATH on the agents (e.g. Perforce, Git, Mercurial).

The checkout mode is configured on the build configuration's Version Control Settings page, in the Checkout Options section (an advanced setting).

TeamCity has the following VCS checkout modes:

Checkout mode	Description
Prefer to checkout files on agent	<p>This is the default setting for the newly created build configurations. When upgrading, the checkout mode settings for existing build configurations are preserved.</p> <p>With this setting enabled, TeamCity will use the agent-side checkout <a href="#">described below</a> if possible. If the agent-side checkout is not possible, TeamCity will display a corresponding <a href="#">health report item</a> and will use the server-side checkout <a href="#">described below</a>.</p> <p>TeamCity falls back to the server-side checkout in the following cases:</p> <ul style="list-style-type: none"><li>• No Git or Mercurial client is found on the agent</li><li>• The Git or Mercurial client is present on the agent, but is of the wrong version</li><li>• The agent has no access to the repository</li><li>• If a Perforce client cannot be found on the agent using the same rules as while performing actual checkout or if stream depot is used and the checkout rules are complex (other than . =&gt; A )</li></ul>
Always checkout files on server	<p>The TeamCity server will <a href="#">export the sources</a> and pass them to an agent before each build. Since the sources are exported rather than checked out, no administrative data is stored in the agent's file system and version control operations (like check-in, label or update) cannot be performed from the agent. TeamCity optimizes communications with the VCS servers by <a href="#">caching the sources</a> and retrieving from the VCS server only the necessary changes. Unless <a href="#">clean checkout</a> is performed, the server sends to the agent incremental patches to update only the files changed since the last build on the agent in the given checkout directory.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"><p> Notes</p><ul style="list-style-type: none"><li>• The server side checkout simplifies administration overhead. Using this checkout mode, you need to install VCS client software on the server only (applicable to Perforce, Mercurial, TFS, Clearcase, VSS). Network access to VCS repository can also be opened to the server only. Thus, if you want to control who has access to the source repositories, the server side checkout is usually more effective.</li><li>• In some cases this checkout mode can lower the load produced on VCS repositories, especially if <a href="#">Clean Checkout</a> is performed often, due to the caching of clean patches by the server.</li><li>• Note that in the server checkout mode the administration directories (like .svn, CVS) are not created on the agent.</li></ul></div>
Always checkout files on agent	<p>The build agent will <a href="#">check out the sources</a> before the build. Agent-side checkout frees more server resources and provides the ability to access version control-specific directories (.svn, CVS, .git); that is, the build script can perform VCS operations (e.g. check-ins into the version control) — in this case ensure the build script uses credentials necessary for the check-in.</p> <p>VCS client software has to be installed on the agent (applicable to Perforce, Mercurial, Git)</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"><p> Notes</p><ul style="list-style-type: none"><li>• Agent checkout is usually more effective with regard to data transfers and VCS server communications. The agent side checkout creates necessary administration directories (like .svn, CVS), and thus allows you to communicate with the repository from the build: commit changes and so on.</li><li>• Machine-specific settings (like configuring SSL communications, etc.) have to be configured on each machine using agent-side checkout.</li><li>• "Exclude" <a href="#">VCS Checkout Rules</a> in most cases cannot improve agent checkout performance because an agent checks out the entire top-level directory included into a build, then deletes the files that were excluded. Perforce and TFS are exceptions to the rule, because before performing checkout, specific client mapping(Perforce)/workspace(TFS) is created based on checkout rules. "Exclude" checkout rules are not supported for Git and Mercurial when using checkout on an agent due to these DVCS limitations. There is a <a href="#">known issue</a> with CVS VCS root ignoring exclude checkout rules when using checkout on an agent.</li><li>• Integration with certain version controls can provide additional options when agent-side checkout is used. For example, <a href="#">Subversion</a>.</li></ul></div>

Do not check out files automatically	<p>TeamCity will not check out any sources automatically, the <a href="#">default build checkout directory</a> will still be created so that you could use it to check out the sources via a build script. Please note that TeamCity will accurately report changes only if the checkout is performed on the revision specified by the <a href="#">build.vcs.number</a>. * properties passed into the build.</p> <p> The build checkout directory will not be cleaned automatically, unless the directory expiration period is configured.</p>
--------------------------------------	---

See also:

[Administrator's Guide: VCS Checkout Rules](#)

## Storing Project Settings in Version Control

On this page:

- [Overview](#)
- [Synchronizing Settings with VCS](#)
  - [Defining Settings to Apply to Builds](#)
    - [Limitations of "use project settings from VCS" mode](#)
- [Storing Secure Settings](#)
  - [Generating Tokens](#)
  - [Implications of Storing Security Data in VCS](#)
- [Settings Format](#)
- [Committing Current Project Settings to VCS](#)
- [Displaying Changes](#)
- [Enabling Versioned Settings after TeamCity Upgrade](#)
- [FAQs](#)

### Overview

TeamCity allows the two-way synchronization of the project settings with the version control repository. Supported VCSs are Git, Mercurial, Perforce, Subversion, and TFS.

You can store settings in the XML format and in the [Kotlin language](#) and define settings programmatically using the [kotlin-based DSL](#).

When you enable two-way settings synchronization:

- each administrative change made to the project settings in the TeamCity Web UI is committed to the version control; the changes are made noting the TeamCity user as the committer;
  - if the settings change is committed to the version control, the TeamCity server will detect the modifications and apply them to the project on the fly.
- Before applying the newly checked-in settings, certain constraints are applied. If the constraints are not met (i.e. the settings are invalid), the current settings are left intact and an error is shown in the UI. Invalid settings are those that cannot be loaded because of constraints, for instance, a build configuration referencing a non-existing VCS root, or having a duplicate id or a duplicate name, etc.

The settings in the VCS are stored in the `.teamcity` folder in the root of the repository the same format as in the TeamCity Data Directory.

## Synchronizing Settings with VCS

By default, the synchronization of the project settings with the version control is disabled.

To enable it, go to Project Settings | Versioned Settings.

**i** The "Enable/disable versioned settings" permission is required (default for the System Administrator role).

The Configuration tab is used to define:

- whether the synchronization settings are the same as in the parent project
- whether the synchronization is enabled.
  - when synchronization is enabled, you can define which settings to use when build starts. See details [below](#).
- which VCS Root is used to store the project settings: you can store the settings either in the same repository as the source code, or in a dedicated VCS root.

Enabling synchronization for a project also enables it for all its subprojects. TeamCity synchronizes all changes to the project settings (including modifications of [build configurations](#), [templates](#), [VCS roots](#), etc.) with the exception of [SSH keys](#).

**i** You can override the synchronization settings inherited from a project for a subproject.

As soon as synchronization is enabled in a project, TeamCity will make an initial commit in the selected repository for the whole project tree (the project with all its subprojects) to store the current settings from the server. If the settings for the given project are found in the specified VCS root (the VCS root for the parent project settings or the user-selected VCS root), a warning will be displayed asking if TeamCity should:

- overwrite the settings in the VCS with the current project settings on the TeamCity server
- import the settings from the VCS replacing the the current project settings on the TeamCity server with those from version control

## Defining Settings to Apply to Builds

There are 2 sources of build settings: the current settings on the TeamCity server, i.e. the latest settings changes applied to the server (either made via the UI, or via a commit into the `.teamcity` directory in the VCS root) and the settings in the VCS on the revision selected for build.

Therefore it is possible to start builds with settings different from those currently defined in the build configuration. For projects where versioned settings are enabled, you can tell TeamCity which settings to take when build starts.

This gives you several possibilities:

- if you are using TeamCity [feature branches](#), you can define a branch specification in the VCS root used for versioned settings, and TeamCity will run a build in a branch using the settings from this branch
- you can now start a [personal build](#) with changes made in the `.teamcity` directory, and these changes will affect the build behavior.
- When running a [history build](#), TeamCity will attempt to use the settings corresponding to the moment of the selected change. Otherwise, the current project settings will be used.

Before starting a build, TeamCity stores configuration for this build in build internal artifacts under `.teamcity/settings` directory. These configuration files can be examined later to understand what settings were actually used by the build.

To define which settings to take when build starts, select one of the following options (on the Project Settings | Versioned Settings page, Configuration tab, click Show advanced options):

- always use current settings: when this option is set, all builds use current project settings from the TeamCity server. Settings changes in branches, history and personal builds are ignored. Users cannot run a build with custom project settings.
- use current settings by default: when this option is set, a build uses the latest project settings from the TeamCity server. Users can run a build with older project settings via the [custom build dialog](#).
- use project settings from VCS:
  - when this option is set, builds in branches and history builds, which use settings from VCS, load settings from the versioned settings revision calculated for the build. Users can change configuration settings in [personal builds from IDE](#) or can run a build with project settings current on the TeamCity server via the [custom build dialog](#).

Limitations of "use project settings from VCS" mode

There are some limitations when the "use project settings from VCS" option is selected and a build on a branch, or a personal, or a history build is run. Certain settings will be ignored and current settings will be used instead. This applies for:

- VCS roots and checkout rules
- Snapshot dependencies
- Certain build Failure Conditions and Build Features which are processed on the server-side (like fail build on message, automatic merge and VCS labeling)
- Build Configuration-level settings not affecting the build directly, like Build Triggers or number of simultaneously running builds

## Storing Secure Settings

It is recommended to store security data outside the VCS. The Project Settings | Versioned Settings | Configuration page has an option to store passwords, API tokens, and other secure settings outside of VCS. This option is enabled by default if versioned settings are enabled for a project for the first time, and not enabled for projects already storing their settings in the VCS.

If this option is enabled, TeamCity stores a random generated id in XML configuration files instead of the scrambled passwords. Actual passwords are stored on the disk under TeamCity data directory and are not checked into the version control system.

 If this option is not enabled, the [security implications](#) listed below should be taken into account before committing security data to the VCS.

## Generating Tokens

When you need to add a password into the versioned settings not via TeamCity UI (e.g. adding settings with Kotlin-based DSL), you will need to add the password to TeamCity and get the corresponding token to use in the settings. The token can be generated via the "Generate Token for password" action available in the Project -> Actions menu.

At this time passwords are not inheritable by projects hierarchy. If a setting in a project (a VCS root, OAuth connection, cloud profile) requires a password, the token generated for this password can be used in this project only. For instance, it is not possible to take a generated token and use it in a similar setting in a subproject. A new token should be generated in this case.

If you need to use a secure value in the nested projects, consider adding a [password parameter](#) with the secure value and using a [reference](#) to the parameter in the nested projects.

## Implications of Storing Security Data in VCS

If you are using a version prior to TeamCity 2017.1 it is recommended to carefully consider the implications of storing security settings in a VCS.

- If the projects or build configurations with settings in a VCS have password fields defined, the values appear in the settings committed into the VCS (though, in scrambled form).
  - If the project settings are stored in the same repository as the source code, anyone with access to the repository will be able to see these scrambled passwords.
  - If the project settings are stored separately from the source code in a dedicated repository, and the Show settings changes in builds option is enabled, any user having the "View VCS file content" permission can see all the changes in the TeamCity UI using the [changes difference viewer](#).
- Being able to change the settings in an arbitrary manner via a VCS, it is possible to trigger builds of any build configurations and obtain settings of any build configurations irrespective of the build configurations permissions configured in TeamCity.
- by committing wrong or malicious settings, a user can affect the entire server performance or server presentation to other users.

It is recommended to store passwords, API tokens, and other secure settings outside of VCS using the corresponding option described above.

Note that SSH keys will not be stored in the VCS repository.

## Settings Format

You can select the settings format: on the Versioned Settings | Configuration page for your project, click Show advanced options.

TeamCity stores project settings:

- in the XML format

- in kotlin-based DSL (see a [dedicated page](#)).



To test kotlin-based settings in a sandbox project, you can download the settings as a zip archive without disabling the UI: on the project settings page, click the Actions menu and select Download settings in kotlin format. A zip archive will be downloaded.

## Committing Current Project Settings to VCS



Before committing settings to the VCS, consider the recommended approach to storing security settings [described above](#).

If you want to commit the current configuration to the VCS (e.g. earlier you committed misconfigured settings to the repository and TeamCity was unable to load it displaying errors and warnings), you can use the Commit current project settings... option on the Versioned Settings | Configuration page.

When TeamCity commits settings into a VCS, it uses a standard commit message noting the TeamCity user as the committer and the project whose settings changed. It is possible to add a fixed custom prefix to each settings change committed by TeamCity via the `teamcity.versionedSettings.commitMessagePrefix` internal property, e.g. `teamcity.versionedSettings.commitMessagePrefix=TC Change\n\n`

## Displaying Changes

TeamCity will not only synchronize the settings, but will also automatically display changes to the project settings the same way it is done for regular changes in the version control. You can configure the changes to be displayed for the affected build configurations: on the Project Settings | Versioned Settings page, Configuration tab, click Show advanced options and check the Show settings changes in builds box.

By default, the VCS trigger will ignore such changes. To enable build triggering on a settings commit, add a trigger rule in the following format: `+:root=Settings_root_id;:*`

All changes in the VCS root where project settings are stored are listed on the Versioned Settings | Change log tab of the Versioned Settings page.

## Enabling Versioned Settings after TeamCity Upgrade

The format of the XML settings files changes from one TeamCity version to another to accommodate the new features and improvements. Generally, the format is not changed within bugfix releases and is changed in minor/major releases. When a TeamCity server is upgraded, the current settings on the TeamCity server are changed from the earlier to the current format.

It is a common practice to upgrade a TeamCity test server with production data before upgrading the production server. In order to avoid accidentally changing the format of the settings which are used on a production server of an older version, versioned settings are disabled after a TeamCity upgrade and the corresponding health item is displayed. System administrators have permissions to enable versioned settings. When enabled, the converted settings in the format of the current TeamCity version will be checked in to the version control. Note that the new settings will be committed to the default branch of the VCS root; the settings stored in other branches will have to be updated manually.

## FAQs

**Q. Can I apply the settings from a TeamCity server of a different version?**

**A.** No, because just like with the TeamCity Data Directory, the format of the settings differs from one TeamCity version to another.

**Q. Where are the settings stored?**

**A.** The settings are stored in the `.teamcity` folder in the root of the VCS root-configured repository path. For Git and Mercurial this is always the root of the repository. The default branch configured in the VCS root is used with [Git](#), [Mercurial](#). You can create a dedicated VCS root to change the branch (or a repository path in case of Perforce, Subversion or TFS).

**Q. Why is there a delay before a build is run after I changed to the settings in the UI?**

**A.** When the settings are changed via the UI, TeamCity will wait for the changes to be completed with a commit to the VCS before running a build with the latest changes.

**Q. Who are the changes authored by?**

**A.** If the settings are changed via the user interface, in Git and Mercurial a commit in the VCS will be performed on behalf of the user who actually made the change via the UI. For Perforce as well as TFS, the name of the user specified in the VCS root is used, and in Subversion the commit message will also contain the username of the TeamCity user who actually made the change via the UI.

See also:

[Administrator's Guide: Kotlin DSL](#)

## Kotlin DSL

Besides storing settings in version control in XML format, TeamCity also allows storing the settings in the DSL (based on [the Kotlin language](#)).

Using the version control-stored DSL enables you to define settings programmatically. Since Kotlin is statically typed, you automatically receive the auto-completion feature in IDE which makes the discovery of available API options a much simpler task.

Check also our blog post series on [Kotlin DSL](#).

On this page:

- [Getting Started with Kotlin DSL](#)
  - [Project Settings Structure](#)
  - [Opening Project in IntelliJ IDEA](#)
  - [Editing Kotlin DSL](#)
  - [Patches](#)
  - [Sharing Kotlin DSL Scripts](#)
- [Advanced Topics](#)
  - [Non-Portable DSL](#)
    - [settings.kts](#)
    - [Project.kt](#)
  - [Debugging Maven 'generate' Task](#)
  - [Ability to Use External Libraries](#)
- [FAQ and Common Problems](#)
  - [How to Add .teamcity as a New Module to a Project?](#)
  - [New URL of Settings VCS Root \(Non portable format\)](#)
  - [How to Read Files in Kotlin DSL](#)
  - [Passwords-Related Questions](#)

### Getting Started with Kotlin DSL

The Kotlin tutorial helps you learn most Kotlin features in a few hours.

You can start working with Kotlin DSL by creating an empty sandbox project on your server and follow the steps below:

1. [Enable versioned settings for your project.](#)

2. Select Kotlin as the format. Make sure the Generate portable DSL scripts option is enabled.
3. Click Apply and TeamCity will commit the settings to your repository.

 After enabling Kotlin for project settings, editing of the project in the web UI may not be available right after the switch. TeamCity needs to detect its own commit in the repository, and only after that editing will be enabled. Usually it takes a minute or two.

## Project Settings Structure

After the commit to the repository, you will get the the .teamcity settings directory with the following files:

- pom.xml
- settings.kts

settings.kts is the main file containing all the project configuration, that's the file we will use to change project settings. pom.xml is only required to open the project in an IDE to get the auto-completion feature as well as ability to compile code, write unit tests for it, etc.

## Opening Project in IntelliJ IDEA

To open the Kotlin DSL project in IntelliJ IDEA, open the .teamcity/pom.xml file as a project. All necessary dependencies will be resolved automatically right away. If all dependencies have been resolved, no errors in red will be visible in the settings.kts. If you already have an IntelliJ IDEA project and want to add Kotlin DSL module to it, see [this section](#).

## Editing Kotlin DSL

If you created an empty project, that's what you'll see in your IDE when you open settings.kts:

```
import jetbrains.buildServer.configs.kotlin.v2018_1.*
/* some comment text */
version = "2018.1"

project {
```

project { } represents the current project whose settings we'll define in the DSL. This is the same project where we enabled versioned settings on the previous step. This project ID and name can be accessed via a special `DslContext` object but cannot be changed via the DSL.

You can create different entities in this project by calling `vcsRoot()`, `buildType()`, `template()`, or `subProject()` methods.

For instance, to add a build configuration with a command line script, we can do the following:

```
import jetbrains.buildServer.configs.kotlin.v2018_1.*
import jetbrains.buildServer.configs.kotlin.v2018_1.buildSteps.script

version = "2018.1"

project {
    buildType {
        id("HelloWorld")
        name = "Hello world"
        steps {
            script {
                scriptContent = "echo 'Hello world!''"
            }
        }
    }
}
```

After that you can submit this change to the repository, TeamCity will detect it and apply. If there are no errors during the script execution, then we should see a build configuration named "Hello world" in our project.

**⚠** To get familiar with Kotlin API, see the online documentation on your local server, accessible via the link on the Versioned Settings project tab in the UI or by running the `mvn -U dependency:sources` command in the IDE. Refer to the documentation on the public TeamCity server as an [example](#).

You can also use the Download settings in Kotlin format option from the project Actions menu. For instance, you can find a project that defines some settings that you want to use in your Kotlin DSL project and use this "download" action to see what the DSL generated by TeamCity looks like.

## Patches

TeamCity allows editing of a project via the web interface, even though the project settings are stored in Kotlin DSL. For every change made in the project via the web interface, TeamCity will generate a patch in the Kotlin format, which will be checked in under the project patches directory with subdirectories for different TeamCity entities. For example, if you change a build configuration, TeamCity will submit the `.teamcity/patches/buildTypes/<id>.kt` script to the repository with necessary changes.

For instance, the following patch adds the [Build files cleaner \(Swabra\)](#) build feature to the build configuration with the ID SampleProject\_Build:

```
changeBuildType(RelativeId("SampleProject_Build")) { // this part finds the build configuration where  
the change has to be done  
    features {  
        add {  
            // this is the part which should be copied to a corresponding place of the settings.kts file  
            swabra {  
                filesCleanup = Swabra.FilesCleanup.DISABLED  
            }  
        }  
    }  
}
```

It is implied that you move the changes from the patch file to you settings.kts and delete the patch file.

## Sharing Kotlin DSL Scripts

Besides simplicity, one of the advantages of the portable DSL script is that the script can be used by more than one project or more than one server (hence the name: portable).

If you have a repository with `.teamcity` containing settings in portable format, then you can easily create another project based on these settings. The [Create Project From URL](#) feature can be used for this.

Simply point TeamCity to your repository and it will detect the `.teamcity` directory and offer to import settings from there:

Administration / <Root project>

### Create Project From URL

✓ The connection to the VCS repository has been verified

⚠ The VCS repository contains `.teamcity/settings.kts` file with settings of some project. Would you like to import these settings?

Import settings from `.teamcity/settings.kts`

Do not import settings, create project from scratch

**⚠** The `settings.kts` script can always access a `DslContext` object which contains the id and some other settings of the current project.

Based on the context, the DSL script can generate slightly different settings, for instance:

```

var deployTarget = if (DslContext.projectId == AbsoluteId("Trunk")) {
    "staging"
} else {
    "production"
}

```

## Advanced Topics

### Non-Portable DSL

Versions before 2018.1 used a different format for Kotlin DSL settings. This format can still be enabled by turning off the Generate portable DSL scripts checkbox on versioned settings page.

When TeamCity generates non-portable DSL, the project structure in the .teamcity directory looks as follows:

- pom.xml
- <project id>/settings.kts
- <project id>/Project.kt
- <project id>/buildTypes/<build conf id>.kt
- <project id>/vcsRoots/<vcs root id>.kt

where <project id> is the ID of the project where versioned settings are enabled. The Kotlin DSL files producing build configurations and VCS roots are placed under the corresponding subdirectories.

### settings.kts

In the non-portable format each project has the following settings.kts file:

```

package MyProject
import jetbrains.buildServer.configs.kotlin.v2018_1.*
/* ... */
version = "2018.1"

project(MyProjectId.Project)

```

This is the entry point for project settings generation. Basically, it represents a Project instance which generates project settings.

### Project.kt

The Project.kt file looks as follows:

```

package MyPackage
import jetbrains.buildServer.configs.kotlin.v2018_1.*
import jetbrains.buildServer.configs.kotlin.v2018_1.Project

object Project : Project({
    uuid = "05acd964-b90f-4493-aa09-c2229f8c76c0"
    id("MyProjectId")
    parentId("MyParent")
    name = "My project"
    ...
})

```

where:

- `id` is the absolute id of the project, the same id we'll see in browser address bar if we navigate to this project
- `parentId` is the absolute id of a parent project where this project is attached

- `uuid` is some unique sequence of characters.

The `uuid` is a unique identifier which associates a project, build configuration or VCS root with its data. If the `uuid` is changed, then the data is lost. The only way to restore the data is to revert the `uuid` to the original value. On the other hand, the `id` of an entity can be changed freely, if the `uuid` remains the same. This is the main difference of the non-portable DSL format from portable. The portable format does not require specifying the `uuid`, but if it happened so that a build configuration lost its history, one has to reattach it again via the web interface.



If the build history is important, it should be restored as soon as possible: after the deletion, there is a [configurable timeout](#) (5 days by default) before the builds of the deleted configuration are removed during the build history clean-up.

In case of non-portable DSL, patches are stored under the project patches directory of `.teamcity`:

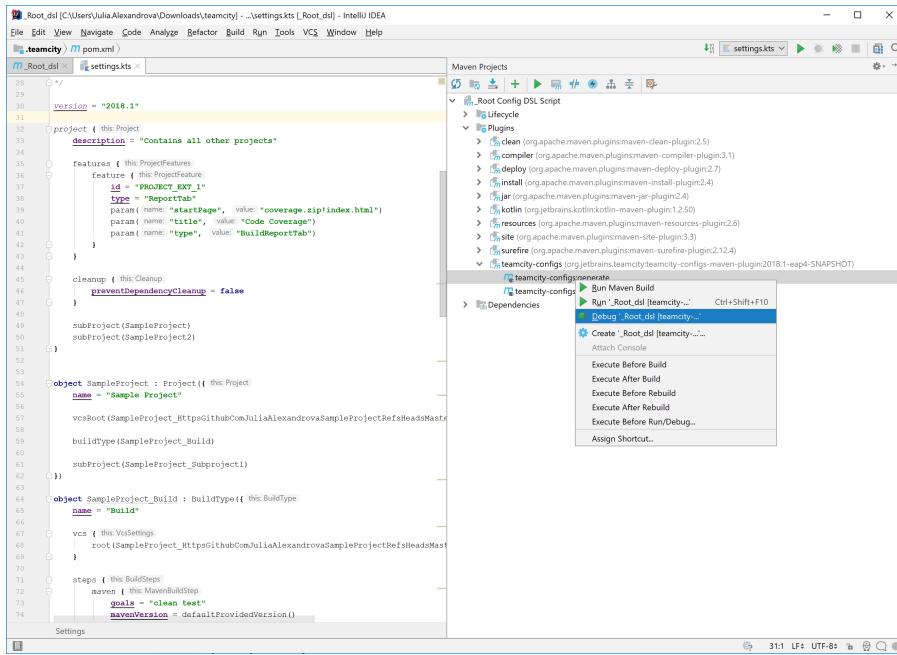
- `pom.xml`
- `<project id>/settings.kts`
- `<project id>/Project.kt`
- `<project id>/patches/<uuid>.kt`

Working with patches is the same as in portable DSL: you need to move the actual settings from the patch to your script and remove the patch.

#### Debugging Maven 'generate' Task

The `pom.xml` file provided for a Kotlin project has the `generate` task, which can be used to generate TeamCity XML files locally from the Kotlin DSL files. This task supports debugging. If you're using IntelliJ IDEA, you can easily start debugging of a Maven task:

1. Navigate to View | Tool Windows | Maven Projects. The Maven Projects tool window is displayed.
2. Locate the task node: Plugins | `teamcity-configs` | `teamcity-configs:generate`, the Debug option is available in the context menu for the task:



#### Ability to Use External Libraries

You can use external libraries in your Kotlin DSL code, which allows sharing code between different Kotlin DSL-based projects.

To use an external library in your Kotlin DSL code, add a dependency on this library to the `.teamcity/pom.xml` file in the settings repository and commit this change so that TeamCity detects it. Then, before starting the generation process, the TeamCity server will fetch the necessary dependencies from the Maven repository, compile code with them, and then start the settings generator.

#### FAQ and Common Problems

##### How to Add `.teamcity` as a New Module to a Project?

Question: How to add the .teamcity settings directory as a new module to an existing project in IntelliJ IDEA?

Solution: In your existing project in IntelliJ IDEA:

- Go to File | Project Structure, or press Ctrl+Shift+Alt+S.
- Select Modules under the Project Settings section.
- Click the plus sign, select Import module and choose the folder containing your project settings. Click Ok and follow the wizard.
- Click Apply. The new module is added to your project structure.

New URL of Settings VCS Root (Non portable format)

Problem: I changed the URL of the VCS root where settings are stored in Kotlin and now TeamCity cannot find any settings in the repository at the new location.

Solution:

- Fix the URL in the Kotlin DSL in the version control and push the fix.
- Disable versioned settings to enable the UI.
- Fix the URL in the VCS root in the UI.
- [Enable versioned settings](#) with same VCS root and the Kotlin format again, TeamCity will detect that the repository contains the .teamcity directory and ask you if you want to import settings.
- Choose to import settings.

How to Read Files in Kotlin DSL

Problem: I want to generate a TeamCity build configuration based on the data in some file residing in the VCS inside the .teamcity directory

Solution: TeamCity executes DSL with the .teamcity as the current directory, so files can be read using the paths relative to the .teamcity directory e.g. File("data/setup.xml"). Files outside the .teamcity directory are not accessible to Kotlin DSL.

Passwords-Related Questions

Prior to TeamCity 2017.1

Problem: I do not want the passwords to be committed to the VCS, even in a scrambled form.

Solution: You can move the passwords to the parent project whose settings are not committed to a VCS.

Problem: I want to change passwords after the settings have been generated.

Solution: The passwords will have to be scrambled manually using the following command in the maven project with settings:

```
mvn -Dtext=<text to scramble> org.jetbrains.teamcity:teamcity-configs-maven-plugin:scramble
```

Since TeamCity 2017.1

Solution: Use tokens instead of passwords. Please refer to the [related section](#).

See also:

[Administrator's Guide: Storing Project Settings in Version Control](#)

Upgrading DSL

- [TeamCity settings format changes](#)
  - Changes which can be performed automatically
  - Changes which cannot be performed automatically
- [Versions in DSL code](#)
  - Maven dependency version
  - Kotlin DSL API version
  - Configs version
- [DSL upgrade procedure](#)

- Update DSL from 2017.2.x to 2018.1.x
  - Updating Docker parameters
    - Docker push command support
  - Adding label property to Gerrit publisher settings in Commit Status Publisher
- Update DSL from 2017.1.x to 2017.2.x
  - DSL version
  - Updating Docker parameters
  - Updating default project template
  - Updating .NET CLI parameters
    - Common parameters
    - dotnet build
    - dotnet clean
    - dotnet msbuild
    - dotnet nuget delete
    - dotnet nuget push
    - dotnet pack
    - dotnet publish
    - dotnet restore
    - dotnet run
    - dotnet test
    - dotnet vstest
- Update DSL from 10.0.x to 2017.1.x
  - Changes requiring manual Kotlin DSL scripts update
    - Converting cloud profiles
    - Names in entities
  - Updating Kotlin DSL configs version to 10.0 to 2017.1
    - Updating DotCover parameters
    - Updating Maven build step parameters
    - Updating PowerShell build step parameters

The TeamCity XML settings format is usually [changed](#) in the major releases. During the first start after the server update, TeamCity converts XML settings files in the TeamCity Data directory to the new format.

When settings are stored in Kotlin DSL, the Kotlin code might need to be changed to still be functional. It is also recommended to update the Kotlin code to the latest config version with each version.

These recommendations are reported as server health reports and are shown in corresponding locations on the server administration UI.

#### TeamCity settings format changes

As far as the DSL is concerned, there are the following types of TeamCity settings changes.

##### Changes which can be performed automatically

These types of settings changes do not require changing the Kotlin DSL as the changes are applied by the server automatically on each settings regeneration from the DSL. It is recommended though to update the DSL to newer configs version to reduce performance hit and make Kotlin scripts closer to the settings you see in UI.

A lot of changes in TeamCity settings fall into this category. For example, some plugin implementing a build step can rename its parameters. The DSL from the previous TeamCity version generates a parameter with the old name, but TeamCity can automatically replace the old parameter name with the new one after DSL execution.

##### Changes which cannot be performed automatically

Some TeamCity settings changes require external information and cannot be performed automatically. For example, in TeamCity 10.0 the settings of the cloud integration were stored in a dedicated file which was not committed to VCS. In TeamCity 2017.1 these settings were moved to the project level. TeamCity cannot perform such a transformation of settings automatically without external data, so manual DSL code update is required.

#### Versions in DSL code

##### Maven dependency version

This is the version specified in pom.xml, it looks like this:

```
<teamcity.dsl.version>10.0.4</teamcity.dsl.version>.
```

It needs to be updated to the version of the TeamCity server you use.

##### Kotlin DSL API version

This is the version included in the Kotlin DSL API package name: `jetbrains.buildServer.configs.kotlin.v10`. New backward-incompatible API will be provided in a new package and the API from older versions will continue to work.

## Configs version

This is the version specified in `settings.kts`, it looks like this:

```
version = "10.0"  
project(Some_Project)
```

TeamCity uses this version to decide which transformations should be performed after DSL execution to make settings up-to-date.

## DSL upgrade procedure

After TeamCity upgrade, the versioned settings are disabled globally on the whole TeamCity server and a corresponding health report is shown in the administration UI. This is done to prevent TeamCity from changing the settings in the version control if you upgrade a non-production copy of the TeamCity server.

If the server is a production installation, enable versioned settings using the action in the health report. This will make TeamCity commit converted XML file to the VCS. For settings in Kotlin format the files inside `plugin-settings` directory and meta-runners will be committed.

At this point, the version control has the previous version of Kotlin DSL scripts.

If some of the settings conversions cannot be applied automatically, TeamCity will disable the versioned settings for the affected projects and a corresponding health report is shown. You should update the DSL as per the notes below and then the versioned settings can be enabled in the project.

For the projects with settings in Kotlin DSL which were not affected by the settings conversions or if the settings can be converted automatically, the Kotlin DSL scripts do not require any immediate changes and you can continue using the projects as usual (versioned settings stay enabled in such projects). However, it is recommended to update the Kotlin scripts to the up-to-date configs version. TeamCity displays a health report for each affected project with instructions on how to update DSL. In order to check if DSL code update is required, load the settings from VCS and see the Current status on the Versioned settings tab: it will show a warning if update is required.

## Update DSL from 2017.2.x to 2018.1.x

This release introduces the new DSL API version, `v2018_1`, its package is `jetbrains.buildServer.configs.kotlin.v2018_1`. The previous API versions work and you can keep using them if you don't want to use portable DSL scripts.

## Updating Docker parameters

If you used Kotlin DSL with TeamCity 2017.2 for Docker plugin configurations, you may need to perform some changes in your Kotlin configuration scripts to make your code compatible with TeamCity 2018.1.

The essence of the changes:

- build step `dockerBuild` is now converted to `dockerCommand`
- the parameters for `DockerBuild` are now sub-parameter for `commandType` selector

Please see the difference:

## Docker Build 2 Docker Command migration

```
# DockerBuild in 2017.2:  
dockerBuild {  
    name = "name of the build step"  
    source = content {  
        content = """  
        FROM busybox  
  
        MAINTAINER KIR <kir@maxkir.com>  
        """".trimIndent()  
    }  
    namesAndTags = "maxkir/maxkir_test"  
}  
#-----  
# DockerBuild in 2018.1:  
dockerCommand {  
    name = "name of the build step"  
    commandType = build {  
        source = content {  
            content = """  
            FROM busybox  
  
            MAINTAINER KIR <kir@maxkir.com>  
            """".trimIndent()  
        }  
        namesAndTags = "maxkir/maxkir_test"  
    }  
}
```

## Docker push command support

### Docker Push

```
dockerCommand {  
    name = "name of the push step"  
    commandType = push {  
        namesAndTags = "maxkir/maxkir_test"  
    }  
}
```

## Adding label property to Gerrit publisher settings in Commit Status Publisher

Since TeamCity 2018.1 Gerrit publisher settings in the [Commit Status Publisher](#) build feature allow customising the Gerrit label that reflects the build status, i.e. now you can use something other than Verified.

To manually modify Kotlin DSL settings, the `label = "Verified"` statement must be added as follows:

```

features {
    ...
    commitStatusPublisher {
        publisher = gerrit {
            server = "<server>"
            gerritProject = "<project>"
            failureVote = "-1"
            successVote = "+1"
            userName = "<user>"
            uploadedKey = "<key>"
            label = "Verified" // the statement to be added
        }
    }
    ...
}

```

Update DSL from 2017.1.x to 2017.2.x

#### DSL version

This release introduces the new DSL API version, `v2017_2`. The previous API version works and you can keep using it if you do not need [the features provided by the new API](#).

 To get sample DSL code for the newly supported features without switching a project to the Kotlin format, use the Download settings in Kotlin format action on the project administration page.

If you used 2017.2 EAPs and tested changing DSL settings via the UI, you need to apply all the UI patches created by TeamCity before upgrading as some API is changed in an incompatible way.

The current package name for the settings generated by TeamCity is `jetbrains.buildServer.configs.kotlin.v2017_2`.

With the new API, you can benefit from a number of [new features](#), including the editable administration UI for Kotlin DSL projects and DSL documentation. To use them for your existing project, your `.kt` files should be switched to packages from the `v2017_2` version.

- To compile this project, you also need to update your `pom.xml`. The easiest way is to invoke the Download settings in Kotlin format action in your project and copy the `pom.xml` from the generated zip archive. Alternatively you can update `pom.xml` yourself with the following:
  - change the `teamcity.dsl.version` to: `<teamcity.dsl.version>2017.2</teamcity.dsl.version>`
  - change the `kotlin` version to: `<kotlin.version>1.1.4-3</kotlin.version>` and add a dependency on `kotlin-runtime` and `kotlin-reflect`:

```

<dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-runtime</artifactId>
    <version>${kotlin.version}</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>org.jetbrains.kotlin</groupId>
    <artifactId>kotlin-reflect</artifactId>
    <version>${kotlin.version}</version>
    <scope>compile</scope>
</dependency>

```

Once you switch the project to new API and check in the changes, TeamCity will detect and apply them and after that web UI editing will be enabled.

To use the new DSL API in a repository with an existing `pom.xml`, the maven dependency version has to be updated to 2017.2.

## Updating Docker parameters

If you used Kotlin DSL with TeamCity 2017.1 for Docker plugin configurations, you may need to perform some changes in your kotlin configuration scripts to make your code compatible with TeamCity 2017.2. Basically, TeamCity provides a converter for Kotlin configuration scripts to perform those changes automatically, but if it does not work due to some reason, the following changes need to be made manually:

- rename the build runner step with the Docker Build runType to DockerBuild
- rename the build runner step with the Docker Compose runType to DockerCompose
- rename the docker-compose.file build parameter to dockerCompose.file

Starting from TeamCity 2017.2, the Docker plugin has its own typed DSL for the [Docker build feature](#), [Docker Wrapper](#), [Docker](#), and [Docker Compose](#) runners.

### Docker Compose and Docker Build

```
steps {
    dockerCompose {
        name = "Start services"
        file = "db-tests/scripts/docker-compose.yml"
    }

    dockerBuild {
        name = "Docker build step"
        source = path {
            path = "some/context/Dockerfile"
        }

        // Case for Dockerfile specified by an URL:
        //source = url {
        //    url =
        //"https://raw.githubusercontent.com/JetBrains/teamcity-docker-minimal-agent/master/ubuntu/Dockerfile"
        //}
    }

    // Case for Dockerfile specified by text:
    //source = content {
    //    content = """
    //        FROM busybox
    //        MAINTAINER Kirill Maximov <kir@jetbrains.com>
    //        """.trimIndent()
    //}

    contextDir = "some/context"
    namesAndTags = "my:tag"
}
```

## Docker Build Feature

```
features {  
    dockerSupport {  
        cleanupPushedImages = true  
        loginToRegistry = on {  
            dockerRegistryId = "PROJECT_EXT_2"  
        }  
    }  
}
```

## Command Line runner with Docker Wrapper

```
script {  
    scriptContent = "ls -IR"  
    dockerImage = "openjdk:8u121"  
    dockerPull = true  
    dockerRunParameters = "--rm -v /some/path:/another/path:ro"  
}
```

## Updating default project template

Since 2017.2 EAP3 TeamCity supports default templates in projects. This setting was stored in a project feature of the "DefaultTemplate" type in EAP3 and EAP4, but since 2017.2 RC the project configuration schema was changed to accommodate for default templates.

To manually convert a DSL project configuration that employs the default template, you will have to delete a corresponding project feature and replace it with the defaultTemplate property assignment as follows:

2017.2 EAP3/4 DSL project configuration with a default template configured:

```
object Project : Project({  
    uuid = "2b241ffb-9019-4e60-9a3a-d5475ab1f312"  
    extId = "ExampleProject"  
    parentId = "_Root"  
    name = "Example Project"  
    ...  
    features {  
        ...  
        feature {  
            id = "PROJECT_EXT_4"  
            type = "DefaultTemplate"  
            param("template", "ExampleProject_MyDefaultTemplate")  
        }  
        ...  
    }  
    ...  
})
```

2017.2 DSL project configuration with the default template configured:

```

object Project : Project({
    uuid = "2b241ffb-9019-4e60-9a3a-d5475ab1f312"
    extId = "ExampleProject"
    parentId = "_Root"
    name = "Example Project"
    defaultTemplate = "ExampleProject_MyDefaultTemplate"
    ...
    features {
        ...
    }
    ...
})

```

#### Updating .NET CLI parameters

Since 2017.2 TeamCity bundles .NET CLI plugin. If you were using Kotlin DSL for the plugin parameters in TeamCity 2017.1.x, you need to change the commands as follows. In TeamCity 2017.1.x:

```

steps {
    step {
        type = "dotnet"
        param("dotnet-command", "build")
        param("dotnet-paths", "WindowsAzure.sln")
    }
}

```

Since TeamCity 2017.2 you could explicitly specify build steps with parameters:

```

steps {
    dotnetBuild {
        projects = "WindowsAzure.sln"
    }
}

```

#### Common parameters

2017.1	2017.2	Comment
dotnet-command	dotnet<Command>	Now the command name reflects the build step name, e.g. dotnetRestore
dotnet-args	args	
dotnet-verbosity	logging	

#### dotnet build

2017.1	2017.2	Comment
dotnet-paths	projects	
dotnet-build-config	configuration	
dotnet-build-framework	framework	
dotnet-build-output	outputDir	

dotnet-build-runtime	runtime	
dotnet-build-no-deps	-	Should be added as the --no-dependencies argument
dotnet-build-not-incremental	-	Should be added as the --no-incremental argument

dotnet clean

2017.1	2017.2
dotnet-paths	projects
dotnet-clean-config	configuration
dotnet-clean-framework	framework
dotnet-clean-output	outputDir
dotnet-clean-runtime	runtime

dotnet msbuild

2017.1	2017.2
dotnet-paths	projects
dotnet-msbuild-config	configuration
dotnet-msbuild-platform	platform
dotnet-msbuild-targets	targets
dotnet-msbuild-runtime	runtime

dotnet nuget delete

2017.1	2017.2
dotnet-nuget-delete-id	packageId
dotnet-nuget-push-source /	packageSource
dotnet-nuget-delete-source	
secure:dotnet-nuget-delete-api-key	apiKey

dotnet nuget push

2017.1	2017.2	Comment
dotnet-paths	packages	
dotnet-nuget-push-source	packageSource	
dotnet-nuget-push-no-symbols	noSymbols	
secure:dotnet-nuget-push-api-key	outputDir	
dotnet-build-runtime	apiKey	
dotnet-nuget-push-no-buffer	-	Should be added as the --disable-buffering true argument

dotnet pack

2017.1	2017.2	Comment
dotnet-paths	projects	
dotnet-pack-config	configuration	
dotnet-pack-runtime	runtime	

dotnet-pack-no-build	skipBuild	
dotnet-pack-output	outputDir	
dotnet-pack-version-suffix	versionSuffix	
dotnet-pack-serviceable	-	Should be added as the --serviceable argument

dotnet publish

2017.1	2017.2
dotnet-paths	projects
dotnet-publish-config	configuration
dotnet-publish-framework	framework
dotnet-publish-output	outputDir
dotnet-publish-runtime	runtime
dotnet-publish-version-suffix	versionSuffix

dotnet restore

2017.1	2017.2	Comment
dotnet-paths	projects	
dotnet-restore-config	configFile	
dotnet-restore-runtime	runtime	
dotnet-restore-packages	packagesDir	
dotnet-restore-source	packageSources	
dotnet-restore-ignore-failed	-	Should be added as the --ignore-failed-sources argument
dotnet-restore-no-cache	-	Should be added as the --no-cache argument
dotnet-restore-parallel	-	Should be added as the --disable-parallel argument
dotnet-restore-root-project	-	Should be added as the --no-dependencies argument

dotnet run

2017.1	2017.2
dotnet-paths	projects
dotnet-run-config	configuration
dotnet-run-framework	framework
dotnet-run-runtime	runtime

dotnet test

2017.1	2017.2
dotnet-paths	projects
dotnet-test-config	configuration
dotnet-test-framework	framework
dotnet-test-no-build	skipBuild
dotnet-test-output	outputDir

dotnet-test-settings-file		settingsFile
dotnet-test-runtime		runtime
dotnet-test-test-case-filter		filter

dotnet vstest

2017.1	2017.2	Comment
dotnet-paths	assemblies	
dotnet-vstest-config-file	settingsFile	
dotnet-vstest-framework	framework	
dotnet-vstest-platform	platform	
dotnet-vstest-filter-type	filter	To filter tests by name, use <code>name { names = "..." }</code> ; for the test case filter, use <code>filter { filter = "..." }</code>
dotnet-vstest-is-isolation	-	Should be added as the <code>/InIsolation</code> argument

Update DSL from 10.0.x to 2017.1.x

TeamCity 2017.1 does not introduce a new Kotlin DSL API, the same package as in TeamCity 10.0.x is used (`jetbrains.buildServer.configs.kotlin.v10`). Several new properties were added to the existing API; to get the latest API for the scripts development, update `teamcity.dsl.version` in `pom.xml` to the `2017.1-SNAPSHOT` for EAP builds and to the `2017.1` for release builds.

#### Changes requiring manual Kotlin DSL scripts update

These changes should be performed before enabling versioned settings in the projects where they were disabled on TeamCity upgrade with the message: "Versioned settings are disabled in this project because its settings files were modified during TeamCity upgrade"

#### Converting cloud profiles

This is only relevant if you use Kotlin DSL for the Root project settings and have cloud profiles.

In 2017.1 cloud profiles were moved from the server level to the Root project level. Since they were not defined in the Kotlin DSL, in case you enable the versioned settings the existing cloud profiles will be deleted from the server. Thus before continuing to use Kotlin DSL for the root project on the server make sure to add the cloud profiles definitions to the root project settings in Kotlin DSL.

To update your settings with the cloud profile information, perform the following:

1. Run the 'Download settings in Kotlin format' action in the Root project and save the zip with the generated DSL
2. Copy project features of type "CloudIntegration" and "CloudProfile" from the `.teamcity/_Root/Project.kt` file to the root project config in your settings
3. Commit your changes to the VCS
4. Enable versioned settings on the 'Versioned Settings' tab of the Root project.

#### Names in entities

Due to the fix of issue [TW-48609](#), if your settings contain nameless entities, TeamCity will report corresponding VCS settings errors. You need to manually set a name parameter to such entities to resolve the errors.

Updating Kotlin DSL configs version to 10.0 to 2017.1

The changes in this section should be done to Kotlin scripts on changing the `scripts config version` from 10.0 to 2017.1

#### Updating DotCover parameters

Parameters used by DotCover were changed and if you use them, make sure the `"dotNetCoverage.tool"` has the `"dotcover"` value. If the `"dotNetCoverage.dotCover.home.path"` parameter is missing, set it to `"%teamcity.tool.JetBrains.dotCover.CommandLineTools.bundled%"`. The result should look like this:

```
param("dotNetCoverage.tool", "dotcover")
param("dotNetCoverage.dotCover.home.path",
"%teamcity.tool.JetBrains.dotCover.CommandLineTools.bundled%")
```

If you use DotCover with custom path (e.g. `/custom/path`), then the result should look like this:

```
param("dotNetCoverage.tool", "dotcover")
param("dotNetCoverage.dotCover.home.path", "/custom/path")
```

#### Updating Maven build step parameters

If you use a typed maven DSL without raw parameters, then this change should not affect you, because typed DSL generates up-to-date settings.

If you specify maven build runner settings via the `param("name", "value")` method, then parameters need to be updated. The easiest way to update settings is to switch to the typed DSL: you can generate settings in the Kotlin format to see what the typed DSL for Maven looks like.

If you want to continue using the `param("name", "value")` method, do the following:

- rename the `"mavenSelection"` parameter to `"maven.path"`, change its old value to the new one:

Old value to be changed	New value
<code>mavenSelection:bundledM2</code>	<code>%teamcity.tool.maven%</code>
<code>mavenSelection:bundledM3_0</code>	<code>%teamcity.tool.maven3%</code>
<code>mavenSelection:bundledM3_1</code>	<code>%teamcity.tool.maven3_1%</code>
<code>mavenSelection:bundledM3_2</code>	<code>%teamcity.tool.maven3_2%</code>
<code>mavenSelection:bundledM3_3</code>	<code>%teamcity.tool.maven3_3%</code>
<code>mavenSelection:default</code>	<code>%teamcity.tool.maven.AUTO%</code>
<code>mavenSelection:custom</code>	the value of the <code>maven.home</code> parameter.

- remove the `maven.home` parameter.

#### Updating PowerShell build step parameters

TeamCity 2017.1 adds support for cross-platform PowerShell. Previously PowerShell builds used only the Desktop edition and were run only on Windows.

To ensure that the existing builds remain restricted to the Desktop edition of PowerShell, set the following property in the existing PowerShell steps:

```
edition = PowerShellStep.Edition.Desktop
```

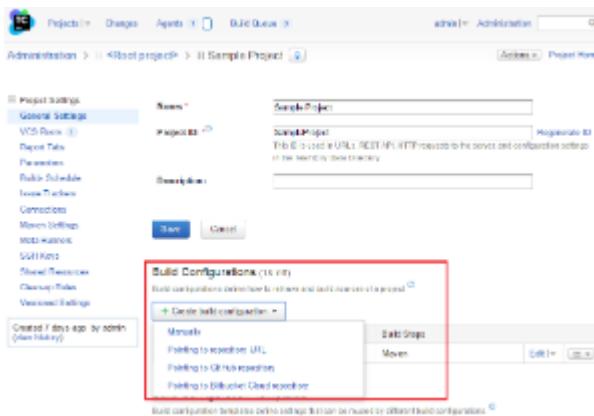
or if you use raw parameters:

```
param("jetbrains_powershell_edition", "Desktop")
```

## Creating and Editing Build Configurations

This section details creating build configurations via the TeamCity web UI. Other options include the [REST API](#) and using TeamCity project configuration in [DSL based on the Kotlin language](#).

TeamCity provides several ways to create a new build configuration for a project using the Project Settings page:



You can create a new build configuration

- Manually
- Pointing to a repository URL
- Pointing to a GitHub repository
- Pointing to Bitbucket repository
- Create a build configuration template, and then [create a build configuration based on the template](#).

When build configurations are created, you can:

- change their order
- edit their settings

### Creating New Build Configuration Manually

1. On the Administration > Projects page, select the desired project in the list. (Alternatively, open the project using the Projects popup, and click the Edit Project Settings link on the right). The Project Settings page opens.
2. On the Project Settings page, Build Configurations section, click Create build configuration.
3. In the Create Build Configuration dialog, specify the name, id and (optionally) description for the build configuration, click Create.
4. Proceed with creating other settings:
  - [Create/edit VCS roots and specify VCS-specific settings](#)
  - On the Build Steps page, configure build steps discovered by the automatic detection. To create them manually by selecting a desired build runner from the drop-down list. Click Save. You can add as many build steps as you need within one build configuration. Note that they will be executed sequentially. In the end, the build gets the merged status and the output goes into the same build log. If some step fails, the rest is executed or not, depending on their [step execution policy](#).
  - Additionally, configure [build triggering options](#), [dependencies](#), [properties and variables](#) and [agent requirements](#).

### Creating New Build Configuration from URL

You can create a build configuration using a VCS URL:

1. On the Administration > Projects page, select the desired project in the list. (Alternatively, open the project using the Projects popup, and click the Edit Project Settings link on the right). The Project Settings page opens.
2. On the Project Settings page, Build Configurations section, click Create build configuration from URL.
3. In the Create Build Configuration dialog, specify a VCS repository URL and, if authentication is required, VCS credentials. Click Create.
4. TeamCity will suggest the build configuration name and will configure the rest of settings for you.
  - it will determine the type of the VCS repository and create a [VCS root](#).
  - TeamCity will attempt to auto-detect build steps: Ant, NAnt, Gradle, Maven, MSBuild, Visual Studio solution files, PowerShell, Xcode project files, Rake, and IntelliJ IDEA projects. If none found, you will have to [configure build steps manually](#).
  - Next, TeamCity UI will display suggestion icons with prompts to create [build triggers](#), [failure conditions](#) and [build features](#). Depending on the build configuration settings, it may suggest some additional configuration options.
5. After the build configuration is created, you can run a build and/or tweak the settings.

#### Creating New Build Configuration pointing to GitHub.com repository

1. Click the Create build configuration button and select Pointing to GitHub.com repository.
  - If you do not have a GitHub connection configured, you will be redirected to the Connections page. Set up the connection as [described here](#), then follow the steps below.
  - If you have a GitHub connection configured, follow the steps below.
2. On the Create Build Configuration From GitHub page, select a repository. TeamCity will verify the repository connection. If the Connection is verified, the new page opens.
3. TeamCity will display the project and build configuration name. If required, modify the names and click Proceed.
4. TeamCity will add a VCS build trigger and attempt to auto-detect build steps: Ant, NAnt, Gradle, Maven, MSBuild, Visual Studio solution files, PowerShell, Xcode project files, Rake, and IntelliJ IDEA projects.  
On the Auto-detected Build Steps page, select the step(s) to use in your build configuration. Click Use selected.  
If no steps found, you will have to [configure build steps manually](#).
5. Your project and a build configuration are configured. Click the Run button to start the build.  
 Depending on the build configuration settings, TeamCity can suggest some additional configuration options. Review the suggested settings  and configure required ones.

#### Creating New Build Configuration pointing to Bitbucket Cloud

1. Click the Create build configuration button and select Pointing to Bitbucket Cloud repository.
  - If you do not have a Bitbucket connection configured, you will be redirected to the Connections page. Set up the connection as [described here](#), then follow the steps below.
  - If you have a Bitbucket connection configured, follow the steps below.
2. On the Create Build Configuration From Bitbucket Cloud page, select a repository. TeamCity will verify the repository connection. If the Connection is verified, the new page opens.
3. TeamCity will display the project and build configuration name. If required, modify the names and click Proceed.
4. TeamCity will add a VCS build trigger and attempt to auto-detect build steps: Ant, NAnt, Gradle, Maven, MSBuild, Visual Studio solution files, PowerShell, Xcode project files, Rake, and IntelliJ IDEA projects.  
On the Auto-detected Build Steps page select the step(s) to use in your build configuration. Click Use selected.  
If no steps found, you will have to [configure build steps manually](#).
5. Your project and a build configuration are configured. Click the Run button to start the build.  
 Depending on the build configuration settings, TeamCity can suggest some additional configuration options. Review the suggested settings  and configure required ones.

#### Creating Build Configuration Template

Creating a new [build configuration template](#) is similar to creating a new configuration:

1. On the Administration > Projects page, select the desired project in the list. (Alternatively, open the project using the Projects popup, and click the Edit Project Settings link on the right). The Project Settings page opens.
2. On the Project Settings page, Build Configuration Templates section, click the Create template button and proceed with configuring [general settings](#), [VCS settings](#) and [build steps](#).

Alternatively, you can create a build configuration template from an existing build configuration:

1. Open the existing build configuration settings page, click Actions at the top right corner of the screen, and select the Extract Template option.
2. Specify the settings required and click Create.

## Creating Build Configuration From Template

To create a build configuration based on a template:

1. On the Administration > Projects page, select the desired project in the list. (Alternatively, open the project using the Projects popup, and click the Edit Project Settings link on the right). The Project Settings page opens.
2. On the Project Settings page, Build Configuration Templates section, locate the desired template and click its name or use Edit.
3. Click Actions at the top right corner of the screen, and select Create Build Configuration.
4. Specify the required settings for the new configuration.



The settings specified in the template cannot be edited in a configuration created from this template. However, some of them can be [redefined in a build configuration](#). Note that modifying the settings of the template itself will affect all configurations based on this template.

Alternatively, you can create a build configuration based on a template as follows:

1. On the Administration > Projects page, select the desired project from the list. (Alternatively, open the project using the Projects popup, and click the Edit Project Settings link on the right). The Project Settings page opens.
2. On the Project Settings page, Build Configurations section, click Create build configuration.
3. On the configuration settings page, use the Based on template drop-down and select a template for your build configuration.

## Ordering Build Configurations

You can view all build configurations of a project on the Project Overview page listed in alphabetical order by default. Administrators can [customize the default order](#).

## Configuring Settings

Configuring settings of a build configuration is described on the following pages:

- [Configuring General Settings](#)
- [NuGet](#)
- [Configuring Build Steps](#)
- [Adding Build Features](#)
- [Configuring Unit Testing and Code Coverage](#)
- [Build Failure Conditions](#)
- [Configuring Build Triggers](#)
- [Configuring Dependencies](#)
- [Configuring Build Parameters](#)
- [Configuring Agent Requirements](#)

Note that editing via the TeamCity Web UI will be disabled for build configurations created via the [REST API](#).

See also:

[Administrator's Guide: Configuring Dependencies | Configuring Build Parameters | Configuring VCS Settings](#)

## Configuring General Settings

When creating a build configuration, specify the following settings:

Setting	Description
Name	The build configuration name
Build Configuration ID	The <a href="#">ID</a> of the build configuration (must be unique across all build configurations and templates in the system).
Description	An optional description for the build configuration.

Build Configuration Type	Select one of the types: - regular build configuration, defining actions and rules to apply to the source code. All the settings above are applicable. - <a href="#">deployment build configuration</a> , which deploys artifacts of other builds to some environment - <a href="#">composite build configuration</a> , which aggregates results from several other builds combined by snapshot dependencies and presents them in a single place
Build Number Format	This value is assigned to the build number. For more information, refer to the <a href="#">Build Number Format</a> section below.
Build Counter	Specify the counter to be used in the build numbering. Each build increases the build counter by 1. Use the Reset counter link to reset counter value to 1.
Artifact Paths	Patterns to define artifacts of a build. For more information, refer to the <a href="#">Artifact Paths</a> section below.
Build Options	Additional options for this build configuration. For more information, refer to the following sections below: <ul style="list-style-type: none"> <li>● <a href="#">Hanging Build Detection</a></li> <li>● <a href="#">Allow Triggering Personal Builds</a></li> <li>● <a href="#">Enable Status Widget</a> <ul style="list-style-type: none"> <li>● <a href="#">HTML Status Widget</a></li> </ul> </li> <li>● <a href="#">Limit Number of Simultaneously Running Builds</a></li> </ul>

#### Build Number Format

In the Build number format field you can specify a pattern which is resolved and assigned to the [Build Number](#) on the build start. The following substitutions are supported in the pattern:

Pattern	Description
%build.counter%	a build counter unique for each build configuration. It is maintained by TeamCity and will resolve to a next integer value on each new build start. The current value of the counter can be edited in the <a href="#">Build counter</a> field.
%build.vcs.number.<VCS_root_name>%	the revision used for the build of the VCS root with <VCS_root_name> name. <a href="#">Read more</a> on the property.
%property.name%	a value of the build property with the corresponding name. All the <a href="#">Predefined Build Parameters</a> are supported (including <a href="#">Reference-only server properties</a> ).



A build number format example:

1.0.%build.counter%.%build.vcs.number.My\_Project\_svn%

Though not required, it is still highly recommended to ensure the build numbers are unique. Please include the build counter in the build number and do not reset the build counter to lesser values.

It is also possible to change the build number from within your build script. For details, refer to [Build Script Interaction with TeamCity](#).

#### Artifact Paths

[Build artifacts](#) are files produced by the build which are stored on TeamCity server and can be downloaded from the TeamCity web UI or used as artifact dependencies by other builds.

On the General Settings page of the build configuration, you can specify patterns for the files on the agent which will be uploaded to the server after the build.

If you have a finished build on an agent, you can use the checkout directory browser (which lists the checkout directory content on the agent) and select artifacts from the tree. TeamCity will place the paths to them into the input field.

The Artifact Paths field supports relative (to the build checkout directory) and absolute paths. Using relative paths is recommended. You can specify exact file paths or patterns, one per line or comma-separated. Patterns support the "\*" and "\*\*" wildcards (see below). Each line can be of the form `[+:]source [> target]` to include and `-:source [> target]` to exclude files or directories to publish as build artifacts. The parts enclosed in square brackets are optional. Rules are grouped by the right part and are applied in the order of appearance, e.g.

```
+:**/* => target_directory  
-:**/folder1 => target_directory
```

will tell TeamCity to publish all files except for folder1 into the target\_directory.

Line format description:

```
file_name|directory_name|wildcard [ => target_directory|target_archive ]
```

Note that although absolute paths are supported in the source part, it is recommended to use paths relative to the [build checkout directory](#).

- file\_name — to publish the file. The name should be relative to the [Build Checkout Directory](#).
- directory\_name — to publish all the files and subdirectories within the directory specified. The directory name should be a path relative to the [Build Checkout Directory](#). The files will be published preserving the directories structure under the directory specified (the directory itself will not be included).
- wildcard — to publish files matching [Ant-like wildcard](#) pattern (only "\*" and "\*\*" wildcards are supported). The wildcard should represent a path relative to the build checkout directory. The files will be published preserving the structure of the directories matched by the wildcard (directories matched by "static" text will not be created). That is, TeamCity will create directories starting from the first occurrence of the wildcard in the pattern.
- You can use [build parameters](#) in the artifacts specification. For example, use "mylib-%system.build.number%.zip" to refer to a file with the build number in the name.

The optional part starting with the `=>` symbol and followed by the target directory name can be used to publish the files into the specified target directory. If the target directory is omitted, the files are published in the root of the build artifacts. You can use `".` (dot) as a reference to the build checkout directory.

The target paths cannot be absolute. Non-relative paths will produce errors during the build.

- target\_directory — (optional) the directory in the resulting build's artifacts that will contain the files determined by the left part of the pattern.
- target\_archive — (optional) the path to the archive to be created by TeamCity by packing build artifacts determined in the left part of the pattern. TeamCity treats the right part of the pattern as target\_archive whenever it ends with a supported archive extension, i.e. `.zip`, `.7z`, `.jar`, `.tar.gz`, or `.tgz`.

Examples:

- `install.zip` — publish file named `install.zip` in the build artifacts
- `dist` — publish the content of the `dist` directory
- `target/*.jar` — publish all jar files in the target directory
- `target/**/*.*.txt=> docs` — publish all the txt files found in the target directory and its subdirectories. The files will be available in the build artifacts under the `docs` directory.
- `reports => reports, distrib/idea*.zip` — publish reports directory as `reports` and files matching `idea*.zip` from the `distrib` directory into the artifacts root.
- Relative paths inside a zip archive can be used, if needed: `results\result1\Dir1\Dir2 => archive.zip!results/result1/Dir1`
- The same target\_archive name can be used multiple times, for example:  
`+:**/*.html => report.zip`  
`+:**/*.css => report.zip!/css/`  
`-:**/*.txt => report.zip`

## Build Options

The following options are available for build configurations:

### Hanging Build Detection

Select the Enable hanging build detection option to detect probably "hanging" builds. A build is considered to be "hanging" if its run time significantly exceeds estimated average run time and the build has not send any messages since the estimation was

exceeded. To properly detect hanging builds, TeamCity has to estimate the average time builds run based on several builds. Thus, if you have a new build configuration, it may make sense to enable this feature after a couple of builds have run, so that TeamCity would have enough information to estimate the average run time.

#### Allow Triggering Personal Builds

You can restrict running [personal builds](#) by unchecking the allow triggering personal builds option (on by default).

#### Enable Status Widget

This option enables retrieving the status and basic details of the last build in the build configuration without requiring any user authentication.

Please note that this also allows getting the status of any specific build in the build configuration (however, builds cannot be listed and no other information except the build status (success/failure/internal error/cancelled) is available).

The status can be retrieved via the HTML status widget described below, or via a single icon with the help of [REST API](#).

#### HTML Status Widget

This feature allows you to get an overview of the current project status on your company's website, wiki, Confluence or any other web page.

When the Enable status widget option is enabled, an HTML snippet can be included into an external web page and will display the current build configuration status.

For build status icon as a single image, check [REST build status icon](#).

The following build process information is provided by the status widget:

- The latest build results,
- Build number,
- Build status,
- Link to the latest build artifacts.

The status widget doesn't require users log in to TeamCity.

When the feature is enabled, you need to include the following snippets of code in the web page source:

- Add this code sample in the `<head>` section (or alternatively, add the `withCss=true` parameter to `externalStatus.html`):

```
<style type="text/css">
@import "<TeamCity_server_URL>/css/status/externalStatus.css";
</style>
```

- Insert this code sample where you want to display the build configuration status:

```
<script type="text/javascript" src="<TeamCity_server_URL>/externalStatus.html?js=1">
</script>
```

- If you prefer to use plain HTML instead of javascript, omit the `js=1` parameter and use `iframe` instead of the script:

```
<iframe src="<TeamCity_server_URL>/externalStatus.html"/>
```

- If you want to include default CSS styles without modifying the `<head>` section, add the `withCss=true` parameter

To provide up-to-date status information on specific build configurations, use the following parameter in the URL as many times as needed:

```
&buildTypeId=<external build configuration ID>
```

It is also possible to show the status of all projects build configurations by replacing "`&buildTypeId=<external build configuration ID>`" with "`&projectId=<external project ID>`". You can select a combination of these parameters to display the needed projects and build configurations on your web page.

You can also download and customize the `externalStatus.css` file (for example, you can disable some columns by using `display: none;` See comments in `externalStatus.css`). But in this case, you must not include the `withCss=true` parameter, but provide the CSS styles explicitly in the `<head>` section, instead.

Enabling the status widget also allows non-logged in users to get the RSS feed for the build configuration.

#### Limit Number of Simultaneously Running Builds

Specify the number of builds of the same configuration that can run simultaneously on all agents. This option helps avoid the situation, when all of the agents are busy with the builds of a single project. Enter 0 to allow an unlimited number of builds to run simultaneously.

See also:

[Concepts: Build Configuration](#)

## NuGet

On this page:

- [Integration Capabilities](#)
- [Typical Usage Scenarios](#)
- [Installing NuGet to TeamCity agents](#)
- [NuGet Packages Cache Clean-up on Agents](#)
- [Using TeamCity as NuGet Server](#)
- [Symbol Packages](#)
- [Proxy Configuration](#)

#### Integration Capabilities

TeamCity integrates with [NuGet](#) package manager and when [NuGet](#) is installed provides the following capabilities:

- [NuGet feed based on the builds' published artifacts](#)
- A set of NuGet runners to be used in builds on Windows OS, as well as on Linux and macOS when [Mono](#) is installed on the agent. Only NuGet 3.2+ on Mono 4.4.2+ is supported.
  - the [NuGet Installer](#) build runner, which installs and updates NuGet packages
  - the [NuGet Pack](#) build runner, which builds NuGet packages
  - the [NuGet Publish](#) build runner, which publishes packages to a feed of your choice
- The [NuGet Dependency Trigger](#), which allows triggering builds on NuGet feed updates.



#### Supported Operating Systems

NuGet build runners are supported on build agents running Windows OS by default. Linux and macOS are supported when [Mono](#) is installed on the agent (NuGet command-line executable for Windows version 3.2 and later is strongly recommended with Mono).



- NuGet build runners require an appropriate version of .NET Framework installed on the agent machine depending on the NuGet.exe version used: NuGet 2.8.6+ requires .NET 4.5+, earlier NuGet versions require .NET 4.0.
- To use packages from an authenticated feed, see the [NuGet Feed Credentials](#) build feature.

#### Typical Usage Scenarios

- To install packages from a public feed, add the [NuGet Installer](#) build step.
- To create a package and publish it to a public feed, add the [NuGet Pack](#) and [NuGet Publish](#) build steps.
- To create a package and publish it to the internal TeamCity NuGet Server, enable TeamCity as a NuGet Server (see the [dedicated page](#)), use the [NuGet Pack](#) build step and [NuGet Publish](#) build steps.
- To trigger a new build when a NuGet package is updated, use the [NuGet Dependency Trigger](#).

#### Installing NuGet to TeamCity agents

The NuGet trigger and the NuGet-related build runners require the NuGet command line binary configured on the server. They are automatically distributed to agents once configured. Several versions can be installed and a version of your choice can be

set as the default one.

To install NuGet.exe on TeamCity:

1. Go to the Administration | Tools tab.
2. Click Install tool and select NuGet.exe.
3. Select whether you want to download (default) NuGet from the public feed or upload your own NuGet package containing NuGet.exe.
  - a. If the Download radio button is chosen, select the NuGet version to install on all build agents.



It is recommended to use release versions of NuGet.

- b. If the Upload radio button is selected, choose your own NuGet package.
4. Specify whether this NuGet version will be default using the related check-box.
5. Click Add to install the selected NuGet version.



Installing NuGet on agents results in agents upgrade.

#### NuGet Packages Cache Clean-up on Agents

NuGet uses several local caches to avoid downloading packages that are already installed, and to provide offline support. If an agent is running out of the space, TeamCity will try to clean NuGet packages cache on the agent.

The caches in the following directories will be cleaned:

- %NUGET\_PACKAGES% environment variable (must be an absolute path)
- %LOCALAPPDATA%\NuGet\Cache
- %LOCALAPPDATA%\NuGet\v3-cache
- %user.home%\nuget\packages

#### Using TeamCity as NuGet Server

Please see the [dedicated page](#).

## Symbol Packages

Publishing of NuGet [symbol packages](#) to an internal TeamCity feed may cause issues when using an external source server. See the [corresponding issue](#) in our public tracker.

#### Proxy Configuration

NuGet command line client supports proxy server configuration via NuGet.config file parameters or environment variables. See [NuGet documentation](#) for more details.

See also:

[Administrator's Guide: NuGet Installer](#) | [NuGet Publish](#) | [NuGet Pack](#) | [NuGet Dependency Trigger](#)

## Using TeamCity as NuGet Feed

On this page:

- [Enabling NuGet Feed](#)
- [Indexing NuGet Packages](#)
  - Indexing packages published as artifacts
  - Using NuGet push command
- [Using TeamCity NuGet Feed](#)

If you want to publish your NuGet packages to a limited audience, e.g. to use them internally, you can use TeamCity as a [NuGet feed](#).



TeamCity running on any of the supported operating systems (Windows, Linux, macOS) can be used as a NuGet feed.

The built-in TeamCity NuGet feed supports API v2 by default.

### Enabling NuGet Feed

To start using TeamCity as a NuGet Server, you need to enable the NuGet Feed at the project level.

Go to the NuGet Feed page in the project settings and enable the feed. Two different links may be displayed on the page:

- public feed URL (with guestAuth prefix): packages will be visible to all users which have access to TeamCity server. If the public feed URL is not available, you need to enable the [Guest user login](#) in TeamCity on the Administration | Authentication page.
- private feed URL (with httpAuth prefix): to access packages, it requires [Basic HTTP authentication](#) and the "View project" user permission.

### Indexing NuGet Packages

#### Indexing packages published as artifacts

By default, TeamCity will not add .nupkg artifacts published by builds into the project NuGet feed. You can select one of the following options:

- to index packages published by selected build configurations only, add the [NuGet packages indexer](#) build feature to these build configurations
- to index all .nupkg files published as build artifacts in the project, enable Automatic NuGet Packages Indexing on the NuGet Feed page of the project settings
- use the [NuGet Pack](#) build step with the Publish created packages checkbox.

Besides, .nupkg files indexing is performed by the agent itself while publishing build artifacts.



Deleting a NuGet Feed with all its contents from a project will remove all [NuGet Packages Indexer](#) features pointing to this feed.

### Using NuGet push command

To publish .nupkg file into TeamCity NuGet feed during the build, you can specify the NuGet feed URL as a package source and the `%teamcity.nuget.feed.api.key%` value as a feed key in the following build steps:

- [NuGet Publish](#)
- [.NET CLI](#) with "nuget push" command

### Using TeamCity NuGet Feed

You can add TeamCity NuGet feeds as package sources package on your developer machine, e.g. to use packages during development: use the [nuget sources](#) command or NuGet package management in your IDE.

You can use TeamCity NuGet feeds to restore packages in your builds: when the [NuGet Installer](#) build step is used, TeamCity will use the [credentials provider](#) to automatically authenticate requests to the private TeamCity NuGet feeds.



The packages available in the feed are bound to the builds' artifacts: they are removed from the feed when the artifacts of the build which produced them are [cleaned up](#). To avoid artifacts cleanup of a specific build, you can [pin this build](#).

 Internet Explorer settings may need to be set to trust the TeamCity Server when working in a mixed authentication environment.

## Configuring Build Steps

When creating a build configuration, it is important to configure the sequence of build steps to be executed.

Build steps are configured on the [Build Steps](#) section of the [Build Configuration Settings](#) page: the steps can be auto-detected by TeamCity or added manually.

Each build step is represented by a [build runner](#) and provides integration with a specific build or test tool. You can add as many build steps to your build configuration as needed. For example, call a NAnt script before compiling VS solutions.

Build steps are invoked sequentially.

The decision whether to run the next build step may depend on the exit status of the previous build steps and the current build status.

The build step status is considered failed if the build process returned a non-zero exit code and the Fail build if build process exit code is not zero build failure condition is enabled (see [Build Failure Conditions](#)); otherwise build step is successful.

Note that the status of the build step and the build can be different. A build step can be successful, but the build can be failed because of another build failure condition, not based on the exit code (like failing a test or something else). On the other hand, if a build step has failed, the build will be failed too.

### Execution policy

You can specify the step execution policy via the Execute step option:

- Only if build status is successful - before starting the step, the build agent requests the build status from the server, and skips the step if the status is failed. This considers the failure conditions processed by the server, like failure on test failures or on metric change. Note that this still can be not exact as some failure conditions are processed on the server asynchronously ([TW-17015](#))
- If all previous steps finished successfully - the build analyzes only the build step status on the build agent, and doesn't send a request to the server to check the build status and considers only important step failures.
- Even if some of previous steps failed: select to make TeamCity execute this step regardless of the status of previous steps and status of the build.
- Always, even if build stop command was issued: select to ensure this step is always executed, even if the build was canceled by a user. For example, if you have two steps with this option configured, stopping the build during the first step execution will interrupt this step, while the second step will still run. Issuing the stop command for the second time will result in ignoring the execution policy: the build will be terminated.



- You can copy a build step from one build configuration to another from the original build configuration settings page.
- You can reorder build steps as needed. Note, that if you have a build configuration inherited from a template, you cannot reorder inherited build steps. However, you can insert custom build steps (not inherited) at any place and in any order, even before or between inherited build steps. Inherited build steps can be reordered in the original template only.
- You can disable a build step temporarily or permanently, even if it is inherited from a build configuration template using the corresponding option in the last column of the Build Steps list.

### Bundled runners

For the details on configuring individual build steps, refer to:

- [.NET CLI \(dotnet\)](#)
- [.NET Process Runner](#)
- [Ant](#)
- [Command Line](#)
- [Deployers](#)
  - Container Deployer
  - FTP Upload
  - SMB Upload
  - SSH Exec
  - SSH Upload
- [Docker](#)
- [Docker Compose](#)
- [Duplicates Finder \(Java\)](#)
- [Duplicates Finder \(ReSharper\)](#)
- [FxCop](#)

- Gradle
- Inspections
- Inspections (ReSharper)
- IntelliJ IDEA Project
- Ipr (deprecated)
- Maven
- MSBuild
- MSpec
- MSTest
- NAnt
- NuGet Installer
- NuGet Pack
- NuGet Publish
- NUnit
- PowerShell
- Rake
- Simple Build Tool (Scala)
- Visual Studio (sln)
- Visual Studio 2003
- Visual Studio Tests
- Working with Meta-Runner
- Xcode Project

See also:

[Concepts: Build Runner](#)

Working with Meta-Runner

A Meta-Runner allows you to extract build steps, requirements and parameters from a build configuration and create a [build runner](#) out of them. This build runner can then be used as any other build runner in a build step of any other build configuration or template.

Basically, a meta-runner is a set of build steps from one build configuration that you can reuse in another; it is an xml definition containing build steps, requirements and parameters that you can utilize in xml definitions of other build configurations. TeamCity allows extracting meta-runners using the web UI.

With a meta-runner, you can easily reuse existing runners, create new runners for typical tasks (e.g. publish to FTP, delete directory, etc.), you can simplify your build configuration and decrease a number of build steps.

All meta-runners are stored on a project level, so they are available within this project and its subprojects only, and are not visible outside. If a meta-runner is stored on the <Root project> level, it is available globally (in all projects).

You can use the existing meta-runners from the TeamCity Meta-Runners Power Pack or create your own meta-runner.

On this page:

- Using Meta-Runners Power Pack
  - Installing Meta-Runner
- Creating Meta-Runner
  - Creating Your Own Meta-Runner from UI
    - Preparing Build Configuration
    - Verifying Build Configuration Works Properly
    - Extracting and Using Meta-Runner
  - Creating Meta-Runner from XML Definition of Build Configuration
  - Creating Build Configuration from Meta-Runner

## Using Meta-Runners Power Pack

[Meta-runners Power Pack for TeamCity](#) available on GitHub is a collection of meta-runners for various tasks like downloading a file, triggering a build, tagging a build, changing a build status, running PHP tasks, etc.

Each file called \*MRPP\_\<some text\>.xml\* contains a definition of a single Meta-runner. Download the required meta-runner (or copy its definition to a file) and install it as described in the section below.

### Installing Meta-Runner

You can install a meta-runner using the TeamCity Web UI. Alternatively, you can do it directly via the file system.

- to install a Meta-runner via the Web UI, go to the Project Settings page, select Meta-Runners from the list of settings on

the left, click Upload Meta-Runner and select the Meta-runner definition file. Save you changes.

- to install a Meta-runner directly to the file system, take the Meta-runner definition file and put it into the `TeamCity Data Directory\config\projects\<project ID>\pluginData\metaRunners` directory, where `\<Project ID\>` is the identifier of a project where you want to place the Meta-runner. If the `metaRunners` directory does not exist, it should be created. Once you place the file on the disk, TeamCity will detect it and load this Meta-runner; no server restart is required.

If the Meta-runner is loaded successfully, you will see it listed on the Meta-Runners page for the project; if you have appropriate permissions, you can modify the definition directly in the TeamCity UI.

The runner is now available in the list of build runners on the build configuration Build Steps page and is represented as a native TeamCity runner with a convenient UI.

A Meta-runner placed into a project will be available to all its subprojects and build configurations. To make a Meta-runner available to all projects, place it in the Root project.

## Creating Meta-Runner

You can create a build configuration via the TeamCity Web UI and extract a meta-runner from it or use the xml definition of an existing build configuration as a meta-runner.

### Creating Your Own Meta-Runner from UI

Let us consider an example of creating a meta-runner.

To create a meta-runner, follow these steps (described below in more detail):

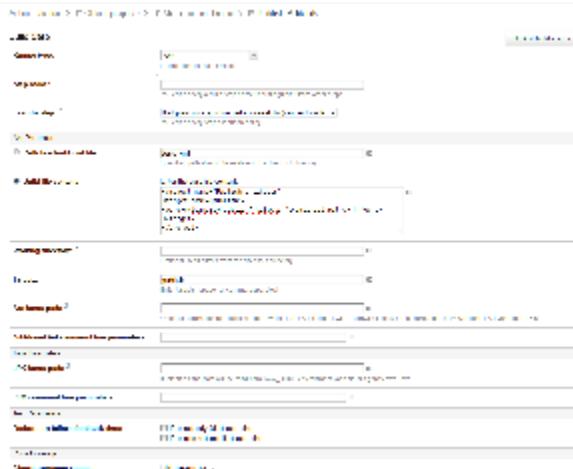
1. prepare a [build configuration to test the build steps we are going to use in our meta-runner](#),
2. [make sure the build configuration is working](#),
3. [extract a meta-runner to the desired project](#).

In this example, we will create a meta-runner to publish some artifacts to TeamCity with the help of corresponding [service message](#).

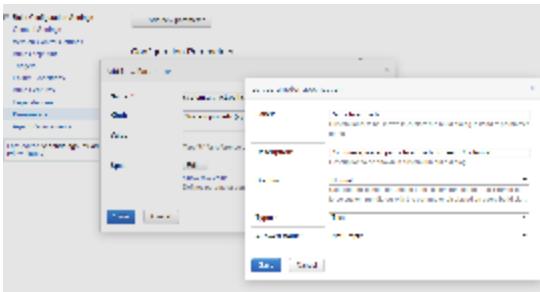
Usually artifacts configured in a build configuration are published when the build finishes. However, sometimes for long builds with multiple build steps we need artifacts faster. In this example, we will create a runner which can be inserted between any build steps and can be configured to publish artifacts produced by previous steps.

### Preparing Build Configuration

The first step is to prepare a build configuration which will work the same way as the meta-runner we would like to produce. Let us use the configuration with a single Ant build step: Ant can be executed on any platform where the TeamCity agent runs; besides, Ant runner in TeamCity supports build.xml specified right in the runner settings. This is important because our build configuration must be self-contained, it cannot take build.xml from the version control repository. So in our case the Ant step settings will look like this:



where `artifact.paths` is a system property. We need to add it on the Parameters tab of the build configuration settings:

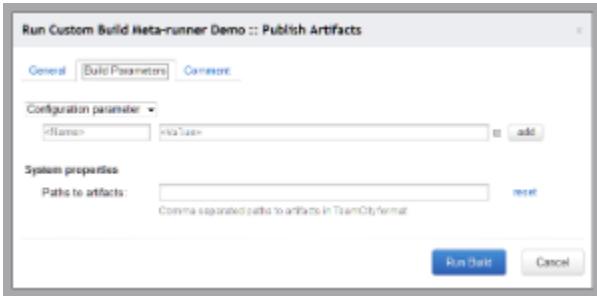


Note that each parameter can have a specification where we can provide the label, description, type of control and specify validation conditions. Before version 8.0 this specification was used by the custom build dialog only. Now this specification is used by a meta-runner too.

**i** Here the Ant build step is used just as an example. In the initial build configuration, you can use any of the available build runners (e.g. MSBuild, .Net process, etc.) and configure the settings, define the parameters for this build step. When you extract a meta-runner from this build configuration, all the settings defined in the build step, and all the build parameters will be added to the meta-runner.

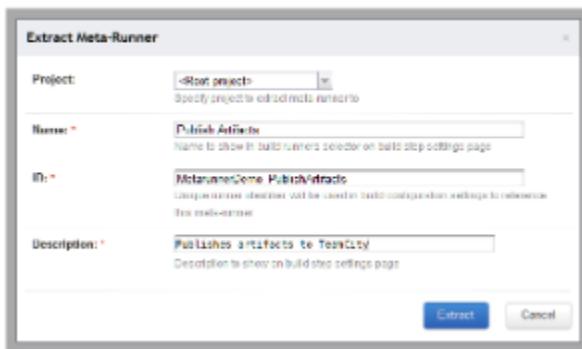
### Verifying Build Configuration Works Properly

Once the build steps and parameters are defined, we need to make sure our build configuration works by running a couple of builds through the custom build dialog:



### Extracting and Using Meta-Runner

If the build configuration works properly, we can create a meta-runner using the Actions button in the top right-hand corner, Extract meta-runner... option:



The Extract Meta-Runner dialog requires specifying the project where the meta-runner will be created. A meta-runner created in a project will be available in this project and all its subprojects. In our case the <Root project> is selected, so the meta-runner will be available in all projects.

We also need to provide the name, description and an ID for the meta-runner: the name and description will be shown in the web interface, an ID is required to distinguish this meta-runner from others.

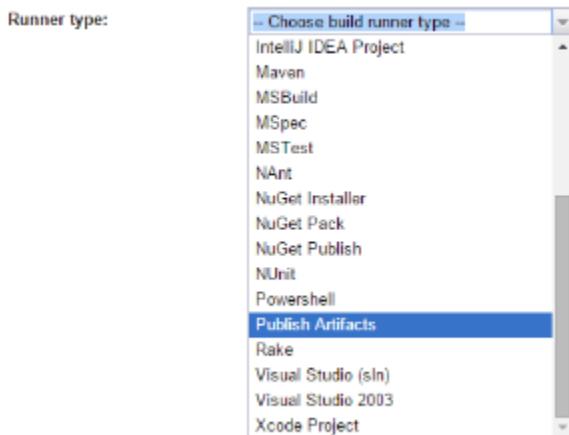
Upon clicking the Extract button, TeamCity will take definitions of all build steps and parameters in this build configuration and create a build runner out of them.

**i** Besides build steps and parameters, a meta-runner can also have requirements: if requirements are defined in the

build configuration, they will be extracted to the meta-runner automatically. Requirements can be useful if the tools used by meta-runner are available on specific platforms only.

Once the meta-runner is extracted, it becomes available in the build runners selector and can be used in any build step just like any other build runner:

### New Build Step



The current meta-runner usages can be seen at the project Meta-Runners page:

When a meta-runner is extracted, all steps will be extracted. If you need to reorder parameters or make some quick fixes in the runner script, you can edit its raw XML definition in the web browser: go to the Administration page of the project -> Meta-Runners and use the Edit option next to the meta-runner. The parameters will be shown in the same order as the `<param>` elements in the XML definition. Definitions of meta-runners are stored in the `TeamCity Data Directory \config\projects\<project ID>\pluginData\metaRunners` folder.

#### Creating Meta-Runner from XML Definition of Build Configuration

Alternatively, you can use the XML definition of an existing build configuration as a meta-runner. To do it, save the definition of this build configuration to a file named as follows: `<runner id>.xml`, where `<runner id>` is the ID of this build runner. Install the meta-runner as described above.

Since a meta-runner looks and works like any other runner, it is also possible to create another meta-runner on the basis of an existing meta-runner.

#### Creating Build Configuration from Meta-Runner

If you need to fix a meta-runner and test your fix, you can create a build configuration from a meta-runner, can change its steps, adjust parameters and requirements, check how it works, and then use the Extract meta-runner action to apply the changes to the existing meta-runner with the same ID.

#### .NET Process Runner

The .NET Process Runner is able to run any .NET assembly under the selected .NET Framework version and platform, optionally with .NET code coverage. You can use it to run xUnit, Gallio or other .NET tests, for which there is no dedicated build runner.



The runner requires .NET Framework installed on the TeamCity Agent.

#### .NET Process Runner Settings

Option	Description
Path	Specify the path to a .NET executable (for example, to the xUnit console)

Command line parameters	Provide newline- or space-separated command line parameters to be passed to the executable.
Working directory	Specify the <a href="#">build working directory</a> if it differs from the <a href="#">build checkout directory</a> .
.NET Runtime	From the Platform drop-down select the desired execution mode on a x64 machine. The supported values are: Auto (MSIL) (default), x86 and x64. From the Version drop-down select the desired .NET Framework version.  <div style="border: 1px solid #ccc; padding: 5px; margin-left: 20px;"> If you have an MSIL .NET 2.0/3.5 executable, TeamCity can enforce it to execute under any required runtime: x86 or x64, and .NET2.0 or .NET 4.0 runtime.</div>

## Code Coverage

If needed, [add code coverage](#).

Note that you do not need to write any additional build scripts.

## See also:

[Administrator's Guide: Configuring .NET Code Coverage](#)

## Ant

This is a runner for Ant build.xml files. TeamCity 2018.1 comes bundled with Ant 1.9.11.

### In this section:

- [Testing Frameworks Support](#)
- [Reporting and Logging](#)
- [Ant Runner Settings](#)
  - [Ant Parameters](#)
    - [ant-net-tasks Tool](#)
  - [Java Parameters](#)
  - [Test Parameters](#)
  - [Docker Settings](#)
  - [Code Coverage](#)

## Testing Frameworks Support

The TeamCity Ant runner supports the JUnit and TestNG frameworks. When tests are run by the `junit` and `testng` tasks directly within the script, TeamCity reports tests on the fly.

By using the `<parallel>` tag in your Ant script, it is possible to have the JUnit and TestNG tasks run in parallel. TeamCity supports this and should concurrently log the parallel processes correctly.

## Reporting and Logging

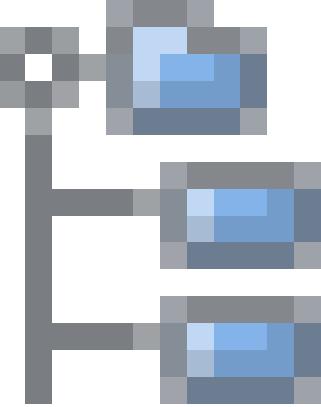
TeamCity collects detailed data from Ant as to the performed activities, provides structured error reporting, and reports tests. However, you can start a build with no specific reporting or to turn off TeamCity-specific logging:

- To disable TeamCity-specific reporting in Ant, use `teamcity.ant.listener.enabled=false` [build configuration parameter](#)
- To disable JUnit reporting, use `teamcity.ant.junit-support.enabled=false` [system property](#)
- To disable TestNG reporting, use `teamcity.ant.testng-support.enabled=false` [system property](#)

## Ant Runner Settings

### Ant Parameters

Option	Description
--------	-------------

Path to build.xml file	If you choose the option, you can type the path to an Ant build script file of the project. The path is relative to the project root directory. Alternatively, click  to choose the file using the VCS repository browser.
Build file content	If you choose this option, click the Type build file content link and type the source code of your custom build file in the text area. Note that the text area is resizable. Use the Hide link to close the text area.
Working directory	Specify the <a href="#">build working directory</a> if it differs from the checkout directory.
Targets	Use this text field to specify valid Ant targets as a list of space-separated strings. The available targets can be viewed in the Web UI by clicking the icon next to the field and added by checking the appropriate boxes. If this field is left empty, the default target specified in the build script file will be run.
Ant home path	Specify the path to the distributive of your custom Ant. You do not need to specify this parameter if you are going to use the Ant 1.9.7 distributive bundled with TeamCity.   Please note, that you should use Ant 1.7 if you want to run JUnit4 tests in your builds.
Additional Ant command line parameters	Optionally, specify additional command line parameters as a space-separated list. For example, you can specify the <a href="#">ant-net-tasks Tool</a> here (see below).

#### ant-net-tasks Tool

The Ant build runner comes with a bundled tool, ant-net-tasks, which includes the jar files required for network tasks, such as FTP, sshexec, scp and mail.

It also contains [missing link Ant task](#) which can be used for REST requests.

To use the tool, specify `-lib "%teamcity.tool.ant-net-tasks%"` in [Additional Ant command line parameters](#) of the runner settings.

#### Java Parameters

Option	Description
JDK	Select a JDK. This section details the available options. The default is JAVA_HOME environment variable or the agent's own Java.

JDK home path	The option is available when <Custom> is selected above. Use this field to specify the path to your custom JDK used to run the build. If the field is left blank, the path to JDK Home is read either from the JAVA_HOME environment variable on agent the computer, or from the env.JAVA_HOME property specified in the <a href="#">build agent configuration</a> file (buildAgent.properties). If these values are not specified, TeamCity uses the Java home of the build agent process itself.
JVM command line parameters	You can specify such JVM command line parameters, e.g. maximum heap size or parameters enabling remote debugging. These values are passed by the JVM used to run your build. Example:  -Xmx512m -Xms256m

### Test Parameters

Tests reordering works the following way: TeamCity provides tests that should be run first (test classes), after that, when a JUnit task starts, it checks whether it includes these tests. If at least one test is included, TeamCity generates a new fileset containing included tests only and processes it before all other filesets. It also patches other filesets to exclude tests added to the automatically generated fileset. After that JUnit starts and runs as usual.

Option	Description
Reduce test failure feedback time:	Use the following two options to instruct TeamCity to run some tests before others. <ul style="list-style-type: none"> <li>Run recently failed tests first - If checked, TeamCity will first run tests failed in the previous finished or running builds as well as tests having a high failure rate (so called blinking tests).</li> <li>Run new and modified tests first - If checked, TeamCity will first run the tests added or modified in the change lists included in the running build.</li> </ul>

 If both options are enabled at the same time, the tests of the new and modified tests group will have higher priority, and will be executed first.

### Docker Settings

In this section, you can specify a Docker image which will be used to run the build step.

### Code Coverage

To learn about configuring code coverage options, refer to the [Configuring Java Code Coverage](#) page.

### See also:

[Administrator's Guide: Configuring Java Code Coverage](#)

### Command Line

Using this build runner, you can run any script supported by the OS.  
In this section:

- [Command Line Runner Settings](#)
  - [General Settings](#)
  - [Docker Settings](#)
  - [Code Coverage](#)

### Command Line Runner Settings

#### General Settings

Option	Description
--------	-------------

Working directory	Specify the <a href="#">Build Working Directory</a> if it differs from the <a href="#">build checkout directory</a> .
Run	Select whether you want to run an executable with parameters or custom shell/batch scripts.
Command executable	The option is available if "Executable with parameters" is selected in the Run dropdown. Specify the executable file of the build runner
Command parameters	The option is available if "Executable with parameters" is selected in the Run dropdown. Specify parameters as a space-separated list.
Format stderr output as: (Since TeamCity 2017.2)	Specify how the error output is handled by the runner: <ul style="list-style-type: none"> <li>error: any output to <code>stderr</code> is handled as an error</li> <li>warning: default; any output to <code>stderr</code> is handled as a warning</li> </ul>
Custom script	A platform-specific script which will be executed as a *.cmd file on Windows or as an executable script in Unix-like environments. The option is available if "Custom script" is selected in the Run dropdown. <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p> TeamCity treats a string surrounded by percentage signs (%) in the script as a <a href="#">parameter reference</a>. To prevent TeamCity from treating the text in the percentage signs as a property reference, use double percentage signs to escape them: e.g. if you want to pass "%Y%m%d%H%M%S" into the build, change it to "%%Y%%m%%d%%H%%M%%S"</p> </div>

## Docker Settings

In this section, you can specify a Docker image which will be used to run this build step.

## Code Coverage

To learn about configuring code coverage options, refer to the [Configuring Java Code Coverage](#) page.

## See also:

[Concepts: Build Runner | Build Checkout Directory | Build Working Directory](#)  
[Administrator's Guide: Configuring Build Steps](#)

## Duplicates Finder (ReSharper)

The Duplicates finder (ReSharper) Build Runner based on [ReSharper Command Line Tools](#) is intended to catch similar code fragments and provide a report on the discovered repetitive blocks of C# and Visual Basic .NET code in Visual Studio 2003, 2005, 2008, 2010, 2012, 2013, and 2015 solutions.

 This runner requires .NET Framework 4.0 (or higher) to be installed on the agent where builds will run.

## Sources

Option	Description
Include	Use newline-delimited Ant-like wildcards relative to the checkout root to specify the files to be included into the duplicates search. Visual Studio solution files are parsed and replaced by all source files from all projects within a solution. Example: <code>src\MySolution.sln</code>
Exclude	Enter newline-delimited Ant-like wildcards to exclude files from the duplicates search (for example, <code>*/generated{*}{}.cs</code> ). The entries should be relative to the checkout root.

[JetBrains ReSharper Command Line Tools Settings](#)

Option	Description
R# CLT Home Directory	Select the ReSharper Command Line Tools version. You can check the installed JetBrains ReSharper Command Line Tools versions on the <a href="#">Administration   Tools</a> page. If you want to run ReSharper duplicates using a specific ReSharper version (e.g. to ensure it matches the version you have installed in Visual Studio), you can use this page to install another version of the tools and can change the default version to be used.

#### Duplicate Searcher Settings

Option	Description
Code fragments comparison	Use these options to define which elements of the source code should be discarded when searching for repetitive code fragments. Code fragments can be considered duplicated, if they are structurally similar, but contain different variables, fields, methods, types or literals. Refer to the samples below:
Discard namespaces	If this option is checked, similar contents with different namespace specifications will be recognized as duplicates.  <pre>NLog.Logger.GetInstance().Log("abcd"); A.Log.Logger.GetInstance().Log("abcd");</pre>
Discard literals	If this option is checked, similar lines of code with different literals will be recognized as duplicates.  <pre>myStatusBar.SetText("Not Logged In"); myStatusBar.SetText("Logging In...");</pre>
Discard local variables	If this option is checked, similar code fragments with different local variable names will be recognized as duplicates.  <pre>int a = 5; a += 6; int b = 5; b += 6;</pre>
Discard class fields name	If this option is checked, the similar code fragments with different field names will be recognized as duplicates.  <pre>Console.WriteLine(myFoo); Console.WriteLine(myBar); ... where myFoo and myBar are declared in the containing class</pre>

Discard types	If this option is checked, similar content with different type names will be recognized as duplicates. These include all possible type references (as shown below): <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>Logger.GetInstance("text"); OtherUtility.GetInstance("text"); ... where Logger and OtherUtility are both type names (thus GetInstance is a static method in both classes)  Logger a = (Logger) GetObject("object"); OtherUtility a = (OtherUtility) GetObject("object");  public int SomeMethod(string param); public void SomeMethod(object[] param);</pre> </div>
Ignore duplicates with complexity lower than	Use this field to specify the lowest level of complexity of code blocks to be taken into consideration when detecting duplicates.
Skip files by opening comment	Enter newline-delimited keywords to exclude files that contain the keyword in the file's opening comments from the duplicates search.
Skip regions by message substring	Enter newline-delimited keywords that exclude regions that contain the keyword in the message substring from the duplicates search. Entering "generated code", for example, will skip regions containing "Windows Form Designer generated code".
Enable debug output	Check this option to include debug messages in the build log and publish the file with additional logs (dotnet -tools-dupfinder.log) as an artifact.

#### Build Failure Conditions

If a build has too many duplicates, you can configure it to fail by setting a [build failure condition](#).

#### Duplicates Finder (Java)

The Duplicates Finder (Java) Build Runner is intended for catching similar code fragments and providing a report on discovered repetitive blocks of Java code. This runner is based on IntelliJ IDEA capabilities, so an IntelliJ IDEA project file (.ipr) or directory (.idea) is required to configure the runner. Since TeamCity 2017.1, in addition to the bundled version, it is possible to install another version of JetBrains IntelliJ Inspections and Duplicates Engine and/or change the defaults using the [Administration | Tools](#) page.

The Duplicates Finder (Java) can also find Java duplicates in projects built by Maven2 or above.



In order to run inspections for your project you should have either an IntelliJ IDEA project file (.ipr)/project directory (.idea), or Maven2 or above pom.xml of your project checked into your version control.

This page contains reference information about the following Duplicates Finder (Java) Build Runner fields:

- [IntelliJ IDEA Project Settings](#)
- [Unresolved Project Modules and Path Variables](#)
- [Project JDKs](#)
- [Java Parameters](#)
- [Duplicate Finder Settings](#)

#### IntelliJ IDEA Project Settings

Option	Description
Project file type	To be able to run IntelliJ IDEA inspections on your code, TeamCity requires either an IntelliJ IDEA project file\directory, Maven pom.xml or Gradle build.gradle to be specified here.

Path to the project	<p>Depending on the type of project selected in the Project file type, specify here:</p> <ul style="list-style-type: none"> <li>For IntelliJ IDEA project: the path to the project file (.ipr) or the path to the project directory the root directory of the project containing the .idea folder).</li> <li>For Maven project: the path to the pom.xml file.</li> <li>For Gradle project: the path to the .gradle file.</li> </ul> <p>This information is required by this build runner to understand the structure of the project.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <span style="color: #0072bc; font-size: 1.2em; border-radius: 50%; width: 1em; height: 1em; display: inline-block; vertical-align: middle;"></span> The specified path should be relative to the checkout directory.         </div>
Detect global libraries and module-based JDK in the *.iml files	<p>This option is available if you use an IntelliJ IDEA project. In IntelliJ IDEA, the module settings are stored in *.iml files, thus, if this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDks when saved. This helps ensure that all references will be properly resolved.</p> <div style="border: 2px solid #f08080; padding: 5px; margin-top: 10px;"> <span style="color: #f08080; font-size: 1.2em; border-radius: 50%; width: 1em; height: 1em; display: inline-block; vertical-align: middle;"></span> Warning            When this option is selected, the process of opening and saving the build runner settings may become time-consuming, because it involves loading and parsing all project module files.         </div>
Check/Reparse Project	<p>This option is available if you use an IntelliJ IDEA project. Click this button to reparse your IntelliJ IDEA project and import the build settings right from the project, for example the list of JDks.</p> <div style="border: 1px solid #f0e68c; padding: 5px; margin-top: 10px;"> <span style="color: #f0e68c; font-size: 1.2em; border-radius: 50%; width: 1em; height: 1em; display: inline-block; vertical-align: middle;"></span> If you update your project settings in IntelliJ IDEA (e.g add new jdks, libraries), remember to update the build runner settings by clicking Check/Reparse Project.         </div>
Working directory	<p>Enter a path to a <a href="#">Build Working Directory</a> if it differs from the <a href="#">Build Checkout Directory</a>.Optional, specify if differs from the checkout directory.</p>

#### Unresolved Project Modules and Path Variables

This section is displayed, when an IntelliJ IDEA module file (.iml) referenced from an IPR-file:

- cannot be found
- allows you to enter the values of path variables used in the IPR-file.

To refresh values in this section click Check/Reparse Project.

Option	Description
<path_variable_name>	<p>This field appears if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In Set value to field, specify a path to the project resources to be used on different build agents.</p>

#### Project JDks

This section provides the list of JDks detected in the project.

Option	Description
JDK Home	<p>Use this field to specify the JDK home for the project.</p> <div style="border: 1px solid #f0e68c; padding: 5px; margin-top: 10px;"> <span style="color: #f0e68c; font-size: 1.2em; border-radius: 50%; width: 1em; height: 1em; display: inline-block; vertical-align: middle;"></span> When building with the Ipr runner, this JDK will be used to compile the sources of corresponding IDEA modules. For Inspections and Duplicate Finder builds, this JDK will be used internally to resolve the Java API used in your project.            To run the build process, the JDK specified in the JAVA_HOME environment variable will be used.         </div>

JDK Jar File Patterns	<p>Click this link to open a text area where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns.</p> <p>The default value is used for Linux and Windows operating systems:</p> <pre>jre/lib/*.jar</pre>
	<p>For Mac OS X, use the following lines:</p> <pre>lib/*.jar</pre> <pre>../Classes/*.jar</pre>
IDEA Home	If your project uses the IDEA JDK, specify the location of the IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

 You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a [requirement](#) for the corresponding property.

## Java Parameters

Option	Description
JDK	Select a JDK. <a href="#">This section</a> details the available options. The default is JAVA_HOME environment variable or the agent's own Java.
JDK home path	The option is available when <Custom> is selected above. Use this field to specify the path to your custom JDK used to run the build. If the field is left blank, the path to JDK Home is read either from the JAVA_HOME environment variable on agent the computer, or from the env.JAVA_HOME property specified in the <a href="#">build agent configuration</a> file (buildAgent.properties). If these values are not specified, TeamCity uses the Java home of the build agent process itself.
JVM command line parameters	You can specify such JVM command line parameters, e.g. maximum heap size or parameters enabling remote debugging. These values are passed by the JVM used to run your build. Example: <pre>-Xmx512m -Xms256m</pre>

## Duplicate Finder Settings

Option	Description
Test sources	If this option is checked, the test sources will be included in the duplicates analysis.   Tests may contain the data which is duplicated intentionally, and verifying tests for duplicates may yield a lot of results creating long builds and "spamming" your reports. We recommend you not select this option.

Include / exclude patterns| Optional, specify to restrict the sources scope to run duplicates analysis on. For details, refer to the section below| #IdeaPatterns]]

Detailization level	Use these options to define which elements of the source code should be distinguished when searching for repetitive code fragments. Code fragments can be considered duplicated if they are structurally similar, but contain different variables, fields, methods, types or literals. Refer to the samples below:
Distinguish variables	If this option is checked, the similar contents with different variable names will be recognized as different. If this option is not checked, such contents will be recognized as duplicated: <pre>public static void main(String[] args) {     int i = 0;     int j = 0;     if (i == j) {         System.out.println("sum of " + i + " and " + j + " = " + i + j);     }      long k = 0;     long n = 0;     if (k == n) {         System.out.println("sum of " + k + " and " + n + " = " + k + n);     } }</pre>
Distinguish fields	If this option is checked, the similar contents with different field names will be recognized as different. If this option is not checked, such contents will be recognized as duplicated: <pre>myTable.addSelectionListener(new SelectionListener() {     public void widgetDefaultSelected(SelectionEvent e) {     }     /*.....*/ });  myTree.addSelectionListener(new SelectionListener() {     public void widgetDefaultSelected(SelectionEvent e) {     }     /*.....*/ });  });</pre>

## Distinguish methods

If this option is checked, the methods of similar structure will be recognized as different. If this option is not checked, such methods will be recognized as duplicated. In this case, they can be extracted and reused.

Initial version:

```
public void buildCanceled(Build build, SessionData data) {  
    /* ... */  
    for (IListener listener : getListeners()) {  
        listener.buildCanceled(build, data);  
    }  
}  
  
public void buildFinished(Build build, SessionData data) {  
    /* ... */  
    for (IListener listener : getListeners()) {  
        listener.buildFinished(build, data);  
    }  
}
```

After analysing code for duplicates without distinguishing methods, the duplicated fragments can be extracted:

```
public void buildCanceled(final Build build, final SessionData data) {  
    enumerateListeners(new Processor() {  
        public void process(final IListener listener) {  
            listener.buildCanceled(build, data);  
        }  
    });  
}  
  
public void buildFinished(final Build build, final SessionData data) {  
    enumerateListeners(new Processor() {  
        public void process(final IListener listener) {  
            listener.buildFinished(build, data);  
        }  
    });  
}  
  
private void enumerateListeners(Processor processor) {/* ... */  
  
for (IListener listener : getListeners()) {  
    processor.process(listener);  
}  
}  
  
private interface Processor {  
    void process(IListener listener);  
}
```

Distinguish types	If this option is checked, the similar code fragments with different type names will be recognized as different. If this option is not checked, such code fragments will be recognized as duplicates.  <pre>new MyIDE().updateStatus() new TheirIDE().updateStatus()</pre>
Distinguish literals	If this option is checked, similar line of code with different literals will be considered different. If this option is not checked, such lines will be recognized as duplicates.  <pre>myWatchedLabel.setToolTipText("Not Logged In");</pre> <pre>myWatchedLabel.setToolTipText("Logging In...");</pre>
Ignore duplicates with complexity lower than	Complexity of the source code is defined by the amount of statements, expressions, declarations and method calls. Complexity of each of them is defined by its cost. Summarized costs of all these elements of the source code fragment yields the total complexity. Use this field to specify the lowest level of complexity of the source code to be taken into consideration when detecting duplicates. For meaningful results start with value 10.
Ignore duplicate subexpressions with complexity lower than	Use this field to specify the lowest level of complexity of subexpressions to be taken into consideration when detecting duplicates.
Check if Subexpression Can be Extracted	If this option is checked, the duplicated subexpressions can be extracted.

 Include / exclude patterns are newline-delimited set of rules of the form:

```
[+|-:]pattern
```

Where the pattern must satisfy these rules:

- must end with either `**` or `*` (this effectively limits the patterns to only the directories level, they do not support file-level patterns)
- references to modules can be included as `[module_name]/<path_within_module>`

Some notes on patterns processing:

- excludes have precedence over includes
- if include patterns are specified, only directories matching these patterns will be included, all other directories will be excluded
- "include" pattern has a special behavior (due to underlying limitations): it includes the directory specified and all the files residing directly in the directories above the one specified.

Example:

```
+: testData/tables/**  
-: testData/**  
-: testdata/**  
-:[testData]/**
```

 For the file paths to be reported correctly, "References to resources outside project/module file directory" option for the project and all modules should be set to "Relative" in IDEA project.

## FxCop

The [FxCop Build Runner](#) is intended for inspecting .NET assemblies and reporting possible design, localization, performance, and security improvements.

If you want TeamCity to display FxCop reports, you can either configure the corresponding build runner, or import XML reports by means of service messages if you prefer to run the FxCop tool directly from the script.

 The FxCop build runner requires FxCop installed on the build agent.

On this page:

- The description of the [FxCop build runner settings](#)
- Details on using [dedicated service messages](#)

For the list of supported FxCop versions, see [Supported Platforms and Environments](#).

### FxCop Build Runner Settings

#### FxCop Installation

Option	Description
FxCop detection mode	When a build agent is started, it detects automatically whether FxCop is installed. If FxCop is detected, TeamCity defines the <code>%system.FxCopRoot%</code> agent system property. You can also use a custom installation of FxCop or the use FxCop checked in your version control. Depending on the selection, the settings displayed below will vary.
Autodetect installation	Select to use the FxCop installation on an agent.
FxCop version	The option is available when autodetect installation is selected. Select one of the options from the dropdown. If you have several versions of FxCop installed on your build agents, it is recommended to select here a specific version of FxCop you want to use to run inspections in your build to avoid inconsistency. As a result, an agent requirement will be created. If you leave the default value of the field ('Any Detected'), TeamCity will use any available agent with FxCop installed. In this case the version of FxCop used in one build may not be the same as the one used in the previous build, thus the number of new problems found will be different from the actual state.
Specify installation root	Select to use a custom installation of FxCop (not the autodetected one), or if you do not have FxCop installed on the build agent (e.g. you can place the FxCop tool in your source control, and check it out with the build sources)
Installation root	The option is available when Specify installation root is selected. Enter the path to the FxCop installation root on the agent machine or the path to an FxCop executable relative to the <a href="#">Build Checkout Directory</a> .



If you want to have the line numbers information and Open in IDE features, run an FxCop build on the same machine as your compilation build because FxCop requires the source code to be present to display links to it.

#### What to inspect

Option	Description
Assemblies	Enter the paths to the assemblies to be inspected (use ant-like wildcards to select files by a mask). FxCop will use default settings to inspect them. The paths should be relative to the <a href="#">Build Checkout Directory</a> and separated by spaces. Enter exclude wildcards to refine the included assemblies list.  Note that there is a limitation to the maximum number of assemblies that can be specified here due to <a href="#">command-line string limitation</a> .

FxCop project file	Enter the path relative to the <a href="#">Build Checkout Directory</a> to an FxCop project.
--------------------	--

## FxCop Options

Search referenced assemblies in GAC	Search the assemblies referenced by targets in Global Assembly Cache.
Search referenced assemblies in directories	Search the assemblies referenced by targets in the specified directories separated by spaces.
Ignore generated code	A new option introduced in FxCop 1.36. Speeds up inspection.
Report XSLT file	The path to the XSLT transformation file relative to the <a href="#">Build Checkout Directory</a> or absolute on the agent machine. You can use the path to the detected FxCop on the target machine (i.e. "%system.FxCopRoot%/Xml/FxCopReport.xsl"). When the Report XSLT file option is set, the build runner will apply an XSLT transform to FxCop XML output and display the resulting HTML in a new "FxCop" tab on the build results page.
Additional FxCopCmd options	Additional options for calling FxCopCmd executable. All options entered in this field will be added to the beginning of the command line parameters.

## Build failure conditions

Check the box to fail a build on the specified analysis errors. Click [build failure condition](#) to define the number of the errors.

## Using Service Messages

If you prefer to call the FxCop tool directly from the script, not as a build runner, you can use the `importData service` messages to import an xml file generated by [the FxCopCmd tool](#) into TeamCity. In this case the FxCop tool results will appear in the [Code Inspection tab](#) of the build results page.

The service message format is described below:

```
#teamcity[importData type='FxCop' path='<path to the xml file>']
```



The TeamCity agent will import the specified xml file in the background. Please make sure that the xml file is not deleted right after the `importData` message is sent.

## See also:

[Concepts: Build Runner](#)

## Gradle

The Gradle Build Runner runs [Gradle](#) projects.



To run builds with Gradle, you need to have Gradle 0.9-rc-1 or higher installed on all the agent machines. Alternatively, if you use the [Gradle wrapper](#), you need to have properly configured Gradle Wrapper scripts checked in to your Version Control.

## In this section:

- [Gradle Parameters](#)

- Run Parameters
- Java Parameters
- Build properties
- Docker Settings
- Code Coverage

#### Gradle Parameters

Option	Description
Gradle tasks	Specify Gradle task names separated by spaces. For example: <code>:myproject:clean :myproject:build</code> or <code>clean build</code> . If this field is left blank, the 'default' task is used. Note that TeamCity currently supports building Java projects with Gradle. Building Groovy/Scala/etc. projects has not been tested.
Incremental building	TeamCity can make use of the Gradle <code>:buildDependents</code> feature. If the Incremental building checkbox is enabled, TeamCity will detect Gradle modules affected by changes in the build, and start the <code>:buildDependents</code> command for them only. This will cause Gradle to fully build and test only the modules affected by changes.
Gradle home path	Specify here the path to the Gradle home directory (the parent of the <code>bin</code> directory). If not specified, TeamCity will use the Gradle from an agent's <code>GRADLE_HOME</code> environment variable. If you don't have Gradle installed on agents, you can use Gradle wrapper instead.
Additional Gradle command line parameters	Optionally, specify the space-separated list of command line parameters to be passed to Gradle.
Gradle Wrapper	If this checkbox is selected, TeamCity will look for Gradle Wrapper scripts in the checkout directory, and launch the appropriate script with Gradle tasks and additional command line parameters specified in the fields above. In this case, the Gradle specified in Gradle home path and the one installed on agent, are ignored.

#### Run Parameters

Option	Description
Debug	Selecting the Log debug messages check box is equivalent to adding the <code>-d</code> Gradle command line parameter.
Stacktrace	Selecting the Print stacktrace check box is equivalent to adding the <code>-s</code> Gradle command line parameter.

#### Java Parameters

Option	Description
JDK	Select a JDK. This section details the available options. The default is <code>JAVA_HOME</code> environment variable or the agent's own Java.
JDK home path	The option is available when <Custom> is selected above. Use this field to specify the path to your custom JDK used to run the build. If the field is left blank, the path to JDK Home is read either from the <code>JAVA_HOME</code> environment variable on agent the computer, or from the <code>env.JAVA_HOME</code> property specified in the <a href="#">build agent configuration</a> file ( <code>buildAgent.properties</code> ). If these values are not specified, TeamCity uses the Java home of the build agent process itself.
JVM command line parameters	You can specify such JVM command line parameters, e.g. maximum heap size or parameters enabling remote debugging. These values are passed by the JVM used to run your build. Example:  -Xmx512m -Xms256m

#### Build properties

The TeamCity system parameters can be accessed in Gradle build scripts in the [same way](#) as Gradle properties. The recommended way to reference properties is as follows:

```
task printProperty << {
    println "${project.ext['teamcity.build.id']}"
}
```

or if the system property's name is a legal Groovy name identifier (e.g. `system.myPropertyName = myPropertyValue`):

```
task printProperty << {
    println "$myPropertyName"
}
```

## Docker Settings

In this section, you can specify a Docker image which will be used to run the build step.

## Code Coverage

Code coverage with [IDEA code coverage engine](#) and JaCoCo is supported.

## See also:

[Administrator's Guide: IntelliJ IDEA Code Coverage](#)

## Inspections

The [Inspections \(IntelliJ IDEA\) Build Runner](#) is intended to run code analysis based on [IntelliJ IDEA inspections](#) for your project. Since TeamCity 2017.1, in addition to the bundled version, it is possible to install another version of JetBrains IntelliJ Inspections and Duplicates Engine and/or change the defaults using the [Administration | Tools](#) page.

IntelliJ IDEA's code analysis engine is capable of inspecting your Java, JavaScript, HTML, XML and other code and allows you to:

- Find probable bugs
- Locate "dead" code
- Detect performance issues
- Improve the code structure and maintainability
- Ensure the code conforms to guidelines, standards and specifications

Refer to [IntelliJ IDEA documentation](#) for more details.



To run inspections for your project, you must have either an IntelliJ IDEA project file (.ipr) or a project directory (.idea) checked into your version control.

The runner also supports Maven2 or above: to use `pom.xml`, you need to open it in IntelliJ IDEA and configure inspection profiles as described in the [IntelliJ IDEA documentation](#). IntelliJ IDEA will save your inspection profiles in the corresponding folder. Make sure you have it checked into your version control. Then specify the paths to the inspection profiles while configuring this runner.

This page contains reference information about the following Inspections (IntelliJ IDEA) Build Runner fields:

- [IntelliJ IDEA Project Settings](#)
- [Unresolved Project Modules and Path Variables](#)
- [Project SDKs](#)
- [Java Parameters](#)
- [Inspection Parameters](#)
- [Getting the same results in IntelliJ IDEA and TeamCity Inspections Build](#)

## IntelliJ IDEA Project Settings

Option	Description
--------	-------------

Project file type	To be able to run IntelliJ IDEA inspections on your code, TeamCity requires either an IntelliJ IDEA project file\directory, Maven pom.xml or Gradle build.gradle to be specified here.
Path to the project	<p>Depending on the type of project selected in the Project file type, specify here:</p> <ul style="list-style-type: none"> <li>For IntelliJ IDEA project: the path to the project file (.ipr) or the path to the project directory the root directory of the project containing the .idea folder).</li> <li>For Maven project: the path to the pom.xml file.</li> <li>For Gradle project: the path to the .gradle file.</li> </ul> <p>This information is required by this build runner to understand the structure of the project.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <span style="color: #0072bc; font-size: 1.2em;">i</span> The specified path should be relative to the checkout directory.         </div>
Detect global libraries and module-based JDK in the *.iml files	<p>This option is available if you use an IntelliJ IDEA project to run the inspections. In IntelliJ IDEA, the module settings are stored in *.iml files, thus, if this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDKs when saved. This helps ensure that all references will be properly resolved.</p> <div style="border: 1px solid #cc0000; padding: 5px; margin-top: 10px;"> <span style="color: #cc0000; font-size: 1.2em;">!</span> <b>Warning</b>            When this option is selected, the process of opening and saving the build runner settings may become time-consuming, because it involves loading and parsing all project module files.         </div>
Check/Reparse Project	<p>This option is available if you use an IntelliJ IDEA project to run the inspections. Click this button to reparse your IntelliJ IDEA project and import the build settings right from the project, for example the list of JDKs.</p> <div style="border: 1px solid #ffcc00; padding: 5px; margin-top: 10px;"> <span style="color: #ffcc00; font-size: 1.2em;">!</span> If you update your project settings in IntelliJ IDEA (e.g add new jdk, libraries), remember to update the build runner settings by clicking Check/Reparse Project.         </div>
Working directory	Enter a path to a <a href="#">Build Working Directory</a> if it differs from the <a href="#">Build Checkout Directory</a> .Optional, specify if differs from the checkout directory.

#### Unresolved Project Modules and Path Variables

This section is displayed when an IntelliJ IDEA module file (.iml) referenced from an IntelliJ IDEA project file:

- cannot be found
- allows you to enter the values of path variables used in the IPR-file.

To refresh the values in this section, click Check/Reparse Project.

Option	Description
<path_variable_name>	This field appears if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In Set value to field, specify a path to the project resources to be used on different build agents.

#### Project SDKs

This section provides the list of SDKs detected in the project.

Option	Description
JDK Home	<p>Use this field to specify the JDK home for the project.</p> <div style="border: 1px solid #ffcc00; padding: 5px; margin-top: 10px;"> <span style="color: #ffcc00; font-size: 1.2em;">!</span> When building with the Ipr runner, this JDK will be used to compile the sources of corresponding IDEA modules. For Inspections and Duplicate Finder builds, this JDK will be used internally to resolve the Java API used in your project.            To run the build process, the JDK specified in the JAVA_HOME environment variable will be used.         </div>

JDK Jar File Patterns	<p>Click this link to open a text area where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns.</p> <p>The default value is used for Linux and Windows operating systems:</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>jre/lib/*.jar</pre> </div>
	<p>For Mac OS X, use the following lines:</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>lib/*.jar</pre> <pre>../Classes/*.jar</pre> </div>
IDEA Home	If your project uses the IDEA JDK, specify the location of the IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

 You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a [requirement](#) for the corresponding property.

## Java Parameters

Option	Description
JDK	Select a JDK. <a href="#">This section</a> details the available options. The default is JAVA_HOME environment variable or the agent's own Java.
JDK home path	The option is available when <Custom> is selected above. Use this field to specify the path to your custom JDK used to run the build. If the field is left blank, the path to JDK Home is read either from the JAVA_HOME environment variable on agent the computer, or from the env.JAVA_HOME property specified in the <a href="#">build agent configuration</a> file (buildAgent.properties). If these values are not specified, TeamCity uses the Java home of the build agent process itself.
JVM command line parameters	You can specify such JVM command line parameters, e.g. maximum heap size or parameters enabling remote debugging. These values are passed by the JVM used to run your build. Example: <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>-Xmx512m -Xms256m</pre> </div>

## Inspection Parameters

In IntelliJ IDEA-based IDEs, the code inspections reported are configured by an [inspection profile](#).

When running the inspections in TeamCity, you can specify the inspection profile to use: first you need to configure the inspection profile in IntelliJ IDEA-based IDE and then specify it in TeamCity.

Follow these rules when preparing inspection profiles:

- if your inspection profile uses scopes, make sure the scopes are shared;
- lock the profile (this ensures that inspections present in TeamCity but not enabled in your IDEA installation will not be run by TeamCity);
- ensure the profile does not have inspections provided by plugins not included into the default IntelliJ IDEA Ultimate distribution (otherwise they will just be ignored by TeamCity);
- for best results, edit the inspection profile in the IntelliJ IDEA of the same version as used by TeamCity (can be looked up in the inspection build log).

The logic of selecting an inspection profile is as follows:

- if the path to the inspection profile is specified, then the profile will be loaded from the file. If the loading fails, the inspection runner will fail too.
- if the name of the inspection profile is specified, the profile is searched for in the project's shared profiles. If there is no such profile, the inspection runner will fail.
- if neither the name nor path is specified, the default profile of the project is used.

Option	Description
Inspections profile path	Use this text field to specify the path to inspections profiles file relative to the project root directory. Use this field only if you do not want to use the shared project profile specified in "Inspections profile name".
Inspections profile name	Enter the name of the desired shared project profile. If the field is left blank and no profile path is specified, the default project profile will be used.
Include / exclude patterns:	Optional, specify to restrict the sources scope to run Inspections on.

 Include / exclude patterns are newline-delimited set of rules of the form:

```
[+|-:]pattern
```

where the pattern must satisfy the following rules:

- must end with either \*\* or \* (this effectively limits the patterns to only the directories level, they do not support file-level patterns);
- references to modules can be included as [<IDEA\_module\_name>]/<path\_within\_module>. If you have a Maven project configured, you can use Maven module's artifactId as <IDEA\_module\_name>;
- the configured paths are treated as relative paths within content roots of the IDEA project modules. That is, the paths should be relative to the module's roots.

Some notes on patterns processing:

- excludes have precedence over includes
- if include patterns are specified, only directories matching these patterns will be included, all other directories will be excluded
- the include pattern has a special behavior (due to underlying limitations): it includes the directory specified and all the files residing directly in the directories above the one specified.

Example:

```
+: testData/tables/**  
-: testData/**  
-: testdata/**  
-: [testData]/**
```

 For the file paths to be reported correctly, the "References to resources outside project/module file directory" option for the project and all modules should be set to "Relative" in IDEA project.

For Maven inspections run, to ensure correct Java is used for the project JDK, define `env.JAVA_HOME` configuration parameter pointing to the JDK to be used as the project JDK.

Getting the same results in IntelliJ IDEA and TeamCity Inspections Build

The code inspections reported by IntelliJ IDEA and TeamCity Java Code Inspections build depend on a number of factors. You would need to ensure equal settings in IntelliJ IDEA and the build to get the same reports.

The relevant settings include:

- **inspections profile** used in IntelliJ IDEA and TeamCity build;
- environment-specific project dependencies (files not in version control, etc.);
- IDE-level settings, like defined SDKs, path variables, etc.;
- generated files: should be present in the TeamCity agent if they are present when working with the project in IntelliJ IDEA;

- IntelliJ IDEA version. It is recommended to use the same IntelliJ IDEA version that is used in the TeamCity build. TeamCity bundled an installation of IntelliJ IDEA. The version is written in the Inspections build log;
- the set and versions of the IntelliJ IDEA plugins that the project relies on.

## Inspections (ReSharper)

The Inspections (ReSharper) runner allows you to use the benefits of [JetBrains ReSharper code quality analysis](#) feature right in TeamCity using the bundled JetBrains ReSharper Command Line Tools. The usage of the tools within TeamCity does not require any additional licensing for ReSharper.

ReSharper analyzes your C#, VB.NET, XAML, XML, ASP.NET, ASP.NET MVC, JavaScript, HTML, CSS code and allows you to:

- Find probable bugs
- Eliminate errors and code smells
- Detect performance issues
- Improve the code structure and maintainability
- Ensure the code conforms to guidelines, standards and specifications

If you want to run ReSharper inspections using a specific ReSharper version (e.g. to ensure it matches the version you have installed in Visual Studio), you can install another version of the tools and change the default version to be used using the [Administration | Tools](#) page.

This page contains reference information about the Inspections (.Net) Build Runner fields:

- Sources to Analyze
- Environment Requirements
- JetBrains ReSharper Command Line Tools Settings
- InspectCode Options
- Build Failure Conditions
- Build before analyze

You can also refer to [ReSharper documentation](#) for more details.



To run inspections for your project, you must have a ReSharper inspection profile for .NET projects.

### Sources to Analyze

Option	Description
Solution file path	The path to .sln file created by Microsoft Visual Studio 2005 or later. The specified path should be relative to the checkout directory.
Projects filter	Specify project name wildcards to analyze only a part of the solution. Leave blank to analyze the whole solution. Separate wildcards with new lines. Example:  <pre>JetBrains.CommandLine.* *.Common *.Tests.*</pre>

### Environment Requirements



In order to launch inspection analysis, you should have .NET Framework 4.0 (or higher) installed on an agent where builds will run.

Option	Description
Target Frameworks	This option allows you to handle the <a href="#">Visual Studio Multi-Targeting</a> feature. Agent requirement will be created for every checked item.  .Net Framework versions 2.0 - 4.7.2 are supported.  <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">  .NET Frameworks client profiles are not supported as target frameworks. </div>

## JetBrains ReSharper Command Line Tools Settings

Option	Description
R# CLT Home Directory	Select the ReSharper Command Line Tools version. You can check the installed JetBrains ReSharper Command Line Tools versions on the <a href="#">Administration   Tools</a> page. If you want to run ReSharper duplicates using a specific ReSharper version (e.g. to ensure it matches the version you have installed in Visual Studio), you can use this page to install another version of the tools and can change the default version to be used.
InspectCode Platform	Select the platform bitness of the InspectCode tool. To find code issues in C++ projects, use the x86 platform.

## InspectCode Options

Option	Description
Custom settings profile path	The path to the file containing ReSharper settings created with JetBrains Resharper 6.1 or later. The specified path should be relative to the checkout directory. If specified, this settings layer has the top priority, so it overrides ReSharper build-in settings. By default, build-in ReSharper settings layers are applied. For additional information about ReSharper settings system, visit <a href="#">ReSharper Web Help</a> and <a href="#">JetBrains .NET Tools Blog</a>
Enable debug output	Check this option to include debug messages in the build log and publish the file with additional logs (dot net-tools-inspectcode.log) as a hidden artifact.
Additional inspectCode.exe arguments:	Specify newline-separated command line parameters to add to calling inspectCode.exe.   Only xml reports are supported by the runner. To get the output xml report, specify the path to the output file here via the -o or -output additional command line arguments. The paths relative to the <a href="#">build checkout directory</a> as well as absolute paths are supported.

## Build Failure Conditions

If a build has too many inspection errors or warnings, you can configure it to fail by setting a build failure condition.

### Build before analyze

In order to have adequate inspections execution results, you may need to build your solution before running analysis. This pre-step is especially actual when you use (implicitly or explicitly) code generation in your project.

### IntelliJ IDEA Project

IntelliJ IDEA Project runner allows you to build a project created in IntelliJ IDEA.

TeamCity versions up to 6.0 had [Ipr \(deprecated\)](#) which is now superseded by IntelliJ IDEA Project runner.

- Supported IntelliJ IDEA features
- IntelliJ IDEA Project Settings
- Unresolved Project Modules and Path Variables
- Project JDKs
- Java Parameters
- Compilation settings
- Artifacts
- Run configurations
- Test Parameters
- Code Coverage

### Supported IntelliJ IDEA features

TeamCity IntelliJ IDEA runner supports subset of IntelliJ IDEA features:

Feature	Status	Notes, limitations
Java		Runner is able to compile Java projects

JUnit 3.x/4.x	, with limitations	<ul style="list-style-type: none"> <li>• Test runner parameters are not supported</li> <li>• running Maven before tests start is not supported</li> <li>• alternative JRE is not supported</li> </ul>
TestNG	, with limitations	<ul style="list-style-type: none"> <li>• Test runner parameters are not supported</li> <li>• running Maven before tests start is not supported</li> <li>• running of the tests from group is not supported</li> <li>• alternative JRE is not supported</li> </ul>
Application run configuration	, with limitations	<ul style="list-style-type: none"> <li>• running Maven before tests start is not supported</li> <li>• altrenative JRE is not supported</li> </ul>
J2EE integration		runner is able to produce WAR and EAR archives with necessary descriptors
JPA		runner adds necessary descriptors in produced artifacts
GWT		runner can invoke GWT compiler and add compiler result to artifacts
Groovy	, with limitations	runner is able to compile projects with Groovy code and run tests written in Groovy, Groovy script run configurations are not supported
Android		
Flex		
Coverage	, if specified in run configurations	IntelliJ IDEA based coverage can be configured separately on the runner settings page
Profiling plugins		

#### IntelliJ IDEA Project Settings

Option	Description
Path to the project	<p>Use this field to specify the path to the project file (.ipr) or path to the project directory (root directory of the project that contains .idea folder). This information is required by this build runner to understand the structure of the project.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">  Specified path should be relative to the checkout directory.         </div>
Detect global libraries and module-based JDK in the *.iml files	<p>If this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDKs when saved. This helps you ensure all references will be properly resolved.</p> <div style="border: 2px solid red; padding: 10px; margin-top: 10px;"> <b>Warning</b>            When this option is selected, the process of opening and saving the build runner settings may become time-consuming, because it involves loading and parsing all project module files.         </div>
Check/Reparse Project	<p>Click to reparse the project and import build settings right from the IDEA project, for example the list of JDKs.</p> <div style="border: 1px solid orange; padding: 10px; margin-top: 10px;">  If you update your project settings in IntelliJ IDEA - add new jdks, libraries, don't forget to update build runner settings by clicking Check/Reparse Project.         </div>
Working directory	<p>Enter a path to a <a href="#">Build Working Directory</a>, if it differs from the <a href="#">Build Checkout Directory</a>.</p> <p>Optional, specify if differs from the checkout directory.</p>

#### Unresolved Project Modules and Path Variables

This section is displayed, when an IntelliJ IDEA module file (.iml) referenced from IPR-file:

- cannot be found

- allows you to enter the values of path variables used in the IPR-file.

To refresh values in this section click Check/Reparse Project.

Option	Description
<path_variable_name>	This field appears, if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In the Set value to field, specify a path to project resources, to be used on different build agents.

## Project JDKs

This section provides the list of JDKs detected in the project.

Option	Description
JDK Home	<p>Use this field to specify JDK home for the project.</p> <p> When building with the Ipr runner, this JDK will be used to compile the sources of corresponding IDEA modules. For Inspections and Duplicate Finder builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process itself the JDK specified in the <code>JAVA_HOME</code> environment variable will be used.</p>
JDK Jar File Patterns	<p>Click this link to open a text area, where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns. The default value is used for Linux and Windows operating systems:</p> <pre>jre/lib/*.jar</pre> <p>For Mac OS X, use the following lines:</p> <pre>lib/*.jar ./Classes/*.jar</pre>
IDEA Home	If your project uses the IDEA JDK, specify the location of IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

 You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a [requirement](#) for the corresponding property.

## Java Parameters

Option	Description
JDK	Select a JDK. <a href="#">This section</a> details the available options. The default is <code>JAVA_HOME</code> environment variable or the agent's own Java.
JDK home path	The option is available when <Custom> is selected above. Use this field to specify the path to your custom JDK used to run the build. If the field is left blank, the path to JDK Home is read either from the <code>JAVA_HOME</code> environment variable on agent the computer, or from the <code>env.JAVA_HOME</code> property specified in the <a href="#">build agent configuration</a> file ( <code>buildAgent.properties</code> ). If these values are not specified, TeamCity uses the Java home of the build agent process itself.

JVM command line parameters	You can specify such JVM command line parameters, e.g. maximum heap size or parameters enabling remote debugging. These values are passed by the JVM used to run your build. Example:  -Xmx512m -Xms256m
-----------------------------	---

#### Compilation settings

Option	Description
Only compile classes required to build artifacts and execute run configurations	Select whether to compile all classes in the project or only those classes which are required by run configurations or for building artifacts.

#### Artifacts

Option	Description
Artifacts to Build	Specify here names of the artifacts to be built that are configured in the IntelliJ IDEA project.

#### Run configurations

Option	Description
Run configurations to execute	Specify here names of IntelliJ IDEA run configurations configured in the project to execute inside TeamCity build. Supported configuration types are: JUnit, TestNG and Application. Note that run configurations specified here should be shared (via "Share" checkbox in IntelliJ IDEA Run/Debug Configurations dialog) and checked in to the version control.  Ant and Build Artifacts tasks specified in the Before launch list of IDEA run configurations are supported.

#### Test Parameters

- To learn more about Run recently failed tests first and Run new and modified tests first options, please refer to the [Running Risk Group Tests First](#) page.
- Run affected tests only (dependency based) option will take build changes into account. With this option enabled runner will compute modules affected by current build changes and will execute only those run configurations which depend on affected modules directly or indirectly.

#### Code Coverage

Specify code coverage options, for the details, refer to [IntelliJ IDEA Code Coverage](#) page.

#### See also:

[Administrator's Guide: IntelliJ IDEA Code Coverage](#)

#### Ipr (deprecated)

This runner provides ability to build [IntelliJ IDEA](#) projects in TeamCity.  
It is superseded by [IntelliJ IDEA Project](#) runner.

This page contains reference information about the IPR build runner fields:

- [Ipr Runner Deprecation](#)
- [IntelliJ IDEA Project Settings](#)
- [Unresolved Project Modules and Path Variables](#)
- [Project JDKs](#)
- [Additional Pre/Post Processing \(Ant\)](#)
- [JUnit Test Runner Settings](#)
- [Code Coverage](#)

#### Ipr Runner Deprecation

Since TeamCity 6.0 Ipr runner is deprecated in favor of [IntelliJ IDEA project](#) runner which uses another implementation approach. In one of the following major TeamCity releases all build configurations with Ipr runner will be automatically converted to IntelliJ IDEA project runner. Since the runners may function differently in specific configurations it is highly recommended to change your current Ipr runner-based configurations to the new runner and check your settings before the final Ipr runner disabling. Please also use the IntelliJ IDEA project runner for all newly created projects and let us know if you have any issues with it.

Apart from differences in the scope of supported IntelliJ IDEA project features, the runners are also different in approach to tests running and coverage.

Namely:

- EMMA coverage is not supported by IntelliJ IDEA project runner. We recommend migrating to IntelliJ IDEA coverage engine if you used EMMA
- in IntelliJ IDEA project runner JUnit tests are launched via IntelliJ IDEA shared run configurations as opposed to Ant's <junit> task in Ipr runner.

Here are the recommended steps to perform the migration from Ipr to IntelliJ IDEA project runner:

1. If your existing Ipr runner has [JUnit Test Runner Settings](#) configured, backup all the settings of the section, for example, into a text file.
2. If you have [code coverage settings](#) configured, save these settings also. (See also related [issue](#))
3. Change the runner type to IntelliJ IDEA Project. All your settings will be migrated except for JUnit and code coverage options.
4. To restore JUnit tests you will need to create a shared run configuration in IntelliJ IDEA and commit the corresponding file into the version control. The name of the run configuration can then be specified in the Run configurations to execute area.
5. For coverage, configure code coverage options anew using your saved settings.

#### IntelliJ IDEA Project Settings

Option	Description
Path to the project	<p>Use this field to specify the path to the project file (.ipr) or path to the project directory (root directory of the project that contains .idea folder). This information is required by this build runner to understand the structure of the project.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <span> i</span> Specified path should be relative to the checkout directory.         </div>
Detect global libraries and module-based JDK in the *.iml files	<p>If this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDKs when saved. This helps you ensure all references will be properly resolved.</p> <div style="border: 1px solid red; padding: 10px; margin-top: 10px;"> <span> !</span> Warning            When this option is selected, the process of opening and saving the build runner settings may become time-consuming, because it involves loading and parsing all project module files.         </div>
Check/Reparse Project	<p>Click to reparse the project and import build settings right from the IDEA project, for example the list of JDKs.</p> <div style="border: 1px solid orange; padding: 10px; margin-top: 10px;"> <span> !</span> If you update your project settings in IntelliJ IDEA - add new jdks, libraries, don't forget to update build runner settings by clicking Check/Reparse Project.         </div>
Working directory	<p>Enter a path to a <a href="#">Build Working Directory</a>, if it differs from the <a href="#">Build Checkout Directory</a>.</p> <p>Optional, specify if differs from the checkout directory.</p>

#### Unresolved Project Modules and Path Variables

This section is displayed, when an IntelliJ IDEA module file (.iml) referenced from IPR-file:

- cannot be found
- allows you to enter the values of path variables used in the IPR-file.

To refresh values in this section click Check/Reparse Project.

Option	Description

<path_variable_name>	This field appears, if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In the Set value to field, specify a path to project resources, to be used on different build agents.
----------------------	--

## Project JDKs

This section provides the list of JDKs detected in the project.

Option	Description
JDK Home	<p>Use this field to specify JDK home for the project.</p> <p> When building with the Ipr runner, this JDK will be used to compile the sources of corresponding IDEA modules. For Inspections and Duplicate Finder builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process itself the JDK specified in the <code>JAVA_HOME</code> environment variable will be used.</p>
JDK Jar File Patterns	<p>Click this link to open a text area, where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns. The default value is used for Linux and Windows operating systems:</p> <pre>jre/lib/*.jar</pre> <p>For Mac OS X, use the following lines:</p> <pre>lib/*.jar ./Classes/*.jar</pre>
IDEA Home	If your project uses the IDEA JDK, specify the location of IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

 You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a **requirement** for the corresponding property.

## Additional Pre/Post Processing (Ant)

Option	Description
Run before build	In the appropriate fields, enter the Ant scripts and targets (optional) that you want to run prior to starting the build. The path to the Ant file should be relative to the project root directory.
Run after build	In the appropriate fields, enter the Ant scripts and targets (optional) that you want to run after the build is completed. The path to the Ant file should be relative to the project root directory.

## JUnit Test Runner Settings

 JUnit test settings map to the attributes of JUnit task. For details, refer to <http://ant.apache.org/manual/OptionalTaskS/junit.html>

Option	Description
Test patterns	<p>Click the Type test patterns link, and specify the required test patterns in a text area. These patterns are used to generate parameters of the <code>batchtest</code> JUnit task section. Each pattern generates either <code>include</code> or <code>exclude</code> section. These patterns are also used to compose classpath for the test run. Each module mentioned in the patterns adds its classpath to the whole classpath.</p> <p>Each pattern should be placed on a separate line and has the following format:</p> <pre style="border: 1px solid black; padding: 10px; margin-top: 10px;">[-]moduleName:[testFileNamePattern]</pre> <p>where:</p> <ul style="list-style-type: none"> <li>• [-]: If a pattern starts with minus character, the corresponding files will be excluded from the build process.</li> <li>• moduleName : this name can contain wildcards.</li> <li>• [testFileNamePattern] : Default value for testFileNamePattern is <code>**/*Test.java</code> , i.e. all files ending with <code>Test.java</code> in all directories. You can use <a href="#">Ant syntax</a> for file patterns. The sample below includes all test files from modules ending with "test" and excludes all files from packages containing the "ui" subpackage:</li> </ul> <pre style="border: 1px solid black; padding: 10px; margin-top: 10px;">*test:/**/*Test.java -*:/**/ui/**/*.java</pre>
Search for tests	In IDEA project, a user can mark a source code folder as either "sources" or "test" root. This drop-down list allows you to specify directories to look for tests: <ul style="list-style-type: none"> <li>• Throughout all project sources: look for tests in both "sources" and "test" folders of your IDEA project.</li> <li>• In test sources only: look through the folders marked as tests root only.</li> </ul>
Classpath in Tests	By default the whole classpath is composed of all classpaths of the modules used to get tests from. The following two options define whether you will use the default classpath, or take it from the specified module.
Override classpath in tests	If this option is checked, you can define test classpath from a single, explicitly specified module.
Module name to use JDK and classpath from	If the option Override classpath in tests is checked, you have to specify the module, where the classpath to be used for tests is specified.
JUnit Fork mode	Select the desired fork mode from the combo box: <ul style="list-style-type: none"> <li>• Do not fork: fork is disabled.</li> <li>• Fork per test: fork is enabled, and each test class runs in a separate JVM</li> <li>• Fork once: fork is enabled, and all test classes run in a single JVM</li> </ul>
New classloader instance for each test	Check this option, if you want a new classloader to be instantiated for each test case. This option is available only if Do not fork option is selected.
Include Ant runtime	Check this option to add Ant classes, required to run JUnit tests. This option is available if fork mode is enabled (Fork per test or Fork once).
JVM executable	Specify the command that will be used to invoke JVM. This option is available if fork mode is enabled (Fork per test or Fork once).
Stop build on error	Check this option, if you want the build to stop if an error occurs during test run.

JVM command line parameters for JUnit	Specify JVM parameters to be passed to JUnit task.
Tests working directory	Specify the path to the working directory for tests.
Tests timeout	Specify the lapse of time in milliseconds, after which test will be canceled. This value is ignored, if Do not fork option is selected.
Reduce test failure feedback time	<p>Use following two options to instruct TeamCity to run some tests before others.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <span style="color: #0072bc; font-size: 1.2em; border-radius: 50%; padding: 2px 5px; margin-right: 5px;"></span> Tests reordering works the following way: TeamCity provides tests that should be run first (test classes), after that when a JUnit task starts, it checks whether it includes these tests. If at least one test is included, TeamCity generates a new fileset containing included tests only and processes it before all other filesets. It also patches other filesets to exclude tests added to the automatically generated fileset. After that JUnit starts and runs as usual.         </div>
Run recently failed tests first	If checked, in the first place TeamCity will run tests failed in previous finished or running builds as well as tests having high failure rate (a so called blinking tests)
Run new and modified tests first	If checked, before any other test, TeamCity will run tests added or modified in change lists included in the running build. <div style="border: 1px solid #ffcc00; padding: 5px; margin-top: 10px;"> <span style="color: #ffcc00; font-size: 1.2em; border-radius: 50%; padding: 2px 5px; margin-right: 5px;"></span> If both options are enabled at the same time, tests of the new and modified tests group will have higher priority, and will be executed in the first place.         </div>
Verbose Ant	Check this option, if the generated JUnit task has to produce verbose output in ant terms.

## Code Coverage

To learn about configuring code coverage options, please refer to the [Configuring Java Code Coverage](#) page.

## Maven

Note that you can create a new Maven-based build configuration [automatically from URL](#), and set up a dependency build trigger , if a specific Maven artifact has changed.



### Remote Run Limitations related to Maven runner

As a rule, a personal build in TeamCity doesn't affect any "regular" builds run on the TeamCity server, and its results are visible to its initiator only. However, in case of using Maven runner, this behavior may differ.

TeamCity doesn't interfere anyhow with the Maven dependencies model. Hence, if your Maven configuration deploys artifacts to a remote repository, they will be deployed there even if you run a personal build. Thereby, a personal build may affect builds that depend on your configuration.

For example, you have a configuration A that deploys artifacts to a remote repository, and these artifacts are used by configuration B. When a personal build for A has finished, your personal artifacts will appear in B. This can be especially injurious, if configuration A is to produce release-version artifacts, because proper artifacts will be replaced with developer's ones, which will be hard to investigate because of Maven versioning model. Plus these artifacts will become available to all dependent builds, not only to those managed by TeamCity.

To avoid this, we recommend not using remote run for build configurations which perform deployment of artifacts.

## On this page:

- [Maven runner settings](#)
  - [Maven Settings](#)
  - [User Settings](#)
  - [Java Parameters](#)
  - [Local Artifact Repository Settings](#)
  - [Incremental Building](#)
  - [Docker Settings](#)
  - [Code Coverage](#)
- [Maven Release with Different VCSs](#)

- [Using Maven Release with Perforce](#)
- [Using Maven Release with Git VCS](#)

#### Maven runner settings

Option	Description
Goals	<p>In the Goals field, specify the sequence of space-separated Maven goals that you want TeamCity to execute.</p> <p>Some Maven goals can use version control systems, and, thus, they may become incompatible with some <a href="#">VCS checkout modes</a>. If you want TeamCity to execute such a goal:</p> <ul style="list-style-type: none"> <li>• Select "Automatically on agent" in the <a href="#">VCS Checkout Mode</a> drop-down list on the Version Control Settings page. This makes the version control system available to the goal execution software.</li> </ul> <p> To use the <code>release:prepare</code> goal with Perforce VCS, see the <a href="#">section</a> below.</p>
Path to POM file	<p>Specify the path to the POM file relative to the <a href="#">build working directory</a>.</p> <p>By default, the property contains a <code>pom.xml</code> file. If you leave this field blank, the same value is put in this field. The path may also point to a subdirectory, and as such <code>&lt;subdirectory&gt;/pom.xml</code> is used.</p>
Additional Maven command line parameters	<p>Specify the list of command line parameters.</p> <div style="border: 1px solid #f0e68c; padding: 5px; margin-top: 10px;">  The following parameters are ignored: <code>-q</code>, <code>-f</code>, <code>-s</code> (if User settings path is provided)     </div>
Working directory	Specify the <a href="#">Build Working Directory</a> if it differs from the <a href="#">build checkout directory</a> .

#### Maven Settings

Choose the Maven version you want to use. Since TeamCity 2017.1, you can [manage the installed versions](#).

<Auto>	The path to Maven installation is taken from the M2_HOME environment variable, otherwise the current default version is used.
<Default>	The bundled version 3.0.5 is used as default. Since TeamCity 2017.1, you can <a href="#">change the defaults</a> .
<Custom>	Provide a path to a custom Maven version.

#### User Settings

Specify what kind of user settings to use here. This is equivalent to the Maven command line option `-s` or `--settings`. The available options are:

<Default>	Settings are taken from the default Maven locations on the agent. For the server logic, see <a href="#">Maven Server-Side Settings</a> .
<Custom>	Enter the path to an alternative user settings file. The path should be valid on agent and also on the server, see <a href="#">Maven Server-Side Settings</a> .
Predefined settings	<p>If there are settings files uploaded to the TeamCity server via the administration UI, you can select one of the available options here. To upload settings file to TeamCity, click <a href="#">Manage settings files</a>.</p> <p>Maven settings are defined on the project level. You can see the settings files defined in the current project or upload files on the Project Settings page using Maven Settings. The files will be available in the project and its subprojects. The uploaded files are stored in the <code>&lt;TeamCity Data Directory&gt;/config/projects/%projectId%/pluginData/mavenSettings</code> directory. If necessary, they can be edited right there. The uploaded files are used both for the agent and server-side Maven functionality.</p> <p>If Custom or Predefined settings are used, the path to the effective user settings file is available inside the maven process as the <code>teamcity.maven.userSettings.path</code> system property.</p>

#### Java Parameters

Option	Description
JDK	Select a JDK. <a href="#">This section</a> details the available options. The default is JAVA_HOME environment variable or the agent's own Java.

JDK home path	The option is available when <Custom> is selected above. Use this field to specify the path to your custom JDK used to run the build. If the field is left blank, the path to JDK Home is read either from the JAVA_HOME environment variable on agent the computer, or from the env.JAVA_HOME property specified in the <a href="#">build agent configuration</a> file (buildAgent.properties). If these values are not specified, TeamCity uses the Java home of the build agent process itself.
JVM command line parameters	You can specify such JVM command line parameters, e.g. maximum heap size or parameters enabling remote debugging. These values are passed by the JVM used to run your build. Example: <pre>-Xmx512m -Xms256m</pre>

### Local Artifact Repository Settings

Select Use own local repository for this build configuration to isolate this build configuration's artifacts repository from other local repositories.

### Incremental Building

Select the Build only modules affected by changes check box to enable incremental building of Maven modules. The general idea is that if you have a number of modules interconnected by dependencies, a change most probably affects (directly or transitively) only some of them; so if we build only the affected modules and take the result of building the rest of the modules from the previous build, we will get the overall result equal to the result of building the whole project from scratch with less effort and time.

Since Maven itself has very limited support for incremental builds, TeamCity uses its own change impact analysis algorithm for determining the set of affected modules and uses a special preliminary phase for making dependencies of the affected modules.

First TeamCity performs own change impact analysis taking into account parent relationship and different dependency scopes and determines affected modules. Then the build is split into two sequential Maven executions.

The first Maven execution called preparation phase is intended for building the dependencies of the affected modules. The preparation phase is to assure there will be no compiler or other errors during the second execution caused by the absence or inconsistency of dependency classes.

The second Maven execution called main phase executes the main goal (for example, `test`), thus performing only those tests affected by the change.

Also check a related [blog post](#) on the topic.

### Docker Settings

In this section, you can specify a Docker image which will be used to run the build step.

### Code Coverage

Coverage support based on IDEA coverage engine is added to Maven runner. To learn about configuring code coverage options, please refer to the [Configuring Java Code Coverage](#) page.



Note: only Surefire version 2.4 and higher is supported.



If you have several build agents installed on the same machine, by default they use the same local repository. However, there are two ways to allocate a custom local repository to each build agent:

- Specify the following property in the `teamcity-agent/conf/buildAgent.properties`:

```
system.maven.repo.local=%system.agent.work.dir%/<subdirectory name>
```

For instance, `%system.agent.work.dir%/m2-repository`

- Run each build agent under different user account.

## Maven Release with Different VCSs

To run the `release:prepare` maven task with different VCS's supported by TeamCity, make sure you're using at least 2.0 version of the [Maven Release Plugin](#).

### Using Maven Release with Perforce

Check the following:

1. Use [ticket-based authentication](#) for Maven Release plugin.
2. Make sure that your `release:prepare` maven task works when it is run from the command line without TeamCity.

In the [Perforce VCS Root Settings](#) of your build configuration in TeamCity:

1. Enable the [checkout on agent](#).
2. Enable [Use ticket-based authentication](#) in Perforce VCS root settings.
3. Make sure your build agent environment doesn't have any occasional P4 variables which can interfere with the execution of Maven Release Plugin.
4. Specify `release:prepare` in the Goals field of the Maven build step and run the build.

### Using Maven Release with Git VCS

1. Use [Git SSH URL](#) as [SCM URL](#) in your pom.xml.
2. Make sure that your `release:prepare` maven task works when it is run from the command line without TeamCity.

On the TeamCity agent:

1. Make sure that the agent has Git installed and added to the agent's \$PATH on Unix-like OS's and to %PATH% environment variable on Windows .
2. On the agent, set your account's identity by executing

```
git config --system user.email "buildserver@example.com"  
git config --system user.name "TeamCity Server"
```

3. Make sure your Git VCS is added to the known hosts database on the agent.

On the TeamCity server:

1. Upload [Git SSH key](#) to your TeamCity server.

In the settings for your build configuration in TeamCity:

2. On the Version Control Settings page, enable the checkout on agent.
3. In your Git VCS root, enable Private Key authentication.
4. Add the [SSH Agent](#) Build feature to your configuration.
5. Specify `release:prepare` in the Goals field of the Maven build step and run the build.

See also:

[Concepts: Build Runner](#)

[Administrator's Guide: Maven Artifact Dependency Trigger | Creating Maven Build Configuration](#)

## MSBuild

This page contains reference information for the MSBuild Build Runner fields.



The MSBuild runner requires .Net Framework or Mono installed on the build agent. Microsoft Build Tools 2013-2017 are supported.

Before setting up a build configuration to use MSBuild as the build runner, make sure you are using an XML build project file with the MSBuild runner.

To build a Microsoft Visual Studio solution file, you can use the [Visual Studio \(.sln\)](#) build runner.

- General Build Runner Options
- Code Coverage
- Implementation notes

## General Build Runner Options

Option	Description
Build file path	Specify the path to the solution to be built relative to the <a href="#">build checkout directory</a> . For example:  vs-addin\addin\addin.sln
Working directory	Optional. Specify the path to the <a href="#">build working directory</a> if it differs from the build checkout directory.
MSBuild version	Select the MSBuild version: .NET Framework, Mono xbuild or Microsoft Build Tools.
MSBuild ToolsVersion	Specify here the version of tools that will be used to compile (equivalent to the <code>/toolsversion:</code> commandline argument).   MSBuild supports compilation to older versions, thus you may set MSBuild version as 4.0 with MSBuild ToolsVersion set to 2.0 to produce .NET 2.0 assemblies while running MSBuild from .NET Framework 4.0. For more information, refer to <a href="http://msdn.microsoft.com/en-us/library/bb383796(VS.100).aspx">http://msdn.microsoft.com/en-us/library/bb383796(VS.100).aspx</a>
Run platform	From the drop-down list select the desired execution mode on a x64 machine.
Targets	A target is an arbitrary script for your project purposes. Enter targets separated by spaces. The available targets can be viewed in the Web UI by clicking the icon next to the field and added by checking the appropriate boxes.
Command line parameters	Specify any additional parameters for <code>MSBuild.exe</code> .
Reduce test failure feedback time	Use this option to instruct TeamCity to run the tests which failed in the previous builds before others.

## Code Coverage

To learn about configuring code coverage options, please refer to the [Configuring .NET Code Coverage](#) page.

## Implementation notes

MSBuild runner generates an MSBuild script that includes user's script. This script is used to add TeamCity-provided MSBuild tasks. Your MSBuild script will be included with the `<Import>` task. If you specified a Visual Studio solution file, it will be called from the `<MSBuild>` task. To disable it, set `teamcity.msbuild.generateWrappingScript` to `false`.

## See also:

[Concepts: Build Runner | Build Checkout Directory](#)  
[Administrator's Guide: NUnit for MSBuild | MSBuild Service Tasks](#)

## MSpec

The MSpec Test Runner is designed specifically to run MSpec tests.



To run tests using MSpec, you need to install it on at least one build agent.

## MSpec Settings

Option	Description
Path to MSpec.exe	A path to <code>mspec.exe</code> file.
.NET Runtime	From the Platform drop-down, select the desired execution mode on a x64 machine. Supported values are: Auto (MSIL) (default), x86 and x64. From the Version drop-down, select the desired .NET Framework version.   If you have MSpec as an MSIL .NET 2.0/3.5 executable, TeamCity can enforce it to execute under any required runtime: x86 or x64, and .NET2.0 or .NET 4.0 runtime.
Run tests from	Specify the .NET assemblies where the MSpec tests to be run are stored.
Do not run tests from	Specify the .NET assemblies that should excluded from the list of found assemblies to test.
Include specifications	Specify comma- or newline separated list of specifications to be executed.
Exclude specifications	Specify comma- or new line separated list of specifications to be excluded.
Additional commandline parameters	Enter additional commandline parameters for <code>mspec.exe</code> .

## Code Coverage

Learn about [configuring code coverage options](#).

## See also:

[Administrator's Guide: Configuring .NET Code Coverage](#)

## MSTest



### Redirection Notice

This page will redirect to [Visual Studio Tests](#).

## NAnt

TeamCity supports NAnt starting from version 0.85.



The TeamCity NAnt runner requires .Net Framework or Mono installed on the build agent.

## MSBuild Task for NAnt

TeamCity NAnt runner includes a task called `msbuild` that allows NAnt to start MSBuild scripts. TeamCity `msbuild` task for NAnt has the same set of attributes as the [NAntContrib package](#) `msbuild` task. The MSBuild build processes started by NAnt will behave exactly as if they were launched by TeamCity MSBuild/SLN2005 build runner (i.e. `NUnit` and/or `NUnitTeamCity` MSBuild tasks will be added to build scripts and logs and error reports will be sent directly to the build server).



`msbuild` task for NAnt makes all build configuration system properties available inside MSBuild script. Note, all property names will have `'.'` replaced with `'_'`.

To disable this, set `false` to `set-teamcity-properties` attribute of the task.

By default, NAnt `msbuild` task checks for current value of NAnt target-framework property to select MSBuild runtime version. This parameter could be overriden by setting `teamcity_dotnet_tools_version` project property with required .NET Framework version, i.e. "4.0".

```

...
<!-- this property enables MSBuild 4.0 -->
<property name="teamcity_dotnet_tools_version" value="4.0"/>
<msbuild project="SimpleEcho.v40.proj">
    ...
</msbuild>
...

```

 To pass properties to MSBuild, use the `property` tag instead of explicit properties definition in the command line.

<nunit2> Task for NAnt

To test all of the assemblies without halting on first failed test please use:

```

<target name="build">
    <nunit2 verbose="true" haltonfailure="false" failonerror="true" failonfailureatend="true">
        <formatter type="Plain" />
        <test haltonfailure="false">
            <assemblies>
                <include name="dll1.dll" />
                <include name="dll2.dll" />
            </assemblies>
        </test>
    </nunit2>
</target>

```

 'failonfailureatend' attribute is not defined in the original `NUnit2` task from NAnt. Note that this attribute will be ignored if the 'haltonfailure' attribute is set to 'true' for either the `nunit2` element or the `test` element.

Below you can find reference information about NAnt Build Runner fields.

#### General Options

Option	Description
Path to a build file	Specify path relative to the <b>Build Checkout Directory</b> .
Build file content	Select the option, if you want to use a different build script than the one listed in the settings of the build file. When the option is enabled, you have to type the build script in the text area.
Targets	Specify the names of build targets defined by the build script, separated by spaces. The available targets can be viewed in the Web UI by clicking the icon next to the field and added by checking the appropriate boxes.
Working directory	Specify the path to the <b>Build Working Directory</b> . By default, the build working directory is set to the same path as the <b>build checkout directory</b> .
NAnt home	Enter a path to the <code>NAnt.exe</code> .
	 Here you can specify an absolute path to the <code>NAnt.exe</code> file on the agent, or a path relative to the checkout directory. Such relative path allows you to provide particular <code>NAnt.exe</code> file to run a build of the particular build configuration.

Target framework	Sets <code>-targetframework</code> : option to 'NAnt' and generates appropriate agent requirements (mono-2.0 target framework will require Mono system property, net-2.0 — DotNetFramework2.0 property, and so on). Selecting unsupported in TeamCity framework (sscli-1.0, netcf-1.0, netcf-2.0) won't impose any agent requirements.
	<p> This option has no effect on framework which used to run <code>NAnt.exe</code>. <code>NAnt.exe</code> will be launched as ordinary exe file if .NET framework was found, and through mono executable, if not.</p>
Command line parameters	Specify any additional parameters for <code>NAnt.exe</code>
	<p> TeamCity passes automatically to NAnt all defined <b>system properties</b>, so you do not need to specify all of the properties here via '-D' option.</p>

## Code Coverage

To learn about configuring code coverage options, please refer to the [Configuring .NET Code Coverage](#) page.

## See also:

Concepts: [Build Runner](#) | [Build Checkout Directory](#) | [Build Working Directory](#)  
Administrator's Guide: [Configuring Build Parameters](#)

## NuGet Pack

The NuGet Pack build runner allows building a NuGet package from a given specification file. If you want to publish this package, add a [NuGet Publish](#) build step.



### Supported Operating Systems

NuGet build runners are supported on build agents running Windows OS by default. Linux and macOS are supported when [Mono](#) is installed on the agent (NuGet command-line executable for Windows version 3.2 and later is strongly recommended with Mono).

Configure the following options of the NuGet Pack runner:

Option	Description
NuGet.exe	Select a NuGet version to use from the drop-down list (you need to have <a href="#">NuGet installed</a> ), or specify a custom path to <code>NuGet.exe</code> .
	<p> An appropriate version of .NET Framework installed on the agent machine is required depending on the NuGet.exe version used: NuGet 2.8.6+ requires .NET 4.5+, earlier NuGet versions require .NET 4.0.</p>
Specification files	Enter path(s) to <code>csproj</code> or <code>nuspec</code> file(s). You can specify as many specification files here as you need. Wildcards are supported. If you specify here a <code>csproj</code> file, you won't have to redefine the version number and copyright information in the spec file.
Prefer project files to .nuspec	Check the box to use the project file (if exists, i.e. <code>.csproj</code> or <code>.vbproj</code> ) for every matched <code>.nuspec</code> file.

Version	Specify the package version. Overrides the version number from the <code>nuspec</code> file. You can use the TeamCity variable <code>%build.number%</code> here.
Base Directory	Select an option from the drop down list to specify the directory where the files defined in the <code>nuspec</code> file are located (the directory against which the paths in <code>&lt;files&gt;&lt;/files&gt;</code> from <code>nuspec</code> are resolved, usually some bin directory). If Use explicit directory is set and the field is left blank, TeamCity will use the build checkout directory as the base directory.
Output Directory	Specify the path where the generated NuGet package is to be put.
Clean output directory	Check the box to clean the directory before packing.
Publish created packages to build artifacts	Check the box if you're using TeamCity as a NuGet repository to publish packages to the TeamCity's NuGet server and be able to use them as regular TeamCity artifacts.
Exclude files	Specify one or more wildcard patterns to exclude when creating a package. Equivalent to the <code>NuGet.exe -Exclude</code> argument.
Properties	Semicolon or new-line separated list of package creation properties. For example, to make a release build, you define here <code>Configuration=Release</code> .
Options	Create tool package - check the box to place the output files of the project to the tool folder. Include sources and symbols - check the box to create a package containing sources and symbols. When specified with a <code>nuspec</code> , it creates a regular NuGet package file and the corresponding symbols package (needed for publishing the sources to <a href="#">SymbolsSource</a> )
Command line parameters	Set additional command line parameters to be passed to <code>NuGet.exe</code> .

See also:

[Administrator's Guide: NuGet Installer | NuGet Publish](#)

## NuGet Publish

The NuGet Publish build runner is intended to publish ([push](#)) your NuGet packages to a given feed (custom or default).



When using TeamCity as a NuGet server, there are two ways to publish packages to the feed:

- as build artifacts of the [NuGet Pack](#) build step using the Publish created packages to build artifacts checkbox - in this case you do not need the NuGet Publish build step
- via the NuGet Publish build step



### Supported Operating Systems

NuGet build runners are supported on build agents running Windows OS by default. Linux and macOS are supported when [Mono](#) is installed on the agent (NuGet command-line executable for Windows version 3.2 and later is strongly recommended with Mono).

This page describes the NuGet Publish runner options:

Option	Description
--------	-------------

NuGet.exe	Select a NuGet version to use from the drop-down list (you need to have <a href="#">NuGet installed</a> ), or specify a custom path to <code>NuGet.exe</code> .
	<p> An appropriate version of .NET Framework installed on the agent machine is required depending on the NuGet.exe version used: NuGet 2.8.6+ requires .NET 4.5+, earlier NuGet versions require .NET 4.0.</p>
Packages	Specify a newline-separated list of NuGet package files ( <code>.nupkg</code> ) to publish to the NuGet feed. List packages individually or use wildcards.
API key	<p>Specify the API key to access a NuGet packages feed.</p> <p>To publish to the TeamCity NuGet server, specify the <code>%teamcity.nuget.feed.api.key%</code> parameter.</p>
Package Source	<p>Specify the destination NuGet packages feed URL to push packages to, e.g. <a href="#">nuget.org</a>. Leave blank to let NuGet decide what package repository to use.</p> <p>To use a <a href="#">TeamCity NuGet feed</a>, specify the URL from the NuGet Feed project settings page. Replacing existing package version in TeamCity internal feed</p> <p>When publishing a package with the same version that already exists in a TeamCity internal NuGet feed, the package will be rejected. To force the TeamCity server to replace the existing NuGet package with a new version, append your feed URL obtainable from the project settings page with the <code>?replace=true</code> parameter, e.g. <code>http://&lt;Teamcity URL&gt;/httpAuth/app/nuget/feed/NuGet/default/v2?replace=true</code></p>
Command line parameters	Enter additional parameters to use when calling the <code>nuget push</code> command.

See also:

[Administrator's Guide: NuGet Installer | NuGet Pack](#)

## NuGet Installer

The NuGet Installer build runner performs NuGet [Command-Line Package Restore](#). It can also (optionally) automatically update package dependencies to the most recent ones.

 Supported Operating Systems  
NuGet build runners are supported on build agents running Windows OS by default. Linux and macOS are supported when [Mono](#) is installed on the agent (NuGet command-line executable for Windows version 3.2 and later is strongly recommended with Mono).

 Make sure that sources that you check out from VCS ([VCS Settings](#)) include the folder called `packages` from your solution folder.

## NuGet Installer settings:

Option	Description
NuGet.exe	Select NuGet version to use from the drop-down list (you need to have <a href="#">NuGet installed</a> ) or specify a custom path to <code>NuGet.exe</code> .
	<p> An appropriate version of .NET Framework installed on the agent machine is required depending on the NuGet.exe version used: NuGet 2.8.6+ requires .NET 4.5+, earlier NuGet versions require .NET 4.0.</p>
Path to solution file	Specify the path to your solution file ( <code>.sln</code> ) where packages are to be installed.
Restore Mode	Select <code>NuGet.exe restore</code> (requires NuGet 2.7+) to restore all packages for an entire solution. The <code>NuGet.exe install</code> command is used to restore packages for versions prior to NuGet 2.7, but only for a single <code>packages.config</code> file.

Restore Options	If needed, select: <ul style="list-style-type: none"> <li>Exclude version from package folder names: Equivalent to the <code>-ExcludeVersion</code> option of the <code>NuGet.exe install</code> command. If enabled, the destination folder will contain only the package name, not the version number.</li> <li>Disable looking up packages from local machine cache: Equivalent to the <code>-NoCache</code> option of the <code>NuGet.exe</code></li> </ul>
Packages Sources	Specify the NuGet package sources. If left blank, <a href="http://nuget.org">http://nuget.org</a> is used to search for your packages. You can also specify your own NuGet repository. If you are using TeamCity as a NuGet repository, select the TeamCity feed using the 'magic wand' icon  or manually specify the URL from the <a href="#">NuGet Feed</a> project settings page. If you use packages from an authenticated feed, configure the <a href="#">NuGet Feed Credentials</a> build feature.
Update Packages	Update packages with help of NuGet update command: Uses the <code>NuGet.exe update</code> command to update all packages under the solution. The package versions and constraints are taken from <code>packages.config</code> files.
Update Mode	Select one of the following: <ul style="list-style-type: none"> <li>Update via solution file - TeamCity uses Visual Studio solution file (<code>.sln</code>) to create the full list of NuGet packages to install. This option updates packages for the entire solution.</li> <li>Update via <code>packages.config</code> - Select to update packages via calls to <code>NuGet.exe update Packages.Config</code> for each <code>packages.config</code> file under the solution.</li> </ul>
Update Options	<ul style="list-style-type: none"> <li>Include pre-release packages: Equivalent to the <code>-Prerelease</code> option of the <code>NuGet.exe update</code> command</li> <li>Perform safe update: Equivalent to the <code>-Safe</code> option of the <code>NuGet.exe update</code> command, that looks for updates with the highest version available within the same major and minor version as the installed package.</li> </ul>

See the [NuGet documentation](#) for complete `NuGet.exe` command line reference.

When you add the NuGet Installer runner to your build configuration, each finished build will have the NuGet Packages tab listing the packages used.

#### See also:

A related TeamCity blog post.  
[Administrator's Guide: NuGet Pack | NuGet Publish](#)

#### NUnit

The NUnit build runner is intended to run NUnit tests right on the TeamCity server. However, there are other ways to report NUnit tests results to TeamCity, please refer to the [NUnit Support](#) page for the details.

**i** Supported NUnit versions: 2.2.10, 2.4.1, 2.4.6, 2.4.7, 2.4.8, 2.5.0, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3. Since TeamCity 9.1, NUnit 3.0 and above is also supported.  
 It is possible to have several versions of NUnit installed on an agent machine and use any of them in a build.

**!** NUnit version 3.4.0 is not supported by the NUnit build runner due to a problem in [NUnit](#). Only version 3.4.0 was affected, other NUnit 3.x versions work fine with TeamCity.

- [NUnit 3 Requirements](#)
  - [Installing NUnit](#)
  - [Installing Extensions](#)
- [NUnit Test Settings](#)
- [Code Coverage](#)

#### NUnit 3 Requirements

##### Installing NUnit

To use the TeamCity NUnit build runner, you need to install the [NUnit NuGet package](#) on TeamCity agents first.

To do that, use one of the following options:

- you can add the NuGet install build step as the first step of your build configuration. For example, you can add a command line build step before the NUnit build step which will install the NUnit.Console NuGet package as follows:

```
%teamcity.tool.NuGet.CommandLine.DEFAULT%\tools\nuget.exe install NUnit.Console -version 3.6.0 -o packages
```

Note that %teamcity.tool.NuGet.CommandLine.DEFAULT% is a reference to NuGet installed under the TeamCity agent. You can install NuGet on agents from the Administration | Tools page, where you can also mark one of the installed NuGet versions as default.

After that the %teamcity.tool.NuGet.CommandLine.DEFAULT% parameter reference should properly resolve to the NuGet installation path on the agent.

Then nunit3-console should appear under the packages directory.

To run tests, in the next NUnit build step, specify the NUnit path in the NUnit settings as packages\NUnit.ConsoleRunner.3.6.0\tools\nunit3-console.exe.

- Another approach is to install NUnit manually on all of the agents in some standard place, and configure the path to nunit-console.exe in your NUnit build step.

## Installing Extensions

Starting from version 3.2.0, NUnit requires the `NUnit.Extension.NUnitProjectLoader` extension to be installed on the TeamCity agent.

Starting from version 3.4.1, NUnit requires the `NUnit.Extension.TeamCityEventListener` extension to be installed on the TeamCity agent.

The NUnit runner checks for the extensions, and if they are not found, in versions 3.2.0 and 3.2.1 the build will fail without a warning; since version 3.4.1 a message is displayed suggesting you install them.

The extensions can be installed in bulk using the [NUnit Console Version 3](#) NuGet package or as separate packages, [NUnit.Extension.TeamCityEventListener](#) and [NUnit.Extension.NUnitProjectLoader](#).

## NUnit Test Settings

NUnit runner	Select the NUnit version to be used to run the tests. The number of settings available will vary depending on the selected version.
Path to NUnit console tool:	Available if NUnit 3.0 is selected. Specify the path to nunit3-console.exe: prior to TeamCity 9.1.4 specify the directory containing the console executable file, since 9.1.4 specify the path to the console executable file including the file name.
Working directory	Available if NUnit 3.0 is selected. Optional. Specify the path to the <a href="#">build working directory</a> if it differs from the directory of the testing assembly.
Path to application configuration file	Available if NUnit 3.0 is selected. Specify the path to the application configuration file to be used when running tests. The path is absolute or relative to the <a href="#">checkout directory</a> .
Additional command line parameters	Available if NUnit 3.0 is selected. Enter <a href="#">additional command line parameters</a> to pass to nunit-console.exe
.NET Runtime	From the Platform drop-down, select the desired execution mode on a x64 machine. Supported values are: Auto (MSIL), x86 and x64. From the Version drop-down, select the desired .NET Framework version. Supported values are: v2.0, v4.0; and <auto>, available if NUnit 3.0 is selected. <b>i</b> For NUnit 3.0, if <auto> is selected, tests will run under the framework they are compiled with.

Run tests from	<p>Specify the .NET assemblies where the NUnit tests to be run are stored. Multiple entries are comma-separated; usage of MSBuild wildcards is enabled. The paths to assembly files are absolute or relative to the <a href="#">checkout directory</a>.</p> <p>In the following example, TeamCity will search for the tests assemblies in all project directories and run these tests.</p> <pre>**\*.dll</pre> <p> All these wildcards are specified relative to the checkout directory.</p>
Do not run tests from	<p>Specify .NET assemblies that should be excluded from the list of assemblies to test. Multiple entries are comma-separated; usage of MSBuild wildcards is enabled. The paths to assembly files are absolute or relative to the <a href="#">checkout directory</a>.</p> <p>In the following example, TeamCity will omit tests specified in this directory.</p> <pre>**\obj\**\*.dll</pre> <p> All these wildcards are specified relative to the checkout directory.</p>
Include categories	<p>Specify NUnit categories of tests to be run. Multiple entries are comma-separated.</p> <p><a href="#">Category expressions</a> are supported here as well; commas, semicolons, and new-lines are treated as global or operations (prior to the expression parsing). Since NUnit 3.0 category expressions are not supported and must be set via the command line as <a href="#">described in the NUNit documentation</a>.</p>
Exclude categories	<p>Specify NUnit categories to be excluded from the tests to be run. Multiple entries are comma-separated.</p> <p><a href="#">Category expressions</a> are supported here as well; commas, semicolons, and new-lines are treated as global or operations (prior to the expression parsing). Since NUnit 3.0 category expressions are not supported and must be set via the command line as <a href="#">described in the NUNit documentation</a>.</p>
Run process per assembly	Available for versions <a href="#">prior to NUnit 3.0</a> only (since TeamCity 9.1.6). Select this option if you want to run each assembly in a new process.
Reduce test failure feedback time	Use this option to instruct TeamCity to run some tests before others.

## Code Coverage

To learn about configuring code coverage options, please refer to the [Configuring .NET Code Coverage](#) page.

 For NUnit 3.x, only [JetBrains dotCover](#) is supported as a coverage tool.

## See also:

[Administrator's Guide: Configuring Unit Testing and Code Coverage](#) | [NUnit Support](#) | [Getting Started with NUnit](#)

## PowerShell

The PowerShell [build runner](#) is specifically designed to run PowerShell scripts.

The plugin responsible for PowerShell integration has been opensourced on GitHub: <https://github.com/JetBrains/teamcity-powershell>

 If you need to run a PowerShell script with elevated permissions, consider using the TeamCity RunAs plugin.

On this page:

- [Cross-Platform PowerShell](#)
- [PowerShell Settings](#)
- [Docker Settings](#)
  - [Current limitations](#)
- [Interaction with TeamCity](#)
- [Error Handling](#)
- [Handling Output](#)
- [Temporary Files](#)
- [Development Links](#)

#### Cross-Platform PowerShell

- Cross-platform PowerShell ([PowerShell Core](#)) is supported on Windows, MacOS and Linux: [download a PowerShell package](#) for your platform and install it on the TeamCity agent.
- Side-by-side installations of PowerShell Desktop and PowerShell Core is supported under Windows.

#### PowerShell Settings

Option		Description
Version		List of PowerShell versions supported by TeamCity. It is passed to <code>powershell.exe</code> as the <code>-Version</code> command line argument.
PowerShell run mode		Select the desired execution mode on a x64 machine:
	Version	<p>Specify a version (e.g. 1.0, 2.0, etc.). It will be compared to the version installed on the agent, and an appropriate requirement will be added. For Core editions, it will be used as the lower bound. On Desktop editions, the exact version will be used (<code>-Version</code> command line argument will be added).</p> <p>If the version field is left blank, no lower bound on the version requirement will be added, no <code>-Version</code> argument will be used on Desktop editions of PowerShell.</p>
Format stderr output as:	Platform	<p>Select platform bitness:</p> <ul style="list-style-type: none"><li>■ x64 - 64-bit, default, the corresponding requirement will be added</li><li>■ x86 - 32-bit, the corresponding requirement will be added</li><li>■ Auto - when it is selected, no platform requirement will be added to the build configuration, and if both 32-bit and 64-bit PowerShells are installed, 64-bit will be preferred.</li></ul>
	Edition	<p>Select a PowerShell edition to be used: (since TeamCity 2017.1)</p> <ul style="list-style-type: none"><li>■ Desktop - closed-source edition bundled with Windows, available only on Windows platforms.</li><li>■ Core - open-source edition based on .Net Core, cross-platform, 64-bit only</li></ul>
Working directory		Specify the path to the <a href="#">build working directory</a> .



To fail a build if "an error message is logged by build runner" (see [Build Failure Conditions](#)), change the default setting of the Error Output selector from warning to error.

Script		Select whether you want to enter the script right in TeamCity, or specify a path to the script: <ul style="list-style-type: none"> <li>File: Enter the path to a PowerShell file. The path has to be relative to the checkout directory.</li> <li>Source: Enter the PowerShell script source. Note that TeamCity <a href="#">parameter references</a> will be replaced in the code.</li> </ul> <p> There is an issue with PowerShell 2.0 not returning the correct exit code when the script contains explicitly defined parameters. As a <a href="#">workaround</a>, either upgrade your PowerShell or to use <code>[Environment]::Exit(code)</code>.</p>
Script execution mode		Specify the PowerShell script execution mode. By default, PowerShell may not allow execution of arbitrary .ps1 files. TeamCity will try to supply the <code>-ExecutionPolicy Bypass</code> argument. If you've selected Execute .ps1 script from external file, your script should be signed or you should make PowerShell allow execution of arbitrary .ps1 files. If the execution policy doesn't allow running your scripts, select Put script into PowerShell stdin mode to avoid this issue.  The <code>-Command</code> mode is deprecated and is not recommended for use with PowerShell of version greater than 1.0
Script arguments		Available if "Script execution mode" option is set to "Execute .ps1 script from external file". Specify here arguments to be passed into PowerShell script. <p> Escaping special symbols TeamCity calls powershell.exe from the console of your operating system (command prompt on Windows, bash or other on Linux).  If parameters containing special symbols are passed to your PowerShell script in double quotes, make sure these characters are properly escaped: use the escape rules depending on your interpreter, e.g. on Windows, if a PowerShell script argument ends with a backslash, use the backslash as the escape symbol for TeamCity to correctly interpret it: <code>"f oo\"</code></p>
Additional command line parameters		Specify parameters to be passed to powershell.exe.

## Docker Settings

To enable support for Docker in PowerShell steps, run the TeamCity server with the `-Dteamcity.docker.runners=jetbrains_powershell internal` property.

In this section, you can specify a Docker image which will be used to run the build step.  
Current limitations

- Execution under Docker requires the PowerShell executable to be added to PATH
- When using Docker to run the build step, only Docker-related build agent requirements are applied to the build
- Selection of Edition in PowerShell build step affects the executable being used (powershell.exe for Desktop, pwsh for Core).
- <Auto> defaults to pwsh (Core)
- To specify a custom PowerShell executable, the `teamcity.powershell.virtual.executable` configuration parameter must be set to the full path of this executable inside the provided image
- Current limitations of the Docker wrapper do not allow Linux containers running under Windows systems

## Interaction with TeamCity

Attention must be paid when using PowerShell to interact with TeamCity through service messages. PowerShell tends to wrap strings written to the console with commands like `Write-Output`, `Write-Error` and similar (see [TW-15080](#)). To avoid this behavior, either use the `Write-Host` command, or adjust the buffer length manually:

```

function Set-PSConsole {
    if (Test-Path env:TEAMCITY_VERSION) {
        try {
            $rawUI = (Get-Host).UI.RawUI
            $m = $rawUI.MaxPhysicalWindowSize.Width
            $rawUI.BufferSize = New-Object Management.Automation.Host.Size ([Math]::max($m, 500),
$rawUI.BufferSize.Height)
            $rawUI.WindowSize = New-Object Management.Automation.Host.Size ($m,
$rawUI.WindowSize.Height)
        } catch {}
    }
}

```

## Error Handling

Due to [this issue](#) in PowerShell itself that causes zero exit code to be always returned to a caller, TeamCity cannot always detect whether the script has executed correctly or not. We recommend several approaches that can help in detecting script execution failures:

- Manually catching exceptions and explicitly returning exit code  
The PowerShell plugin does not use the cmd wrapper around `powershell.exe`. It makes returning the explicit exit code possible.

```

try {
    # your code here
} Catch {
    $ErrorMessage = $_.Exception.Message
    Write-Output $ErrorMessage
    exit(1)
}

```

- Setting Error Output to `Error` and adding a build failure condition  
In case syntax errors and exceptions are present, PowerShell writes them to `stderr`. To make TeamCity fail the build, set Error Output option to `Error` and add a [build failure condition](#) that will fail the build on any error output.
- Failing build on certain message in build log  
Add a [build failure condition](#) that will fail the build on a certain message (say "POWERSHELL ERROR") in the build log.

```

$ErrorMessage = "POWERSHELL ERROR"
try {
    # your code here
} Catch {
    Write-Output $ErrorMessage
    exit(1)
}

```

## Handling Output

To properly handle non-ASCII output from PowerShell, the correct encoding must be set both on the PowerShell side and on the TeamCity side.

- To set the output encoding for PowerShell to UTF-8, add the following line to the beginning of your PowerShell script:

```
[Console]::OutputEncoding = [System.Text.Encoding]::UTF8
```

- To set the encoding on the TeamCity agent side, either set the java launch option `-Dfile.encoding=UTF-8`, or set the build configuration parameter `teamcity.runner.commandline.stdstreams.encoding` value to `UTF-8`

## Temporary Files

The TeamCity PowerShell plugin uses temporary files as an entry point; these files are created in the build temporary folder and removed after the PowerShell build step is finished. To keep the files, set the `powershell.keep.generated` or `teamcity.dont.delete.temp.files` configuration parameter to `true`.

## Development Links

The PowerShell support is implemented as an open-source plugin. For development links refer to the [plugin's page](#).

### Rake

 TeamCity Rake runner supports the `Test::Unit`, `Test-Spec`, `Shoulda`, `RSpec`, `Cucumber` test frameworks. It is compatible with Ruby interpreters installed using Ruby Version Manager (MRI Ruby, JRuby, IronRuby, REE, MacRuby, etc.) with `rake 0.7.3` gem or higher.

## In this section:

- [Prerequisites](#)
- [Important Notes](#)
- [Rake Runner Settings](#)
  - [Rake Parameters](#)
  - [Ruby Interpreter](#)
  - [Launching Parameters](#)
  - [Tests Reporting](#)
- [Known Issues](#)
- [Additional Runner Options](#)
- [Development Links](#)

## Prerequisites

Make sure to have Ruby interpreter (MRI Ruby, JRuby, IronRuby, REE, MacRuby, or etc) with `rake 0.7.3` gem or higher (mandatory) and all necessary gems for your Ruby (or ROR) projects and testing frameworks installed on at least one build agent.

You can install several Ruby interpreters in different folders. On Linux/MacOS it is easier to configure using [RVM](#) or [rbenv](#). It is possible to install Ruby interpreter and necessary Ruby gems using the [Command Line](#) build runner step. If you want to automatically configure agent requirements for this interpreters, you need to register its paths in the build agent configuration properties and then refer to such property name in the [Rake build runner configuration](#).

To install a gem, execute:

```
gem install <gem's name>
```

You can refer to the [Ruby Gems Manuals](#) for more information.

Instead of the `gem` command, you can install gems using the `Bundler` gem.

 If you use Ruby 1.9 for `Shoulda`, `Test-Spec` and `Test::Unit` frameworks to operate, the '`test-unit`' gem must be installed.

 To use the `minitest` framework, '`minitest-reporters`' gem must be installed. See details in the [RubyMine webhelp](#).

## Important Notes

- Ruby's pending specs are shown as Ignored Tests in the Overview tab.
- Rake Runner uses its own unit tests runner and loads it using the `RUBYLIB` environment variable. You need to ensure your program doesn't clear this environment variable, but you may append your paths to it.
- If you run RSpec with the `--color` option enabled under Windows OS, RSpec will suggest you install the `win32console` gem. This warning will appear in your build log, but you can ignore it. TeamCity Rake Runner doesn't support coloured output in the build log and doesn't use this feature.
- Rake Runner runs spec examples with a custom formatter. If you use additional console formatter, your build log will contain redundant information.

- `Spec::Rake::SpecTask.spec_opts` of your Rakefile is affected by `SPEC_OPTS` command line parameter. Rake Runner always uses `SPEC_OPTS` to set up its custom formatter. Thus you should set up Spec Options in Web UI. The same limitation exists for Cucumber tests options.
- To include HTML reports into the Build Results, you can add the corresponding [report tab](#) for them.

## Rake Runner Settings

### Rake Parameters

Option	Description
Path to a Rakefile file	Enter a Rakefile path if you don't want to use the default one. The specified path should be relative to the <a href="#">Build Checkout Directory</a> .
Rakefile content	Type in the Rakefile content instead of using the existing Rakefile. The new Rakefile will be created dynamically from the specified content before running Rake.
Working directory	Optional. Specify if differs from the <a href="#">Build Checkout Directory</a> .
Rake tasks	Enter space-separated tasks names if you don't want to use the 'default' task. For example, 'test:functionals' or 'mytask:test mytask:test2'.
Additional Rake command line parameters	Specified parameters will be added to 'rake' command line. The command line will have the following format:  <div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin-left: auto; margin-right: auto;">           ruby rake &lt;Additional Rake command line parameters&gt; &lt;TeamCity Rake Runner options, e.g TESTOPTS&gt; &lt;tasks&gt;         </div>

### Ruby Interpreter

Option	Description
Use default Ruby	Use Ruby interpreter settings defined in the <a href="#">Ruby environment configurator</a> build feature settings or the interpreter will be searched in the <code>PATH</code> .
Ruby interpreter path	The path to Ruby interpreter. The path cannot be empty. This field supports values of environment and system variables. For example:  <div style="border: 1px solid #ccc; padding: 10px; width: fit-content; margin-left: auto; margin-right: auto;">           %env.I_AM_DEFINED_IN_BUILDAGENT_CONFIGURATION%         </div>
RVM interpreter	Specify here the RVM interpreter name and optionally a gemset configured on a build agent. Note, the interpreter name cannot be empty. If gemset isn't specified, the default one will be used.

### Launching Parameters

Option	Description
Bundler: bundle exec	If your project uses the <a href="#">Bundler requirements</a> manager and your Rakefile doesn't load the bundler setup script, this option will allow you to launch rake tasks using the 'bundle exec' command emulation. If you want to execute 'bundle install' command, you need to do it in the <a href="#">Command Line</a> step before the Rake runner step. Also, remember to set up the <a href="#">Ruby environment configurator</a> build feature to automatically pass Ruby interpreter to the command line runner.
Debug	Check the Track invoke/execute stages option to enable showing Invoke stage data in the build log.

### Tests Reporting

Option	Description

<p>Attached reporters</p> <p>If you want TeamCity to display the test results on a dedicated <a href="#">Tests tab</a> of the Build Results page, select here the testing framework you use: Test::Unit, Test-Spec, Shoulda, RSpec or Cucumber.</p>	 If you're using RSpec or Cucumber, make sure to specify here the user options defined in your build script, otherwise they will be ignored.
---	---

## Known Issues

- If your Rake tasks or tests run in parallel in the scope of one build, the build output and tests results will be inaccurate.
- If you are using RVM, it is recommended to start TeamCity agent when the current rvm sdk isn't set or to invoke the "rvm system" at first.

## Additional Runner Options

These options can be configured using system properties in the [Build Parameters](#) section.

Option	Description
system.teamcity.rake.runner.gem.rake.version	Allows to specify which rake gem to use for launching a rake build.
system.teamcity.rake.runner.gem.testunit.version	If your application uses the test-unit gem version other than the latest installed (in Ruby sdk), specify it here. Otherwise the Test::Unit test reporter may try to load the incorrect gem version and affect the runtime behavior. If the test-unit gem is installed but your application uses Test::Unit bundled in Ruby 1.8.x SDK, set the version value to 'built-in'.
system.teamcity.rake.runner.gem.bundler.version	Launches bundler emulation for the specified bundler gem version (the gem should be already installed on an agent).
system.teamcity.rake.runner.custom.gemfile	Customizes Gemfile if it isn't located in the checkout directory root.
system.teamcity.rake.runner.custom.bundle.path	Sets BUNDLE_PATH if TeamCity doesn't fetch it correctly from <Gemfile containing directory>/bundle/config.

## Development Links

Rake support is implemented as an open-source plugin. For development links refer to the [plugin's page](#).

## Simple Build Tool (Scala)

The Simple Build Tool (Scala) runner natively supports [SBT](#) builds: you can build your code, run tests and see the results in a handy way in TeamCity. The supported SBT version 0.13.+.  
The runner, formerly provided as a [standalone plugin](#), is bundled since TeamCity 9.1.

### SBT runner settings

#### SBT parameters

Option	Description
SBT commands	Commands to execute, e.g. clean "set scalaVersion:="2.11.6"" compile test or ;clean;set scalaVersion:="2.11.6";compile;test.
SBT installation mode	When the default <Auto> option is selected, the latest SBT version will be installed on every TeamCity agent your build will be running. To specify an existing installation, use the <Custom> mode. The sbt-launch.jar from the \bin directory of the SBT home will be launched.
SBT home path	Available if <Custom> is selected in the option above. The path to the existing SBT installation directory.
Working directory	Optional. Specify the <a href="#">build working directory</a> if it differs from the <a href="#">build checkout directory</a> .

## Java Parameters

Option	Description
JDK	When <Default> is selected, the JDK specified in the JAVA_HOME environment variable on the agent or the agent's own Java is used to run the build process. Set to <Custom> to use a custom JDK.
JDK home path	Available if <Custom> is selected in the option above. Specify the path to your custom JDK which will be used to run the build.
JVM command line parameters	Specify the desired Java Virtual Machine parameters, such as maximum heap size or parameters that enable remote debugging. These settings are passed to the JVM used to run your build. Example:  -Xmx512m -Xms256m

## Visual Studio 2003

Visual Studio 2003 Build Runner supports building Microsoft Visual Studio 2003 .NET projects. To build Microsoft Visual Studio 2005-2017 projects, see [Visual Studio \(sln\) Build Runner](#).



- The Visual Studio 2003 build runner uses NAnt instead of MS Visual Studio 2003 to perform the build. As a result the agent is required to have .NET Framework 1.1 installed, however under certain conditions .NET Framework SDK 1.1 might be required. This NAnt solution task may behave differently than MS Visual Studio 2003. See <http://nant.sourceforge.net/release/latest/help/tasks/solution.html> for details.
- To use this runner you need to configure the [NAnt runner](#).

Option	Description
Solution file path	A path to the solution to be built is relative to the <a href="#">build checkout directory</a> . For example:  vs-addin\addin\addin.sln
Working directory	Specify the <a href="#">Build Working Directory</a> .
Configuration	Specify the name of the solution configuration to build.
Projects output	This group of options enables you to use the default output defined in the solution, or specify your own output path.
Output directory for all projects	This option is available, if Override project output option is checked. Specify the directory where the compiled targets will be placed.
Resolve URLs via map	Click this radio button, if you want to map the URL project path to the physical project path. If this option is selected, specify mapping in the Type the URL's map field.
Type the URL's map	Click this link and specify the desired map in the text area. Use the following format:  http://localhost:8111=myProjectPath/myProject  where <ul style="list-style-type: none"> <li>http://localhost:8111 is a host where the project will be uploaded</li> <li>myProjectPath/myProject is the project root</li> </ul>

Resolve URLs via WebDAV	<p>Click this radio button, if you want the URLs to be resolved via WebDav.</p> <div style="border: 2px solid red; padding: 5px;"> <p> Make sure that all the necessary files are properly updated. The build agent may not update information from VCS implicitly.</p> </div>
MS Visual Studio reference path	Check this option, if you want to automatically include reference path of MS Visual Studio to the build.
NAnt home	Specify path to the NAnt executable to run builds of the build configuration. The path can be absolute, relative to the build checkout directory; also you can use an environment variable.
Command line parameters	<p>Specify any additional parameters for <code>NAnt.exe</code></p> <div style="border: 2px solid green; padding: 5px;"> <p> TeamCity passes automatically to NAnt all defined <a href="#">system properties</a>, so you do not need to specify all of the properties here via '-D' option. You can create necessary properties at the Build Parameters section of the build configuration settings.</p> </div>
Run NUnit tests for	<p>Specify .Net assemblies, where the NUnit tests to be run are stored. Multiple entries are comma-separated; usage of NAnt wildcards is enabled. In the following example, TeamCity will search for the tests assemblies in all project directories and run these tests.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>**\*.dll</pre> </div> <div style="border: 2px solid lightblue; padding: 5px; margin-top: 10px;"> <p> All these wildcards are specified relative to path that contains the solution file.</p> </div>
Do not run NUnit tests for	<p>Specify .NET assemblies that should be excluded from the list of found assemblies to test. Multiple entries are comma-separated; usage of NAnt wildcards is enabled. In the following example, TeamCity will omit tests specified in this directory.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>**\obj\**\*.dll</pre> </div> <div style="border: 2px solid lightblue; padding: 5px; margin-top: 10px;"> <p> All these wildcards are specified relative to path that contains the solution file.</p> </div>
Reduce test failure feedback time	Use following option to instruct TeamCity to run some tests before others.
Run recently failed tests first	If checked, in the first place TeamCity will run tests failed in previous finished or running builds as well as tests having high failure rate (a so called blinking tests)

See also:

[Administrator's Guide: NUnit for MSBuild](#)

Visual Studio (.sln)

This page contains reference information for the Visual Studio (.sln) Build Runner that builds Microsoft Visual Studio 2005-2015, and since TeamCity 10.0.4, Microsoft Visual Studio 2017 solution files.

To build Microsoft Visual Studio 2003 solution files, use the [Visual Studio 2003](#) runner.

 The Visual Studio (.sln) build runner requires the proper version of Microsoft Visual Studio installed on the build agent.

## General Build Runner Options

Option	Description
Solution file path	Specify the path to the solution to be built relative to the <a href="#">Build Checkout Directory</a> . For example: vs-addin\addin\addin.sln
Working directory	Specify the <a href="#">Build Working Directory</a> . (optional)
Visual Studio	Select the Visual Studio version.
Targets	Specify the Microsoft Visual Studio targets specific for the previously selected Visual Studio version. The possible options are Build, Rebuild, Clean, Publish or a combination of these targets based on your needs. Multiple targets are space-separated.
Configuration	Specify the name of Microsoft Visual Studio solution configuration to build (optional).
Platform	Specify the platform for the solution. You can leave this field blank, and TeamCity will obtain this information from the solution settings (optional).
Command line parameters	Specify additional command line parameters to be passed to the build runner. Instead of explicitly specifying these parameters, it is recommended to define them on the <a href="#">Parameters</a> page.

See also:

[Troubleshooting: Visual Studio logging](#)

## Visual Studio Tests

The Visual Studio Tests runner integrates [MSTest](#) runner and [VSTest console runner](#). Support for both frameworks enables TeamCity to execute tests and automatically import their test results.

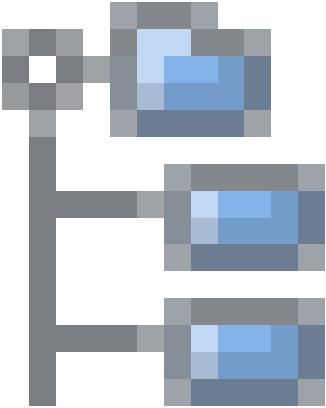
The Visual Studio Test Runner requires Visual Studio Test Agent or Microsoft Visual Studio installed on the build agent.

On this page:

- [Visual Studio Tests runner settings](#)
  - [VSTest Settings](#)
    - [Custom test logger](#)
    - [MSTest settings](#)

### Visual Studio Tests runner settings

Option	Description
Test engine type	Select the tool used to run tests: VSTest or MSTest.
Test engine version	Select the version of the tool from the drop-down. By default, the available VSTest and MSTest installations are autodetected by TeamCity: <ul style="list-style-type: none"><li>■ MSTest versions 2005-2017 are supported.</li><li>■ VSTest versions 2012-2017 are supported.</li></ul> You can specify a custom path to the test runner here as well. TeamCity parameters are supported. <p> You can specify the agent property <code>teamcity.dotnet.vstest.VS_VERSION.install.dir</code> pointing to a VSTest distribution path and use it here, where <code>VS_VERSION</code> can be 11.0, 12.0, 14.0, or 15.0. The property will have priority over registry values.</p>

Test file names	This field is mandatory for VSTest and optional for MSTest. Specify the new-line separated list of paths to assemblies to run tests on in the included assemblies list. Exclude assemblies from the test run by specifying paths to them in the corresponding field. <a href="#">Wildcards</a> are supported.  Paths to the assemblies must be relative to the <a href="#">build checkout directory</a> .
Run configuration file	(Optional) Specify the typical <code>.runsettings</code> file for <a href="#">VSTest</a> or <code>.testsettings</code> file for <a href="#">MSTest</a> . You can use 
Additional command line parameters	Enter additional command line parameters for the selected test engine.  Note that tests run with MSTest are not reported on-the-fly. The Microsoft Developer Network lists the available options for <a href="#">VSTest</a> and <a href="#">MSTest</a> .

The rest of settings will vary depending on the engine to run tests with:

- [VSTest Settings](#)
- [MSTest settings](#)

#### VSTest Settings

Option	Description
Target platform	Select the platform bitness. Note that specifying x64 target platform will force the <code>vstest.console</code> process to be run in the isolated mode
Framework	If the default is specified, <code>vstest.console</code> will select the target framework automatically. You can also choose the .Net platform manually using the drop-down.
Test names	(optional) Of all tests discovered in the included assemblies, only the tests with the names matching the provided values will be run. For multiple values, use new line.   If the field is empty, all tests will be run. See details in the <a href="#">Microsoft documentation</a> .
Test case filter	Run tests that match the given expression. See details in the <a href="#">Microsoft documentation</a> .  Cannot be used with the option above.
Run in isolation	Run the tests in an isolated process

<p><b>Use real-time test reporting</b></p> <ul style="list-style-type: none"> <li>• If this checkbox is selected, a custom TeamCity test logger will be used for real-time reporting. See the <a href="#">next section</a> for details.</li> <li>• If not selected, VSTest will report tests to TeamCity after the run (the discovery of custom logger will not be attempted).</li> </ul> <div style="border: 1px solid #f0e68c; padding: 5px; margin-top: 10px;"> <p> When using this option, it is recommended that you check the number of tests in the produced <code>.trx</code> file against the build log. In case of inconsistencies, switch to the custom logger.</p> </div>
---

## Custom test logger

VSTest.Console supports custom loggers, i.e. libraries that can handle events that occur when tests are being executed. TeamCity 9.0+ has a custom logger that provides real-time test reporting.

The logger must be installed manually on the agent machine, as it requires dlls to be copied to the `Extensions` folder of the VSTest.Console. No agent restart is needed when the custom logger is installed.

To install the custom logger:

- a. Download the [custom logger](#)
- b. Extract the contents of the downloaded archive on the agent machine:
  - for VisualStudio 2017 update 5 onwards - to `PROGRAM_FILES(x86)\Microsoft Visual Studio\2017<Edition>\Common7\IDE\Extensions\TestPlatform\Extensions`
  - for VisualStudio 2017 up to update 4 - to `PROGRAM_FILES(x86)\Microsoft Visual Studio\2017<Edition>\Common7\IDE\CommonExtensions\Microsoft\TestWindow\Extensions`
  - for VisualStudio 2015 - to `PROGRAM_FILES\Microsoft Visual Studio 14.0\Common7\IDE\CommonExtensions\Microsoft\TestWindow\Extensions`
  - for VisualStudio 2013 - to `PROGRAM_FILES\Microsoft Visual Studio 12.0\Common7\IDE\CommonExtensions\Microsoft\TestWindow\Extensions`
  - for VisualStudio 2012 - to `PROGRAM_FILES\Microsoft Visual Studio 11.0\Common7\IDE\CommonExtensions\Microsoft\TestWindow\Extensions`
- Check that the custom logger was installed correctly by executing `vstest.console.exe /ListLoggers` in the console on the agent machine. If the logger was installed correctly, you will see the logger with FriendlyName `TeamCity` listed:

```
VSTest.TeaмCityLogger.TeaмCityLogger
Uri: logger://TeамCityLogger
FriendlyName: TeamCity{info}
```

## MSTest settings

Option	Description
MSTest metadata	Enter a value for <code>/testmetadata:file</code> argument. See details in the <a href="#">Microsoft documentation</a>
Testlist from metadata to run	Edit the Testlist. Every line will be translated into <code>/testlist:line</code> argument. See details in the <a href="#">Microsoft documentation</a> .
Test	Names of individual tests to run. This option will be translated into the series of <code>/test:</code> arguments. See details in the <a href="#">Microsoft documentation</a> .
Unique	Run the test only if one unique match is found for any specified test in test section

Results file	<p>Enter a value for the <code>/resultsfile:file name</code> command-line argument. Parameter references are supported. It is recommended to leave the field blank to generate a temporary <code>*.trx</code> file in the build temporary directory. Note that the directory may be cleaned between the builds.</p> <p>To save the test run results to a named non-default file, enter a value for the <code>/resultsfile:file name</code> command-line argument.</p> <ul style="list-style-type: none"> <li>If the path specified is relative, it will be relative to the build checkout directory. If the specified file already exists in the checkout directory, the build agent will attempt to delete the file. If the agent fails to delete it, the build will fail.</li> </ul> <div style="border: 1px solid #f0e68c; padding: 10px; margin-top: 10px;">  If you need to generate your results file in the checkout directory, consider adding the <a href="#">Swabri</a> build feature to your configuration.     </div> <ul style="list-style-type: none"> <li>To create a <code>*.trx</code> report in the build temporary directory, use <code>%system.teamcity.build.tempDir%</code>.</li> <li>If the path specified is absolute, it will be used "as is".</li> </ul>
--------------	---

## Xcode Project

The Xcode Project Build Runner supports Xcode 3 (target-based build), Xcode 4 (scheme-based build), Xcode 5-9.

The runner provides structured build log based on Xcode build stages, detects compilation errors, reports tests from the `xcodebuild` utility, adds automatic agent requirements for the appropriate version of tools installed (Xcode, SDKs, etc.) and reporting tools via agent properties.

 To run an Xcode build, you need to have one or more build agents running Mac OS X with installed Xcode.

### Working with different Xcode setups:

- If only one Xcode version is installed on the agent machine, it will be used by default. The agent restart is required if Xcode was installed/updated.
  - If several Xcode versions are installed, perform one of the following:
    - specify the path to the required version in the [Path to Xcode](#) of the Xcode Project build step settings
    - select the default XCode distribution using the `xcode-select` tool (details on [Apple.com](#)).
- The path to Xcode: `/Applications/Xcode.app/Contents/Developer`  
 Command to switch: `sudo xcode-select -s path_to_xcode_distribution`

 Note that if you use `xcode-select`, the agent must be restarted, as it only detects Xcode distributions and reports them to the server during startup.

## Xcode Project Runner Settings

Setting	Build	Description
Path to the project or workspace		The path to a ( <code>.xcodeproj</code> ) project file or a ( <code>.xcworkspace</code> ) workspace file, should be relative to the checkout directory. For Xcode 3 build, only the path to the project file is supported.
Working directory		Specify the <a href="#">build working directory</a> .
Path to Xcode		Specify the path to Xcode on the agent. The build will be run using this Xcode.

Build		Select either a target-based (for project) or scheme-based (for project and workspace) build. Depending on the selection, the settings displayed will vary.
Scheme	Scheme-based	Xcode scheme to build. The list of available schemes is formed by parsing your project/workspace files in the VCS. Make sure your Path to the project or workspace is set correctly and click the Check/Reparse Project button to show/refresh the schemes list. Note that a scheme must be shared to be shown in the list (to check if your scheme is shared, verify that it is located under the <code>xcshareddata</code> folder and not under the <code>xcuserdata</code> one, and that the <code>xcshareddata</code> folder is committed to your VCS. To check the latter, use the VCS tree popup next to the Path to the project or workspace field). More information on managing Xcode schemes is available in the <a href="#">Apple documentation</a> .
Build output directory	Scheme-based	Check the Use custom box to override the default path for the files produced by your build. Specify the custom path relative to the checkout directory.
Target	Target-based	Xcode target to execute. The list of available targets is formed by parsing your project files in the VCS. Make sure your Path to the project is set correctly and click the Check/Reparse Project button to show/refresh the targets list.
Configuration	Target-based	Xcode configuration. The list of available configurations is formed by parsing your project files in the VCS. Since the configuration depends on the target, you must choose the target first. Make sure your Path to the project is set correctly and click the Check/Reparse Project button to show/refresh the configurations list.
Platform	Target-based	Select from the default, iOS, Mac OS X or Simulator - iOS or any other platform (if it is provided by the agent) to build your project on.
SDK	Target-based	You can choose a SDK to build your project with (the list of available SDKs is formed according to the SDKs available on your agents for the platform selected).
Architecture	Target-based	You can choose an architecture to build your project with (the list of available architectures is formed according to the architectures available on your agents for the platform selected).
Build action(s)		<p>Xcode build action(s). The default actions are <code>clean</code> and <code>build</code>. A space-separated list of the following actions is supported: <code>clean</code>, <code>build</code>, <code>test</code>, <code>archive</code>, <code>installsrc</code>, <code>install</code>; the order of actions will be preserved during execution.</p> <p> It is not recommended to change this field unless you are sure you want to change the number or order of actions.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <p> If your project is built under Xcode 5+, checking the Run test option below automatically adds the <code>test</code> build action to the list (unless the option is already explicitly specified in the current field).</p> </div>
Run tests		Check this option if want to run tests after you project is built.
Additional command line parameters		Other command line parameters to be passed to the <code>xcodebuild</code> utility.

See also:

[Concepts: Build Runner](#)

Deployers

Deployer build runners enable TeamCity to upload artifacts to external locations in a number of ways.

The following build runners are available:

- [SMB Upload](#)
- [FTP Upload](#)
- [SSH Upload](#)
- [Container Deployer](#)
- [SSH Exec](#)

Deployer build runners are implemented as a plugin bundled since TeamCity 10.0. For earlier TeamCity versions use the [stand](#)

-alone plugin.

## SMB Upload

SMB upload enables TeamCity to upload files/directories to Windows shares via Server Message Block (SMB) protocol.

The settings common for all runners are described [on a separate page](#); this page details the SMB Upload settings.

 The fields below support [parameter references](#): any text between percentage signs (%) is considered a reference to a property by TeamCity. To prevent TeamCity from treating the text in the percentage signs as a property reference, use two percentage signs to escape them: e.g. if you want to pass "%Y%m%d%H%M%S" into the build, change it to "%%Y%%m%%d%%H%%M%%S"

Option	Description
Deployment Target	
Target URL	<p>The URL should point to a host + share at least. Subdirectories are allowed here and will be created if missing. Valid examples:</p> <div style="border: 1px solid #ccc; padding: 10px;"><pre>\\\host\share_name \\host\share_name\some\path \\host\c\$\Temp</pre></div>
Name resolution	
	<p>The DNS only name resolution allows switching JCIFS to "DNS-only" mode. May fix performance or out of memory exceptions (see <a href="#">this bitbucket issue</a> for details). Is equivalent to following JCIFS settings:</p> <div style="border: 1px solid #ccc; padding: 10px;"><pre>-Djcifs.resolveOrder=DNS -Djcifs.smb.client.dfs.disabled=true</pre></div>
Deployment Credentials	
Username	Specify the username
Password	Specify the password
Domain	Specify the domain
Deployment Source	
Path to sources	Specify the deployment sources as a newline- or comma-separated list of paths to files/directories for deployment. Ant-style wildcards like dir/**/*.zip and target directories like *.zip => winFiles,unix/distro.tgz => linuxFiles, where winFiles and linuxFiles are target directories, are supported.

## FTP Upload

FTP Upload allows deploying files/directories to an FTP server.

The settings common for all runners are described on a separate page; this page details the FTP Upload settings.

**i** The fields below support [parameter references](#): any text between percentage signs (%) is considered a reference to a property by TeamCity. To prevent TeamCity from treating the text in the percentage signs as a property reference, use two percentage signs to escape them: e.g. if you want to pass "%Y%m%d%H%M%S" into the build, change it to "%%Y%%m%%d%%H%%M%%S"

Option	Description
Deployment Target	
Target host	Specify an FTP server (usehostname or IP address) and a remote directory (relative to the FTP user's home). To use an absolute *nix path, use %2F as the forward slash. For example:  <pre>ftp://hostname.com/ hostname.com:34445/subdir 127.0.0.1/%2Fetc/</pre>
Secure connection	Choose between an insecure (FTP) and a secure connection (FTPS, FTPES).
Deployment Credentials	
Authentication method	Select either Anonymous (will submit username "anonymous" and a single space as the password) or username/password (for custom credentials)
FTP modes	
FTP Mode	Select the passive or active mode
Transfer Mode	Optional. Select an FTP transfer mode to force: the ASCII or Binary FTP transfer modes (if the automatically detected mode leads to broken files transfer)
Deployment Source	
Paths to sources	Specify the deployment sources as a newline- or comma-separated list of paths to files/directories to be deployed. Ant-style wildcards like dir/**/*.zip and target directories like *.zip => winFiles,unix/distr o.tgz => linuxFiles, where winFiles and linuxFiles are target directories, are supported.

## SSH Upload

SSH Upload allows uploading files/directories via SSH (using SCP or SFTP protocols).

The settings common for all runners are described on a separate page; this page details the SSH Deployer settings.

 The fields below support [parameter references](#): any text between percentage signs (%) is considered a reference to a property by TeamCity. To prevent TeamCity from treating the text in the percentage signs as reference to a property, use two percentage signs to escape them: e.g. if you want to pass "%Y%m%d%H%M%S" into the build, change it to "%%Y%%m%%d%%H%%M%%S"

Option	Description
Deployment Target	
Target	<p>Target should point to an SSH server location. The syntax is similar to the one used by the *nix scp command:</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"><code>{hostname IP_address}[:target_dir[/sub_path]]</code></div> <p>where target_dir can be absolute or relative; sub_path can have any depth.</p>
Transport protocol:	Select a protocol to transfer data over SSH. The available options are: SCP and SFTP
Port	Optional. By default, port 22 is used.
Deployment C redentials	The settings in this section will vary depending on the selected authentication method.
Authentication method	<p>Select an authentication method.</p> <ul style="list-style-type: none"><li>Uploaded key - uses the key(s) uploaded to the project. See <a href="#">SSH Keys Management</a> for details.</li><li>Default private key will try to perform private key authentication using the <code>~/.ssh/config</code> settings. If no settings file exists, will try to use the <code>~/.ssh/rsa_pub</code> public key file. No passphrases should be set.</li><li>Custom private key will try to perform private key authentication using the given public key file with given passphrase</li><li>Password - simple password authentication.</li><li>SSH-Agent - use ssh-agent for authentication, the <a href="#">SSH-Agent build feature</a> must be enabled.</li></ul> <div style="border: 1px solid #fca; padding: 10px; margin-top: 10px;"><p> <b>Limited security note</b> The current secure connection implementation accepts any certificate provided by the remote host. No trust checks are performed!</p></div>
Deployment Source	
Paths to sources:	Specify the deployment sources as a newline- or comma-separated list of paths to file/directories. Ant-style wildcards like <code>dir/**/*.*zip</code> and target directories like <code>*.zip =&gt; winFiles,unix/distro.tgz =&gt; linuxFiles</code> , where <code>winFiles</code> and <code>linuxFiles</code> are target directories, are supported.

## Container Deployer

The Container Deployer build runner allows deploying WAR application archives to different containers. The runner supports the following Tomcat versions: 5.x, 6.x, 7.x and 8.x.

The settings common for all runners are described on a separate page; this page details the Container Deployer settings.

**i** The fields below support [parameter references](#): any text between percentage signs (%) is considered a reference to a property by TeamCity. To prevent TeamCity from treating the text in the percentage signs as a property reference, use two percentage signs to escape them: e.g. if you want to pass "%Y%m%d%H%M%S" into the build, change it to "%%Y%%m%%d%%H%%M%%S"

Option	Description
Deployment Target	
Target	Enter target container info in the following format: {hostname IP}[:port]
Container Type:	Select the version of Tomcat from the dropdown. The default "Manager" web app must be deployed to the target Tomcat. The user must have role "manager-script".
Deployment Credentials	
Username	Specify the username. The user must have "manager-script" role assigned.
Password	Specify the password. Configuration parameters can be used
Deployment Source	
Path to war archive	Specify the source war archive to be deployed.

## SSH Exec

SSH Exec enables TeamCity to execute arbitrary remote commands using SSH.

The settings common for all runners are described on a separate page; this page details the SSH Exec runner settings.

**i** The fields below support [parameter references](#): any text between percentage signs (%) is considered a reference to a property by TeamCity. To prevent TeamCity from treating the text in the percentage signs as reference to a property, use two percentage signs to escape them: e.g. if you want to pass "%Y%m%d%H%M%S" into the build, change it to "%%Y%%m%%d%%H%%M%%S"

Option	Description
Deployment Target	
Target	Target should point to an SSH server location. Enter hostname or IP address.
Port	Optional. By default, port 22 is used.
Use pty	Optional. By default, a pty will not be allocated
Deployment Credentials	The settings in this section will vary depending on the selected authentication method.

Authentication method	<p>Select an SSH authentication method.</p> <ul style="list-style-type: none"> <li>• Select an authentication method.           <ul style="list-style-type: none"> <li>• Uploaded key uses the key(s) uploaded to the project. See <a href="#">SSH Keys Management</a> for details.</li> <li>• Default private key will try to perform private key authentication using the <code>~/.ssh/config</code> settings. If no settings file exists, will try to use the <code>~/.ssh/rsa_pub</code> public key file. No passphrases should be set.</li> <li>• Custom private key will try to perform private key authentication using the given public key file with given passphrase</li> <li>• Password - simple password authentication.</li> <li>• SSH-Agent - use ssh-agent for authentication, the <a href="#">SSH-Agent build feature</a> must be enabled.</li> </ul> </li> </ul> <div style="border: 1px solid #f0c987; padding: 5px; margin-top: 10px;">  <b>Limited security note</b>            The current secure connection implementation accepts any certificate provided by the remote host.            No trust checks are performed!         </div>
SSH Commands	
Commands:	Specify a new-line delimited set of commands that will be executed in the remote shell. The remote shell will be started in the home directory of an authenticated user. The shell output will be available in the TeamCity build log.

## Docker

TeamCity comes with built-in Docker integration, which includes the Docker runner (formerly Docker Build), a runner for Docker commands.

	<b>Requirements</b> The integration requires Docker installed on the build agents. Docker Compose also needs to be installed to use the <a href="#">Docker Compose</a> build runner.
---	---

## Supported Environments

TeamCity-Docker support can run on Mac, Linux, and Windows build agents. It uses the `'docker'` executable on the build agent machine, so it should be runnable by the build agent user.



- On Linux, the integration will run if the installed Docker is detected.
- On Windows, the integration works in the Windows container mode only. Docker on Windows with the Linux container mode enabled is not supported, an [error](#) is reported in this case.
- On macOS, the official [Docker support for Mac](#) should be installed for the user running the build agent.

The Docker runner supports the `build`, `push`, `tag` Docker commands.

When creating TeamCity projects/ build configurations from a repository URL, the runner is offered as build step during auto-detection, provided a Dockerfile is present in the VCS repository.

Setting		Description
Docker Command		Select the docker command. Depending on the selected command, the settings below will vary.
Docker Command Parameters		
build	Dockerfile source	Depending on the selected source, the settings below will vary. The available options include File, a URL or File content.
	Path to file	Available if File is selected as the source. Specify the path to the Docker file. The path should be relative to the <a href="#">checkout directory</a> .
	Context folder	Available if File is selected as the source. Specify the context for the docker build. If blank, the enclosing folder for Dockerfile will be used.
	URL to file	Available if URL is selected as the source. The URL can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts, and plain text files. See <a href="#">Docker documentation</a> for details.
	File Content:	Available if the file content is selected as the source. You can enter the content of the Dockerfile into the field.
	Image name:tag	Provide a newline-separated list of image name:tag(s)
	Additional arguments for 'build' command	Supply additional arguments to the docker build command. See <a href="#">Docker documentation</a> for details.
push	Remove image from agent after push	If selected, TeamCity will remove the image with <code>docker rmi</code> at the end of the step
	Image name:tag	Provide a newline-separated list of image name:tag(s)
other	Command name	Docker sub-command, like <code>push</code> or <code>tag</code> . For run, use <a href="#">Docker Wrapper</a>
	Working directory	
	Additional arguments for the command	Additional arguments that will be passed to the docker command.

## Docker Compose

TeamCity-Docker integration includes the Docker Compose runner.



### Requirements

The integration requires Docker installed on the build agents. Docker Compose also needs to be installed to use the [Docker Compose](#) build runner.

## Supported Environments

TeamCity-Docker support can run on Mac, Linux, and Windows build agents. It uses the '`docker`' executable on the build agent machine, so it should be runnable by the build agent user.



- On Linux, the integration will run if the installed Docker is detected.
- On Windows, the integration works in the Windows container mode only. Docker on Windows with the Linux container mode enabled is not supported, an error is reported in this case.
- On macOS, the official Docker support for Mac should be installed for the user running the build agent.

## Docker Compose

The runner allows starting [Docker Compose](#) build services and shutting down those services at the end of the build.

The Docker Compose runner supports one or several [Docker Compose YAML file](#)(s) with a description of the services to be used during the build. The path to the docker-compose.yml file(s) should be relative to the [checkout directory](#). When specifying several files, separate them with a space.

The executed commands are

```
# The commands are executed with the current working directory, where the docker-compose file resides.  
docker-compose -f <docker-compose.yml> [-f <docker-compose2.yml>] up -d  
# At the end of the build, for each docker compose build step the build agent will run:  
docker-compose -f <docker-compose.yml> [-f <docker-compose2.yml>] down -v
```

When using Docker Compose with images which support [HEALTHCHECK](#), TeamCity will wait for the `healthy` status of all containers, which support this parameter.

If the start of Docker Compose was successful, the TeamCity agent will register the `TEAMCITY_DOCKER_NETWORK` environment variable containing the name of the Docker Compose default network. This network will be passed transparently to the [Docker Wrapper](#) when used in some build runners.

## .NET CLI (dotnet)

TeamCity comes with built-in support of .NET CLI toolchain providing .NET CLI (dotnet) build steps, CLI detection on the build agents, and auto-discovery of build steps in your repository.

This page provides details on configuring the .NET CLI (dotnet) runner. Also see [the related blog post](#).



- .NET Core SDK has to be installed on your build agent machines.
- The .NET CLI tools path has to be added to the `PATH` environment variable. You can also configure the `DOTNET_HOME` environment variable for your TeamCity build agent user, e.g: `DOTNET_HOME=C:\Program Files\dotnet\`

Option	Description
Command:	Select a <code>dotnet</code> command from the drop-down. Depending on the selected command, some of the options below will vary. The currently supported commands are: <ul style="list-style-type: none"> <li>• <code>build</code></li> <li>• <code>clean</code></li> <li>• <code>pack</code></li> <li>• <code>publish</code></li> <li>• <code>restore</code></li> <li>• <code>run</code></li> <li>• <code>test</code></li> <li>• <code>vstest</code></li> <li>• <code>msbuild</code></li> <li>• <code>nuget delete</code></li> <li>• <code>nuget push</code></li> </ul>
Projects:	Specify paths to projects and solutions. Wildcards are supported. Parameter references are supported. If you have a finished build, you can use the file/directory chooser here.
Working directory:	Optional, set if differs from the checkout directory. Parameter references are supported. If you have a finished build, you can use the file/directory chooser here.
Framework:	Specify the target framework, e.g. <code>netcoreapp</code> or <code>netstandard</code> . Parameter references are supported.
Configuration:	Specify the target configuration, e.g. Release or Debug. Parameter references are supported.
Runtime:	Specify the target runtime. Parameter references are supported.
Output directory:	The directory where to place outputs. Parameter references are supported. If you have a finished build, you can use the file/directory chooser here.
Version suffix:	Defines the value of the <code>\$(VersionSuffix)</code> property in the project. Parameter references are supported.
Command line parameters:	Enter additional command line parameters for <code>dotnet</code> .
Logging verbosity:	Select from the <Default>, Minimal, Normal, Detailed or Diagnostic.

#### Docker Settings

Since TeamCity 2018.1 the .NET CLI build step can be run in a [specified Docker container](#).

#### Code Coverage

[JetBrains dotCover](#) is supported as a coverage tool for the `msbuild`, `test`, and `vstest` commands.

#### Parameters Reported by Agent

When starting, the build agent reports the following parameters:

Parameter	Description
<code>DotNetCLI</code>	The .NET CLI version
<code>DotNetCLI_Path</code>	The path to .NET CLI executable
<code>DotNetCoreSDKx.x_Path</code>	The .NET Core SDK version

#### Adding Build Features

A "build feature" is a piece of functionality that can be added to a build configuration to affect running builds or reporting build results.

Build Configuration Settings | Build Features page displays the configured features and allows you to manage them. It is possible to disable a configured build feature temporarily or permanently at any time, even if it is inherited from a build configuration template using the option in the last column of the Build Features list.

The currently available build features are:

- AssemblyInfo Patcher
- Automatic Merge
- Build Files Cleaner (Swabra)
- Commit Status Publisher
- Docker Support
- File Content Replacer
- Free disk space
- NuGet Feed Credentials
- NuGet Packages Indexer
- Performance Monitor
- Ruby Environment Configurator
- Shared Resources
- SSH Agent
- VCS Labeling
- XML Report Processing

## VCS Labeling

TeamCity can label (tag) sources of a particular build (automatically or manually) in your version control. The list of labels applied and their application status is displayed on the [Changes tab](#) of the build results page.

On this page:

- Automatic VCS labeling
- Manual VCS labeling
- Subversion Labeling Rules
- [Labeling Rule Examples](#)

### Automatic VCS labeling

You can set TeamCity to label the sources of a build depending on the build status automatically. The process takes place in the background after the build finishes and does not affect the build status, which means that a labeling failure is not a standard [notification event](#). However, the users subscribed for [notifications about failed builds](#) of the current build configuration will be notified about a labeling failure.

Any errors encountered during labeling are reported on the [Changes tab](#) of the build results page.

Labeling is configured for a build configuration/template.

Automatic VCS labeling is configured on the [Build Features page](#) of the Build Configuration settings.

To configure automatic labeling, you need to specify the root to label and the labeling pattern. If you have [branches](#) configured for your build configuration, you can label builds from [branches you select](#).

You can override the labeling settings inherited from a template completely; you can also apply different labels to different VCS roots.



Labeling uses the credentials specified for the VCS root and the write access to the sources repository is required.

Note that if you change the VCS settings of a build configuration, they will be used for labeling only in the new builds.

"Moving" labels (a label with the same name for different builds, e.g. "SNAPSHOT") are currently supported only for CVS.

For an example of using the Teamcity VCS labeling feature to automate tag creation, refer to this [external posting](#).

### Manual VCS labeling

To label the sources manually:

Navigate to the [build results](#) page, click Actions and select Label this build sources from the drop-down.

Manual labeling uses the VCS settings actual for the build.

### Subversion Labeling Rules

To label Subversion VCS roots, additional configuration - [labeling rules](#) defining the SVN repository structure - is required.

Labeling rules are specified as newline-delimited rules in the following format:

```
TrunkOrBranchRepositoryPath => tagDirectoryRepositoryPath
```

The repository paths can be relative and absolute (starting from "/"). Absolute paths are resolved from the SVN repository root (the topmost directory you have in your repository), relative paths are resolved from the TeamCity VCS root.

When creating a label, the sources residing under `TrunkOrBranchRepositoryPath` will be put into the `tagDirectoryRepositoryPath/tagName` directory, where `tagName` is the name of the label as defined by the labeling pattern of the build configuration. If no sources match the `TrunkOrBranchRepositoryPath`, no label will be created.

The `tagDirectoryRepositoryPath` path must already exist in the repository.

If the `tagDirectoryRepositoryPath` directory already contains a subdirectory with the current label name, the labeling process will fail, and the old tag directory won't be deleted or affected.

For example, there is a VCS root with the URL `svn://address/root/project` where `svn://address/root` is the repository root, and the repository has the structure:

```
-project  
--trunk  
--branch1  
--branch2  
--tags
```

In this case the labeling rules should be:

```
/project/trunk=>/project/tags  
/project/branch1=>/project/tags  
/project/branch2=>/project/tags
```

#### Labeling Rule Examples

You can use variables substitution in both labeling rules and labeling patterns. See a labeling rule example in a VCS root used in different configurations:

```
/projects/%projectName%/trunk => /projects/%projectName%/tags
```

This will require you to set the `%projectName%` configuration parameter in the build configuration settings.

By default, TeamCity will append the label name to the end of the specified target path. If you want to have a different directory structure and put the label in the middle of the target path, you can use the following syntax:

```
/project/trunk => /tagged_configurations/%%system.build.label%%/project  
/modules/module1/trunk => /tagged_configurations/%%system.build.label%%/module1  
/modules/module2/trunk => /tagged_configurations/%%system.build.label%%/module2
```

Thus, `%%system.build.label%%` will be replaced with the tag name (please note the double %% sign at the beginning - it is important).

#### Ruby Environment Configurator

The Ruby environment configurator build feature passes Ruby interpreter to all build steps. The build feature adds the selected Ruby interpreter and gems bin directories to the system PATH environment variable and configures other necessary environment variables in case of the [RVM](#) interpreter. E.g. in the [Command Line](#) build runner you will be able to directly use such commands as ruby, rake, gem, bundle, etc. Thus if you want to install gems before launching the [Rake](#) build runner, you need to add the [Command Line](#) build step which launches a custom script, e.g.:

```
gem install rake --no-ri --no-rdoc  
gem install bundler --no-ri --no-rdoc
```

## Ruby Environment Configurator Settings

Option	Description
Ruby interpreter path	<p>the path to Ruby interpreter. If not specified, the interpreter will be searched in the <code>PATH</code>. In this field you can use values of environment and system variables. For example:</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">%env.I_AM_DEFINED_IN_BUILDAgent_CONFIGURATION%</div>
RVM interpreter	<p>Specify here the RVM interpreter name and optionally a gemset configured on a build agent. Note, that the interpreter name cannot be empty. If gemset isn't specified, the default one will be used.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"><p><b>i</b> This option can be used if you don't want to use the <code>.rvmrc</code> settings, for instance to run tests on different ruby interpreters instead of those hard-coded in the <code>.rvmrc</code> file.</p></div>
RVM with <code>.rvmrc</code> file	Specify here the path to a <code>.rvmrc</code> file relative to the checkout directory. If the file is specified, TeamCity will fetch environment variables using the <code>rvm-shell</code> and will pass it to all build steps.
Fail build if Ruby interpreter wasn't found	Check the option to fail a build if the Ruby environment configurator cannot pass the Ruby interpreter to the step execution environment because the interpreter wasn't found on the agent.

## See also:

[Administrator's Guide: Configuring Build Steps | Command Line | Rake](#)

## Build Files Cleaner (Swabra)

Swabra (originally from the Russian noun 'shvabra' - a mop, also from the English verb 'swab' - clean with a mop) is a bundled plugin allowing you to clean files created during the build.

The plugin remembers the state of the file tree after the sources checkout and deletes all the newly added files at the end of the build or at the next build start depending on the settings. Swabra also detects files modified or deleted during the build and reports them to the build log (however, such files are not restored by the plugin). The plugin can also ensure that by the start of the build there are no files modified or deleted by previous builds and initiate clean checkout if such files are detected.

Moreover, Swabra gives the ability to dump processes which lock directory by the end of the build (requires `handle.exe`)

Swabra can be added as a build feature to your build configuration regardless of what set of build steps you have. By configuring its options you can enable scanning the checkout directory for newly created, modified and deleted files and enable file locking processes detection.

**i** Swabra should be used with the `automatic checkout` only: after this build feature is configured, it will run before the first build step to remember the state of the file tree after the sources checkout and to restore it after the build.

The checkout directory state is saved into a file in the `caches` directory named `<checkout_directory_name_hash>.snapshot` using the `DiskDir` format. The path to the checkout directory to be cleaned is saved into the `snapshot.map` file. The snapshot is used later (at the end of the build or at the next build start) to determine which files and folders are newly created, modified or deleted. It is done based on the actual files' presence, last modification data and size comparison with the corresponding records in the snapshot.

## Configuring Swabra Options

Option	Description
Files cleanup	Select whether you want to perform build files cleanup, and when it will be performed.
Clean checkout	Select the Force clean checkout if cannot restore clean directory state option to ensure that the checkout directory corresponds to the sources in the repository at the build start. If Swabra detects any modified or deleted files in the checkout directory before the build start, it will enforce <a href="#">clean checkout</a> . The build will fail if Swabra cannot delete some files created during the previous build.  If this option is disabled, you will only get warnings about modified and deleted files.
Paths to monitor	<p>Specify a newline-separated set of + -:path rules to define which files and folders are to be involved in the files collection process (by default and until explicitly excluded, the entire checkout directory is monitored). The path can be relative (based on the <a href="#">build's checkout directory</a>) or absolute and can include Ant-like wildcards. If no +: or -: prefix is specified, a rule as treated as "include".</p> <p>Rules on any path must come in the order from more general to more concrete. The top level path must always point to a directory. Specifying a directory affects its entire content and sub-directories. Note also that Swabra is case-sensitive.</p> <p>Examples:</p> <ul style="list-style-type: none"> <li>- :*/dir/* excludes all <code>dir</code> folders and their content,</li> <li>- :some/dir, +:some/dir/inner excludes <code>some/dir</code> folder and all its content except for the <code>inner</code> subfolder and its content</li> <li>+ :./file.txt includes only the specified file in the build checkout directory into monitoring</li> <li>- :file.txt excludes the specified file in the build checkout directory from monitoring</li> </ul> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">  Note that after removing some exclude rules, it is advisable to run a clean checkout.     </div>
Locking processes	Select whether you want Swabra to inspect the checkout directory for processes locking files in this directory, and what to do with such processes. Note that <code>handle.exe</code> is required on agents for locking processes detection.
Verbose output	Check this option to enable detailed logging to build log.

### Default excluded paths

If the build is set up to checkout on the agent, by default Swabra ignores all `.svn`, `.git`, `.hg`, `cvs` folders and their content. To turn off this behaviour, specify an empty `swabra.default.rules` configuration parameter.

### Installing Handle

You can install `handle.exe` from the Administration | Tools page.

Click the Install Tool button and select Sysinternals handle.exe from the list of tools.

Specify whether you want to download the latest version of `handle.exe` or upload it manually choosing the path on local machine, and click Add. TeamCity will download or upload `handle.exe` and send it to Windows agents.

`handle.exe` is present on agents only after the upgrade.

Please note that running `handle.exe` requires some additional permissions for the build agent user. For more details please read [this thread](#).

### Debug options

Generally snapshot file is deleted after files collection. Set the `swabra.preserve.snapshot` system property to preserve snapshots for debugging purposes.

### Clean Checkout

Please note that Swabra may sometimes cause [clean checkout](#) to restore clean checkout directory state.

To avoid unnecessary frequent clean checkouts, always set up identical Swabra build features for build configurations working in the same checkout directory.

Build configurations work in the same checkout directory if either the same custom checkout directory path or same VCS settings configured for them.

### Development links

See plugin page at [Swabra](#).

## XML Report Processing

The XML Report processing [build feature](#) allows using report files produced by an external tool in TeamCity. TeamCity will parse the specified files on the disk and report the results as the build results.

The report parsing can also be initiated from within the build via [service messages](#).

XML Report Processing supports the following testing frameworks:

- JUnit Ant task
- Maven Surefire/Failsafe plugin
- NUnit-Console XML reports
- TRX reports (for MSTest 2005-2015 and VSTest 2012-2015. Since TeamCity 10.0.4 - version 2017 of both engines is supported.)
- Google Test XML reports
- XML output from CTest

and the following code inspection tools:

- FindBugs (code inspections only): only FindBugs native format is supported (see corresponding [xsd](#)). The XML report generated by the FindBugs Maven Plugin is NOT supported: it has completely different schema layout and elements.
- PMD
- Checkstyle
- JSLint XML reports

and the following code duplicates tools:

- PMD Copy/Paste Detector XML reports

The bundled XML Report Processing plugin monitors the specified report paths, and when the matching files are detected, they are parsed according to the report type specified. For some report types, parsing of partially saved files is supported, so reporting is started as soon as first data is available and more data is reported as it is written on the disk.

The plugin takes into account only the files updated since the build start (determined by means of the last modification file timestamp).

## Configuring XML Report Processing

Add XML Report Processing as a [build feature](#) and configure its settings:

- Choose the report type and specify monitoring rules in the form of +| - :path separating them by a comma or new line.



To be processed, report XML files (or a directory) must be located in the checkout directory, and the path must be relative to [this directory](#).

Paths without the + | : prefix are treated as including. Ant-style wildcards are supported, e.g. `dir/**.xml` means all files with the `.xml` extension under the "dir" directory.



TeamCity loads generated reports once when they are created: make sure your build procedure generates files with unique names for each tests set without overwriting report files. Also, the files should have a modification timestamp within the build time, as otherwise they will be discarded as "outdated" (e.g. assumed to have been generated by a previous build).

- Check the Verbose output option to enable detailed logging to the build log.
- For FindBugs report processing, it is necessary to specify the path to the FindBugs installation on the agent. It will be used for retrieving actual bug patterns, categories and their messages.
- For FindBugs, PMD and Checkstyle code inspections reports processing you can specify maximum errors and warnings limits, exceeding which will cause a build failure. Leave these fields blank if there are no limits.

## Development Links

See plugin page at [XML Test Reporting](#).

See also:

[Concepts: Build Runner | Testing Frameworks](#)

## AssemblyInfo Patcher

The AssemblyInfo Patcher [build feature](#) allows setting a build number to an assembly automatically, without having to patch

the `AssemblyInfo.cs` files manually. When adding this build feature, you only need to specify the version format. Note that you can use TeamCity parameter references here.

 **AssemblyInfo Patcher** should be used with the [automatic checkout](#) only: after this build feature is configured, it will run before the first build step. TeamCity will first perform replacement in the files found in the build checkout directory and then run your build.

TeamCity searches for all `AssemblyInfo` (including `GlobalAssemblyInfo`) files: `.cs`, `.cpp`, `.fs`, `.vb` in their standard locations under the checkout directory and replaces the parameter for the `AssemblyVersion`, `AssemblyFileVersion` and `AssemblyInformationalVersion` attributes with the build number you have specified in the TeamCity web UI.

At the end of the build the files are reverted to the initial state.

Note that this feature will work only for standard projects, i.e. created by means of the Visual Studio wizard, so that all the `AssemblyInfo` files and content have a standard location.

 For replacing a wider range of values in a larger number of files, consider using the [File Content Replacer](#) build feature.

## Free disk space

The [Free disk space build feature](#) allows ensuring certain free disk space on the agent before the build by deleting files managed by the TeamCity agent (other build's checkout directories and various caches). When the feature is not configured, the default free space for a build is 3 GB.

On this page:

- [Analyzing and freeing disk space](#)
- [Configuring free disk space](#)
- [Other ways to set the free disk space value](#)
  - [Configuring artifacts cache](#)

### Analyzing and freeing disk space

Before the build and before each build preparation stage, the agent will check the currently available free disk space in three locations: the agent's system, the agent's temp directory, and the build checkout directory. All the locations have to meet the same specified requirement. If the failure condition is specified, the build will fail if either of the locations does not meet the requirement.

If the amount is less than required, the agent will try to delete the data of other builds before proceeding.

The data cleaned includes:

- the checkout directories that were marked for [deletion](#);
- contents of other build's checkout directories in the reversed most recently used order
- the cache of previously downloaded artifacts (that were downloaded to the agent via TeamCity artifact dependencies)
- cleaning the local [Docker caches](#)

If you need to make sure a checkout directory is never deleted while freeing disk space, set the `system.teamcity.build.checkoutDir.expireHours` property to "never" value. See more at [Build Checkout Directory](#).

### Configuring free disk space

You can use the [Free disk space build feature](#) to alter the default 3 GB of required disk space. Configure the settings below:

Setting	Description
Required free space:	You can specify a custom free disk space value here (in bytes or using one of the kb, mb, gb or tb suffixes).
Fail build if sufficient disk space cannot be freed:	Check the box to add the corresponding build failure condition.

### Other ways to set the free disk space value

For compatibility reasons the free disk space value can be specified via the properties below. However, using the Free disk space build feature is recommended as the properties can be removed in the future TeamCity versions.

The properties can be defined:

- globally for a build agent (in agent's `buildAgent.properties` file)
- for a particular build configuration by specifying its system properties.

The required free space value is defined with the following properties:

`system.teamcity.agent.ensure.free.space` for the build checkout directory.

`system.teamcity.agent.ensure.free.temp.space` for the agent's 'temp' directory. If `teamcity.agent.ensure.free.temp.space` is not defined, the value of the `teamcity.agent.ensure.free.space` property is used.

The values of these properties specify the amount of the available free disk space to be ensured before the build starts. The value should be a number followed by kb, mb, gb, kib, mib, or gib suffix. Use no suffix for bytes.

e.g. `system.teamcity.agent.ensure.free.space = 5gb`

#### Configuring artifacts cache

A TeamCity build agent maintains a cache of published and downloaded build artifacts to reduce network transfers to the same agent. The cache is stored in the `<Build Agent home>\system\artifacts_cache` directory and is cleaned automatically provided the Free disk space build feature is configured correctly.

If caching artifacts is undesirable (for example, when the artifacts are large and not used within TeamCity, or if the artifacts cache directory is located not on the same disk as the build checkout directory, or if the builds do not define the Free disk space build feature and the default 3Gb is not sufficient for a build), caching artifacts on the agent can be turned off by adding the `teamcity.agent.filecache.publishing.disabled=true` configuration parameter to a project or one of the build configurations of a project. However, the agent will still cache artifacts downloaded as artifact dependencies.

#### See also:

[Administrator's Guide: TeamCity Server Disk Space Watcher | Build Failure Conditions](#)

## Performance Monitor

The Performance Monitor build feature allows you to get the statistics on the CPU, disk and memory usage during a build run on a build agent.

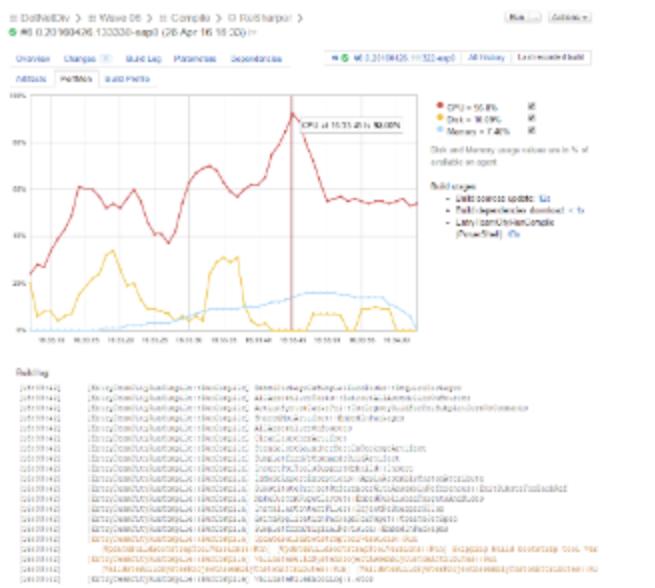
The Performance Monitor supports Windows, Linux, MacOs, Solaris, and, since TeamCity 2017.1, FreeBSD operating systems. All the operating systems except Windows require [Perl](#) installed.

Note that the performance monitor reports the load of the whole operating system. It will not report proper results if you have more than one agent running on the same host, or an agent and a server installed on the same machine.

 If you run your build agent as a Windows [service](#), the user starting the agent must be a member of the Performance Monitor Users group to be able to monitor performance metrics. Users can be added to the group via the `lusrmgr.msc` command.

When enabled, each build has an additional tab called PerfMon on the build results page, where this statistics is presented as a graph. The CPU usage shows the average CPU load during the build, the Disk and Memory usage values are relative to the total disk and memory available on the agent machine.

Clicking on a point in the graph displays the corresponding value (CPU in screenshot below) as well as the build log with the highlighted fragment for the selected time:



## Shared Resources

The Shared Resources [build feature](#) allows limiting concurrently running builds using a shared resource, such as an external (to the CI server) resource, e.g. test database, or a server with a limited number of connections, etc.

Some of such resources may be accessed concurrently but allow only a limited number of connections, others require exclusive access. Adding different locks to shared resources addresses these cases: now you can define a resource on the project level, configure its parameters (e.g. type and quota) and then use this resource in specific build configurations by adding the Shared Resources build feature to them. The build starts once the lock on the resource is acquired; on the build completion the lock is released. While the builds using the resource are [running](#), the resource is unavailable, and the other builds requiring locks will be waiting in the [queue](#).

On this page:

- [Adding and Editing Shared Resources](#)
  - [Types of Shared Resources](#)
- [Using Shared Resources in Build Configurations](#)
  - [Locks for Resources with Quotas](#)
    - [Lock Fairness](#)
  - [Locks for Resources with Custom Values](#)
  - [Locks for resources in composite builds](#)
- [Viewing Shared Resources Usage](#)
- [Development Links](#)

### Adding and Editing Shared Resources

You can add, edit shared resources, and explore their details (origin of the resource, its usage, etc.) on the Shared Resources tab of the project configuration page.

The resource name can contain only alpha-numeric characters and underscores ("\_") - maximum 80 characters - and should start with a Latin letter.

Shared resources defined at a project level are available in all its subprojects and build configurations.

On the subproject level, it is not possible to edit a resource inherited from a project.

However, it is possible to redefine a resource inherited from a project by creating a resource with the same name but with different settings in a subproject.

For example:

1. Create a resource A with the infinitive quota in project Parent.
2. Go to its subproject Child 1, and created a resource A (the same name as the resource in Parent) with the quota of 5. TeamCity will treat this resource as redefined in subproject Child 1: i.e. the settings of the resource A defined at the project level (infinite quota) will be propagated to all the other subprojects, with the exception of subproject Child 1 where resource A will have the quota of 5.

The usage of a resource is calculated by scanning the subtree of the current project; thus, if several projects use the same resource, only the usages in the current one will be counted.

## Types of Shared Resources

When you click Add new resource, three types of resources are available:

- Infinite resource is a shared resource with an unlimited number of read locks.
- Resource with quota: quota is a maximum number of read locks that can be acquired on the resource.
- Resource with custom values: a custom value (e.g. a URL) is passed to the build that has acquired a lock on such a resource.

## Using Shared Resources in Build Configurations

To define which build configuration(s) will use the resources, add this build feature to the build configuration settings.

When adding a shared resource, you need to select a resource available to this build configuration and define a lock.

The following locks are available:

### Locks for Resources with Quotas

For this resource type, two types of locks are supported:

- Read locks - shared (multiple running builds with read locks are allowed).
- Write locks - exclusive (only a single running build with a write lock is allowed).

A resource with a quota will allow concurrent access to multiple builds for reading but will restrict access to a single build for writes to the resource. Until all read locks are released, the build requiring a write lock will not start and will wait in the queue while new readers are able to acquire the lock.

 A build with a resource quota set to zero will not run.

### Lock Fairness

While read locks enable multiple builds to concurrently access a shared resource with a quota > 1, lock fairness ensures that the build queue is not violated. It means, that if there is shared resource with a quota >1, and there are several queued builds with read locks and a build with a write lock in between the readers, the build with the write lock will wait until all builds with read locks earlier in the queue finish and release the lock. Then the build requiring a write lock will be executed, and only after that the other readers can acquire the lock. Lock fairness does not allow builds with read locks interfere with build queue ordering and 'slip' past the build that is waiting for a write lock to become available.

### Locks for Resources with Custom Values

Resources with custom values support three types of locks:

- Locks on any available value: a build that uses the resource will start if at least one of the values is available. If all values are being used at the moment, the build will wait in the queue.
- Locks on all values: a build will lock all the values of the resource. No other builds that use this resource will start until the current one is finished.
- Locks on specific value: only a specific value of the resource will be passed to the build. If the value is already taken by a running build, the new build will wait in the [queue](#) until the value becomes available.

When the resource is defined and the locks are added, the build gets a configuration parameter with the name of the lock and with the value of the resource string (`teamcity.locks.readLock.<lockName>` or `teamcity.locks.writeLock.<lockName>`), e.g. the parameter name can be: `teamcity.locks.readLock.databaseUrl`.

### Locks for resources in composite builds

Shared resources can now be locked for composite builds as well. A lock on the specified resource will be acquired when a composite build starts (when the first build of the composite build chain starts); the lock will be released when the composite build finishes (the last build in its chain is finished).

The locks acquired on composite builds affect only these composite builds and are not propagated to their individual parts. For example, if a resource has a quota of N, then N composite builds that have a read lock on this resource can be run concurrently. The number of concurrent individual builds inside these composite builds will not be affected by the resource quota.

## Viewing Shared Resources Usage

Since TeamCity 2018.1, if at least one resource lock is defined in a build configuration, you can view the resources used by the build on the Shared Resources tab of the [Build Results](#) page. The tab displays the resources and their type, including the locks used by the build for each resource.

Clicking the resource name takes you back to the the [shared resources configuration](#) page on the project level.

## Development Links

See the [Shared Resources plugin page](#)

See also:

[JetBrains TV TeamCity Shared Resources Screencast](#)

## Automatic Merge

The Automatic Merge build feature tracks builds in branches matched by the configured filter and merges them into a specified destination branch if the build satisfies the condition configured (e.g. the build is successful). The merge occurs after the build finishes. The feature is supported for Git and Mercurial VCS roots for build configurations with enabled [feature branches](#). Team City also allows merging branches [manually](#).

### Automatic Merge Settings

Check [Adding Build Features](#) for notes on how to add a build feature.

All branches that are used in this feature must be present in a repository and included into the Branch Specification of the current build configuration.

Option	Description
Watch builds in branches	A <a href="#">filter</a> for logical names of the branches whose build's sources will be merged. Specify newline-delimited of rules in the form of <code>+ -:logical branch name</code> (with an optional <code>*</code> placeholder). Parameter references are supported here.
Merge into branch	A <a href="#">logical name</a> of the destination branch the sources will be merged to. Parameter references are supported here. The branch must be present in a repository and included into the Branch Specification.
Merge commit message	A message for a merge commit. The default is set to <code>Merge branch '%teamcity.build.branch%'</code> . Parameter references are supported here.
Perform merge if	A condition defining when the merge will be performed (either for successful builds only, or if build from the branch does not add new problems to destination branch).
Merge policy	Select to create a merge commit or do a fast-forward merge.

## Cascading Merge

It is possible to define a cascade of merge operations by adding several such features to a build configuration.

For example, you want to automatically merge all feature branches into an `integration` branch, and then configure another merge from the `integration` to the default branch.

1. Create the `integration` branch on your VCS repository.
2. Add the Automatic Merge build feature to your configuration.
  - a. In the Watch builds in branches filter, specify

`+:feature-*`

- b. In the Merge into branch, specify your `integration`. This will merge your feature branches to the `integration` branch.
3. Add one more Automatic Merge build feature to your configuration.
  - a. In the Watch builds in branches filter, specify

`+:integration`

- b. In the Merge into branch, leave your default branch.

See also a related [TeamCity blog post](#).

## NuGet Feed Credentials

When using NuGet packages from an external authenticated feed during a build on TeamCity, the credentials for connecting to that feed have to be specified.

Adding this information to source control is not a secure practice, so TeamCity provides the NuGet Feed Credentials [build feature](#) which allows interacting with feeds that require authentication.

When editing the build configuration, from the list of available [Build Features](#) select NuGet Feed Credentials. In the dialog that is opened, specify the feed URL and the credentials to connect to the feed.

You can add this build feature to any build configuration, one for every feed that requires authentication.

 Since TeamCity 9.1.4, when using TeamCity as the [internal Nuget server](#), the credentials specified via this build feature are ignored; the internal TeamCity authentication is used.

## NuGet Packages Indexer

When the NuGet packages indexer build feature is added to a build configuration, all .nupkg files published as artifacts of builds based on this configuration will be indexed and added to the selected NuGet feed. Indexing is performed on the agent side.

### SSH Agent

The SSH Agent [build feature](#), runs an SSH agent with the selected [uploaded SSH key](#) during a build. When your build script runs an SSH client, it uses the SSH agent with the loaded key.

Check [SSH Keys Management](#) for SSH key upload notes.

### Agent Setup

TeamCity SSH Agent uses a native ssh agent from the OpenSSH included with Linux and Mac OS X, so the feature works out of the box for these OS's. For Windows, OpenSSH needs to be installed (e.g. as a part of CygWin, MinGW or a part of Git distribution for Windows).

The SSH agent must be added to \$PATH on Unix-like OS's and to %PATH% on Windows.

For each TeamCity build agent a separate ssh agent is started, so it is possible to use this feature if several build agents are installed on the same machine.

### Disabling SSH host key checking

The first time you connect to a remote host, the SSH client asks if you want to add a remote host's fingerprint to the known hosts database at `~/.ssh/known_hosts`.

To avoid such prompts during a build, you need to configure the known hosts database beforehand. If you trust the hosts you are connecting to, you can disable known hosts checks:

- either for all connections by adding something like this in `~/.ssh/config`:

```
Host *
  StrictHostKeyChecking no
```

- or for an individual command by running an ssh client with the `-o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no` options.

See more information in the man pages for `ssh`, `ssh-agent` and `ssh-add` commands.

## File Content Replacer

File Content Replacer is the [build feature](#) which processes text files by performing regular expression replacements before a build. After the build, it restores the file content to the original state.

 File Content Replacer should be used with the [automatic checkout](#) only: after this build feature is configured, it will run before the first build step. TeamCity will first perform replacement in the specified files found in the build checkout directory and then run your build.

The common case of using File Content Replacer is replacing one attribute at a time in particular files, e.g. it can be used to patch files with the build number.

You can add more than one File Content Replacer build feature if you wish to:

- replace more than one attribute,
- replace one and the same attribute in different files/projects with different values.

This feature extends the capabilities provided by [AssemblyInfo Patcher](#).

Check the [Adding Build Features](#) section for notes on how to add a build feature.

On this page:

- [File Content Replacer Settings](#)
  - [Templates](#)
    - [.NET templates](#)
    - [.Net Core csproj templates](#)
    - [MFC templates](#)
    - [Xcode templates](#)
  - [Examples](#)
    - Extending an attribute value with a custom suffix
    - Patching only specified files
    - Specifying path patterns which contain spaces
    - Changing only the last version part / build number of the `AssemblyVersion` attribute:

#### File Content Replacer Settings

You can specify the values manually or use value presets for replacement, which can be edited if needed.

Option	Description
Template (optional):	File Content Replacer provides a template for every attribute to be replaced. Clicking the Load Template... button displays the combobox with templates containing value presets for replacement. The templates can be filtered by language (e.g. c#), file (e.g. <code>AssemblyInfo</code> ) or attribute (e.g. <code>AssemblyVersion</code> ) by typing in the combobox. When a template is selected, the settings are automatically filled with predefined values. See the <a href="#">section below</a> for template details.
Process files:	Click Edit file list and specify paths to files where the values to be replaced will be searched. Provide a newline- or comma-separated set of rules in the form of <code>+ -:[path relative to the checkout directory]</code> . Ant-like wildcards are supported, e.g. <code>dir/**/*.cs</code> . If a <a href="#">pre-defined template</a> is selected, the files associated with that template will be used.
File encoding:	By default, TeamCity will auto-detect the file encoding. To specify the encoding explicitly, select it from the drop-down. When specifying a custom encoding, make sure it is <a href="#">supported</a> by the agent. If a <a href="#">pre-defined template</a> is selected, the file encoding associated with that template will be used.
Find what:	Specify a pattern to search for, in the <a href="#">regular expression</a> format. <a href="#">MULTILINE</a> mode is on by default. If a <a href="#">pre-defined template</a> is selected, the pattern associated with that template will be used.
Match case:	By default, the comparison is case-sensitive. Uncheck for case-insensitive languages. If a <a href="#">pre-defined template</a> is selected, the comparison associated with that template will be used.

Regex:	<p>The box is checked by default and applies to both the search and replacement strings. Uncheck to use fixed strings.</p> <p><b>⚠</b> If you use <b>versioned settings</b> (XML or Kotlin DSL), in addition to the default <code>REGEX</code> and non-default <code>FIXED_D_STRINGS</code> mode, the <code>REGEX_MIXED</code> mode is available. In this mode, the search pattern is interpreted as a regular expression, but the replacement text will be <b>quoted</b> (the <code>\</code> and <code>\$</code> characters have no special meaning).</p> <p>See a sample File Content Replacer configuration for settings in <a href="#">Kotlin</a>:</p> <pre>features {     replaceContent {         fileRules = "**/*"         pattern = "(?iu)the\h+pattern\h+to\h+search\h+for"         regexMode = FileContentReplacer.RegexMode.REGEX_MIXED         replacement = """%teamcity.agent.work.dir%\nd_r\bin\isf"""     } }</pre>
Replace with:	<p>Type the text to be used for replacing the characters in the Find what box. To delete the characters in the Find what box from your file, leave this box blank.</p> <p><code>\$N</code> sequence references N-th capturing group. All backslashes (<code>\</code>) and dollar signs (<code>\$</code>) without a special meaning should be quoted (as <code>\\\</code> and <code>\\$</code>, respectively).</p>

## Templates

This section lists the available replacement templates.

### .NET templates

The templates for replacing the following **Assembly** attributes are provided (see [this section](#) for comparison with [AssemblyInfo Patcher](#)):

- `AssemblyTitle`
- `AssemblyDescription`
- `AssemblyConfiguration`
- `AssemblyCompany`
- `AssemblyProduct`
- `AssemblyCopyright`
- `AssemblyTrademark`
- `AssemblyCulture`
- `AssemblyVersion`
- `AssemblyFileVersion`
- `AssemblyInformationalVersion`
- `AssemblyKeyFile`
- `AssemblyKeyName`
- `AssemblyDelaySign`
- `CLSCompliant`
- `ComVisible`
- `Guid`
- `InternalsVisibleTo`
- `AllowPartiallyTrustedCallers`
- `NeutralResourcesLanguageAttribute`

### .Net Core csproj templates

- `AssemblyName`
- `AssemblyTitle`
- `AssemblyVersion`
- `Authors`
- `Company`

- Copyright
- Description
- FileVersion
- PackageId
- PackageVersion
- Product
- Title
- Version
- VersionPrefix
- VersionSuffix

## MFC templates

The templates for replacing the following MFC C++ resource keys are provided:

- FileDescription
- CompanyName
- ProductName
- LegalCopyright
- FileVersion\*
- ProductVersion\*

**⚠** In MFC \*.rc files, both `FileVersion` and `ProductVersion` occur twice, once in a dot-separated (e.g. `1.2.3.4`) and once in a comma-separated (e.g. `1,2,3,4`) form. If your `%build.number%` parameter has a format of `1.2.3.{0}`, using two build parameters with different delimiters instead of a single `%build.number%` is recommended.

## Xcode templates

The templates for replacing the following Core Foundation Keys in `Info.plist` files are provided:

- `CFBundleVersion`
- `CFBundleShortVersionString`
- or both `CFBundleVersion` and `CFBundleShortVersionString` at the same time

## Examples

### Extending an attribute value with a custom suffix

Suppose you do not want to replace your `AssemblyConfiguration` with a fixed literal, but want to preserve your `AssemblyConfiguration` from `AssemblyInfo.cs` and just extend it with a custom suffix, e.g.:

```
[assembly: AssemblyConfiguration("${AssemblyConfiguration} built by TeamCity")]
```

### Do the following:

change the default replacement	\$1MyAssemblyConfiguration\$7	to	\$1\$5 built by TeamCity\$7
--------------------------------	-------------------------------	----	-----------------------------

For changing complex regex patterns, [this external tool](#) might be useful.

### Patching only specified files

The default `AssemblyInfo` templates follow the usual Visual Studio project/solution layout; but a lot of information may be shared across multiple projects and can reside in a shared file (e.g. `CommonAssemblyInfo.cs`).

Suppose you want to patch this shared file only; or you want to patch `AssemblyInfo.cs` files on a per-project basis.

### Do the following:

1. Load the `AssemblyInfo` template corresponding to the attribute you are trying to process (e.g. `AssemblyVersion`)
2. Change the list of file paths in the [Look in](#) field from the default `*/Properties/AssemblyInfo.cs` to `*/CommonAssemblyInfo.cs` or list several files comma- or new-line separated files here, e.g. `myproject1/Properties/AssemblyInfo.cs, myproject2/Properties/AssemblyInfo.cs`.

### Specifying path patterns which contain spaces

Spaces are usually considered a part of the pattern, unless they follow a comma, as the comma is recognised as a delimiter.

Note that the TeamCity server UI trims leading and trailing spaces in input fields, so a single-line pattern like `<spaces>foo.bar`

will become `foo.bar` upon save. The following workarounds are available:

For a single pattern containing leading spaces	For [a single pattern containing] trailing spaces	Alternatively, to add a single pattern containing leading spaces, use an explicit inclusion rule:
<pre>'&lt;spaces&gt;foo.bar</pre>	<pre>foo.bar&lt;spaces&gt;,</pre>	<pre>+:&lt;spaces&gt;foo.bar </pre>

Changing only the last version part / build number of the `AssemblyVersion` attribute:

Suppose, your `AssemblyVersion` in `AssemblyInfo.cs` is `Major.Minor.Revision.Build` (set to `1.2.3.*`) , and you want to replace the `Build` portion (following the last dot (the `*`) only.

1. Load the `AssemblyVersion` in `AssemblyInfo (C#)_` template and change the default pattern:

```
(^\\s*\\[\\s*assembly\\s*:\\s*((System\\s*.\\.)?\\s*Reflection\\s*.\\.)?\\s*AssemblyVersion(Attribute)?\\s*\\(\\s*@?\\\"))([0-9\\*]+\\.?) + (\"\\s*\\\")\\s*\\])
```

to

```
(^\\s*\\[\\s*assembly\\s*:\\s*((System\\s*.\\.)?\\s*Reflection\\s*.\\.)?\\s*AssemblyVersion(Attribute)?\\s*\\(\\s*@?\\\"))([0-9\\*]+\\.+) + [0-9\\*]+ (\"\\s*\\\")\\s*\\])
```

and change the default replacement:

```
$1\\%build.number%$7
```

to

```
$1$5\\%build.number%$7
```



#### %build.number% format

Make sure your `%build.number%` format contains just a decimal number without any dots, or you may end up with a non-standard version like `1.2.3.4.5.6.2600` (i.e. `%build.counter%` is a valid value here while `4.5.6.%build.counter%` is not).

## Commit Status Publisher

Commit status publisher is a [build feature](#) which allows TeamCity to automatically send build statuses of your commits to an external system. The feature is implemented as an [open-source plugin](#) bundled with TeamCity.

The supported systems are:

- JetBrains Upsource
- GitHub (the build statuses for pull requests are supported as well)
- GitLab
- TFS/VSTS-hosted Git
- Atlassian Bitbucket Server (formerly Stash) and Atlassian Bitbucket Cloud

- Gerrit Code Review tool 2.6+

 GitLab

If you use a recent version of GitLab ( $\geq 9.0$ ), it is recommended to use the GitLab URL of the following format: `http[s]://<hostname>[:<port>]/api/v4` as GitLab will stop supporting the v3 API in GitLab 11. If you have `/api/v3` in your current TeamCity configurations, they may stop working with GitLab 11+, so consider changing the server URL to `api/v4`.

For older versions of GitLab, use the GitLab URL of the format `http[s]://<hostname>[:<port>]/api/v3`.

 Bitbucket

Make sure that the [TeamCity server URL](#) is FQDN, e.g. `http://myteamcity.domain.com:8111`. Short names, e.g. `http://myteamcity:8111` are rejected by the Bitbucket API.

For Bitbucket Cloud team accounts, it is possible to use the team name as the username, and the API key as the password.

 Gerrit

Commit Status Publisher in TeamCity 2018.1 supports Gerrit versions 2.6+. For configuring integration with earlier Gerrit versions please contact our [support](#).

 TFS/VSTS

Personal Access Tokens can be used for authentication. If a [VSTS connection](#) is configured, the personal access token can be automatically filled from the project connection.

To use the tool:

1. [Add the build feature](#) to your build configuration,
2. Use the default All attached VCS roots option if you want Commit Status Publisher to attempt publishing statuses for commits in all attached VCS roots or select a single repository for publishing build statuses
3. select your system as the publisher, and specify its connection details and credentials
4. Test the connection
5. Save your settings.

See the example below to configure sending the status of builds with changes included in your pull request from TeamCity to GitHub.

1. Configure the [branch specification](#) in your VCS Root ensuring that it includes pull requests. Detailed information is available in the Branch specification section of this TeamCity [blog post](#).
2. [Add the build feature](#):
  - Use the default All attached VCS roots option to publish statuses for commits in all attached VCS roots
  - Select GitHub as the publisher and specify its connection details and credentials and test the connection:

3. Save your settings.
  4. Commit changes to your source code and create a pull request in GitHub, then run a build with your changes in TeamCity.
- The Commit Status Publisher will inform you on the status of the build with your pull request changes:

- 1) It will show you whether the check is in progress ●, whether it failed ✘ or is successful ✓
- 2) hovering over the commit status will display the build summary
- 3) clicking the build status sign or the Details link will open the build results page in TeamCity:

This information is also available on the Commits tab of your pull request details:

The screenshot shows a GitHub pull request page for a project named "Julia-Alexandrova / Sample-Project". A specific pull request, #3, is selected. At the top, there are buttons for "Code", "Issues", "Pull requests", "Wiki", "Graphs", and "Settings". Below the pull request number, it says "Julia-Alexandrova wants to merge 3 commits into master from Julia-Alexandrova-patch-1". Under the commit list, there is a "Commits on May 24, 2016" section. The first commit, "Update HelloWorld.java" by Julia-Alexandrova committed 20 minutes ago, has a green checkmark icon and a tooltip "Success: Finished TeamCity Build Commit Status Publisher :: Build : Tests passed: 4, ignored: 13". This commit is highlighted with a blue border. The other two commits in the list have red X icons.

Similarly to the previous page, clicking the build status icon opens the [build results](#) page in the TeamCity web UI:

The screenshot shows the TeamCity build results page for build #4, which was triggered on May 24, 2016, at 16:39:20. The build took 28 seconds. The results table shows 4 tests passed and 13 tests ignored. The agent used was unit-425 (Default pool). The build was triggered by a user on May 24, 2016, at 16:39:39. The branch triggering the build was refs/heads/Julia-Alexandrova-patch-1.

Result	Tests passed: 4, Ignored: 13	Agent	unit-425 (Default pool)
Time:	24 May 16 16:39:20 - 16:39:28 (8s)	Triggered by:	you on 24 May 16 16:39
Branch:	<a href="#">refs/heads/Julia-Alexandrova-patch-1</a>		

## Docker Support

TeamCity comes with built-in Docker integration, which includes the Docker Support build feature. [Adding this build feature](#) will enable docker events monitoring: such operations as docker pull, docker run will be detected. The build feature adds the Docker Info tab to the [build results](#) page providing information on Docker-related operations. It also provides the following options:

- the ability to clean-up the images
- automatic login to an authenticated registry before the build and logout of it after the build

These options require a configured connection to a docker registry.  
Clean-up of images

If you have a build configuration which publishes images, you need to remove them at some point. You can select the corresponding option and instruct TeamCity to remove the images published by a certain build when the build itself is [cleaned up](#). It works as follows: when an image is published, TeamCity stores the information about the registry of the images published by the build. When the [server clean-up](#) is run and it deletes the build, all the configured connections are searched for the address of this registry and the images published by the build are cleaned up using the credentials specified in the connection found.

### Automatic Login to/Logout of Docker Registry

If you need to log in to a registry requiring authentication before a build, select the corresponding option and a connection to Docker configured in the project settings. Automatic logout will be performed after the build finishes.

## Configuring Unit Testing and Code Coverage

In this section:

- [.NET Testing Frameworks Support](#)
- [Configuring .NET Code Coverage](#)
- [Java Testing Frameworks Support](#)
- [Configuring Java Code Coverage](#)
- [Running Risk Group Tests First](#)

### .NET Testing Frameworks Support

To support the real-time reporting of test results, TeamCity should either run the tests using its own test runner or be able to interact with the testing frameworks so it receives notifications on test events. Custom TeamCity-aware test runners are used

to implement the bundled support for the testing frameworks.

#### NUnit

Please, refer to the [NUnit Support](#) page for details.

#### MSTest

Please, refer to the [MSTest Support](#) page for details.

#### MSpec

Dedicated test runner is available for MSPEC support. Please, refer to the [MSpec](#) page for details.

#### Gallio

Starting with version [3.0.4](#) [Gallio](#) supports on-the-fly test results reporting to TeamCity server.

Other testing frameworks (for example, MbUnit, NBehave, NUnit, xUnit.Net, and csUnit) are supported by Gallio and, thus, can provide tests reporting back to TeamCity.

As for coverage, Gallio supports NCover, to include coverage HTML reports to TeamCity build tab. See [Including Third-Party Reports in the Build Results](#).

#### xUnit

General information about xUnit support from its authors. Also a related [blog post](#).

See also:

[Troubleshooting: Visual C Build Issues](#)

#### NUnit Support

##### NUnit runner

The easiest way to set up NUnit tests reporting in TeamCity is to add [NUnit build runner](#) as one of the steps to your [build configuration](#) making sure the [requirements](#) are met and specify there all the required parameters.

**i** Supported NUnit versions: 2.2.10, 2.4.1, 2.4.6, 2.4.7, 2.4.8, 2.5.0, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3. Since TeamCity 9.1, NUnit 3.0 and above is also supported.

It is possible to have several versions of NUnit installed on an agent machine and use any of them in a build.

**!** NUnit version 3.4.0 is not supported by the [NUnit build runner](#) due to a problem in [NUnit](#). Only version 3.4.0 was affected, other NUnit 3.x versions work fine with TeamCity.

For details refer to [NUnit build runner](#) page.

Alternative approaches

If using [NUnit build runner](#) is inapplicable, TeamCity provides the following ways to configure NUnit tests reporting in TeamCity:

- TeamCity supports the standard `<nunit2>` [NAnt task](#).
- TeamCity provides the `<NUnitTeamCity>` [MSBuild task](#) and supports the `<NUnit>` [MSBuild task](#) from [MSBuild Community tasks](#).
- TeamCity provides its own NUnit Test Launcher that can be configured [in the MSBuild build script](#) or launched from the [command line](#).
- [TeamCity Addin for NUnit](#) is available to turn on reporting on the NUnit level without build procedure modifications.
- The bundled [XML Test Reporting plugin](#) allows importing any xml report to TeamCity. In this case it is not always possible to track results on the fly. You can add the XML Report Processing build feature to your build configuration, or use the following service message: `##teamcity[importData type='sometype' path='<path to the xml file>']`. Learn more: [XML Report Processing](#), [Build Script Interaction with TeamCity](#).
- TeamCity allows configuring tests reporting manually via [service messages](#).

Comparison matrix:

Approach	Real-Time Reporting	Execution without TeamCity	Tests Reordering	Implicit TeamCity .NET Coverage
NUnit runner	✓	✗	✓	✓
<nunit2> NAnt task	✓	✓/✗*	✓	✓
<NUnit> MSBuild task	✓	✓/✗*	✓	✓
<NUnitTeamCity> MSBuild task	✓	✓/✗*	✓	✓
TeamCity Addin for NUnit	✓	✗	✗	✗
TeamCity NUnit Test Launcher	✓	✗	✓	✓
XML Reporting Plugin	✗	only xml	N/A	N/A

\* TeamCity-provided tasks may have different syntax/behavior. Some workarounds may be required to run the script without TeamCity.

In addition to the common test reporting features, TeamCity relieves a headache of running your NUnit tests under x86 process on the x64 machine by introducing an explicit specification of the platform and runtime environment versions. You can define whether to use .NET Framework 1.1, 2.0 or 4.0 started under a MSIL, x64 or x86 platform.

This section covers:

- TeamCity NUnit Test Launcher
- NUnit for NAnt Build Runner
- NUnit for MSBuild
- MSBuild Service Tasks
- NUnit Addins Support
- TeamCity Addin for NUnit

See also:

[Administrator's Guide: NUnit build runner](#) | [Getting Started with NUnit](#) | [MSTest Support](#) | [Running Risk Group Tests First](#) | [XML Report Processing](#)

TeamCity NUnit Test Launcher

TeamCity provides its own NUnit tests launcher that can be used from command line. The tests are run according to the passed parameters and, if the process is run inside the TeamCity build agent environment, the results are reported to the TeamCity agent.

**i** Supported NUnit versions: 2.2.10, 2.4.1, 2.4.6, 2.4.7, 2.4.8, 2.5.0, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3. For NUnit 3.x, use the [NUnit build runner](#).

It is possible to have several versions of NUnit installed on an agent machine and use any of them in a build.



- If you need to access the path to the TeamCity NUnit launcher from some process, you can add the `%system.teamcity.dotnet.nunitlauncher%` environment variable.
- Values surrounded with "%" in custom scripts in the [Command line runner](#) are treated as TeamCity references.

You can pass the following command line options to the TeamCity NUnit Test Launcher:

```
 ${teamcity.dotnet.nunitlauncher} <.NET Framework> <platform> <NUnit vers.>
 [/category-include:<list>] [/category-exclude:<list>] [/addin:<list>] <assemblies to test>
```

Option	Description
<.NET Framework>	Version of .NET Framework to run tests. Acceptable values are v1.1, v2.0, v4.0 or ANY

<platform>	Platform to run tests. Acceptable values are x86, x64 and MSIL.
	 For .NET Framework 1.1 only MSIL option is available.
<NUnit vers.>	Test framework to use. The value has to be specified in the following format: NUnit-<version>.
/category-include:<list>	The list of categories separated by ',' (optional). Category expression are supported since <a href="#">NUnit v2.4.6</a> to NUnit v3.0 not inclusive.
/category-exclude:<list>	The list of categories separated by ',' (optional). Category expression are supported since <a href="#">NUnit v2.4.6</a> to NUnit v3.0 not inclusive.
/addin:<list>	List of third-party NUnit addins to use (optional).
<assemblies to test>	List of assemblies paths separated by ';' or space.
/runAssemblies:processPerAssembly	Specify, if you want to run each assembly in a new process.

### Category Expression

Beginning with NUnit 2.4.6 and up to but not including NUnit v3.0, a Category Expression can be used. The table shows some examples:

Expression	Action
A B C	Selects tests having any of the categories A, B or C.
A,B,C	Selects tests having any of the categories A, B or C.
A+B+C	Selects only tests having all three of the categories assigned.
A+B C	Selects tests with both A and B OR with category C.
A+B-C	Selects tests with both A and B but not C.
-A	Selects tests not having category A assigned.
A+(B C)	Selects tests having both category A and either of B or C.
A+B,C	Selects tests having both category A and either of B or C.

Note: As shown by the last two examples, the comma operator (,) is equivalent to the pipe (|) but has a higher precedence. The order of evaluation is as follows:

1. Unary exclusion operator (-)
2. High-precedence union operator (,)
3. Intersection and set subtraction operators (+ and binary -)
4. Low-precedence union operator (|)

Note: Since the operator characters have special meaning, you should avoid creating a category that uses any of them in its name. For example, the category "db-tests" could not be used in the command line, as it appears to mean "run category db, except for category tests." The same limitation applies to the characters having a special meaning for the shell you are using.

## Examples

The following examples assume that the `teamcity.dotnet.nunitlauncher` property is set as a system property on the [Parameters](#) page of the Build Configuration.

Run tests from an assembly:

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.2.10 Assembly.dll
```

Run tests from an assembly with NUnit categories filter

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.2.10 /category-include:C1 /category-exclude:C2  
Assembly.dll
```

Run tests from assemblies:

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.5.0 /addin:Addin1.dll;Addin2.dll Assembly.dll  
Assebly2.dll
```

NUnit for NAnt Build Runner

 Supported NUnit versions: 2.2.10, 2.4.1, 2.4.6, 2.4.7, 2.4.8, 2.5.0, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3.

This section assumes, that you already have a NAnt build script with configured `nunit2` task in it, and want TeamCity to track test reports without making any changes to the existing build script. Otherwise, consider adding [NUnit build runner](#) as one of the steps for your build configuration.

In order to track tests defined in NAnt build via standard `nunit2` task, TeamCity provides custom [`<nunit2>` task](#) implementation, and automatically replaces the original `<nunit2>` task with its own task. Thus when the build is triggered, TeamCity starts TeamCity NUnit Test Launcher using own implementation of `<nunit2>`. This allows you to leave your build script without changes and receive on-the-fly test reports in the TeamCity.

 If you don't want TeamCity to replace the original `nunit2` task, consider the following options:

- Use NUnit console with TeamCity Addin for NUnit.
- Import xml tests results via XML Test Report plugin.
- Use command line TeamCity NUnit Test Launcher.
- Configure reporting tests manually via service messages.
- To disable `nunit2` task replacement set `teamcity.dotnet.nant.replaceTasks` system property with value false.

TeamCity `nunt2` task implementation supports additional options that can be specified either as NAnt `<property>` tasks in the build script, or as System Properties under Build Configuration -> Build Parameters.

The following options are supported for TeamCity `<nunit2>` task implementation:

Property name	Description
<code>teamcity.dotnet.nant.nunit2.failonfailureatend</code>	Run all tests regardless of the number of failed ones, and fails if at least one test has failed.
<code>teamcity.dotnet.nant.nunit2.platform</code>	Sets desired runtime execution mode for .NET 2.0 on x64 machines. Supported values are x86, x64 and ANY(default).
<code>teamcity.dotnet.nant.nunit2.platformVersion</code>	Sets desired .NET Framework version. Supported values are v1.1, v2.0, v4.0. Default value is equal to NAnt target framework
<code>teamcity.dotnet.nant.nunit2.version</code>	Specifies which version of the NUnit runner to use. The value has to be specified in the following format: NUnit-<version>. It is possible to have several versions of NUnit installed on an agent machine and use any of them in a build.
<code>teamcity.dotnet.nant.nunit2.addins</code>	Specifies the list of third-party NUnit addins used for NAnt build runner.
<code>teamcity.dotnet.nant.nunit2.runProcessPerAssembly</code>	Set true if you want to run each assembly in a new process.

TeamCity NUnit test launcher will run tests in the .NET Framework, which is specified by NAnt target framework, i.e. on .NET Framework 1.1, 2.0 or 4.0 runtime. TeamCity also supports test categories for `<nunit2>` task.

 Adding the listed properties to the NAnt build script makes it TeamCity dependent. To avoid this, specify properties as System Properties under Build Configuration, or consider adding `<if>` task.

 If you need TeamCity test runner to support third-party NUnit addins, please, refer to the [NUnit Addins Support](#) section for the details.

## Examples

Start tests from a single assembly files under x64 mode on .NET 2.0.

```
<property name="teamcity.dotnet.nant.nunit2.platform" value="x64" />
<nunit2>
    <formatter type="Plain" />
    <test assemblyname="MyProject.Tests.dll" />
</nunit2>
```

Run all tests from category C1, but not C2.

```
<nunit2 verbose="true" haltonfailure="false" failonerror="true">
    <formatter type="Plain" />
    <test>
        <assemblies>
            <include name="dll.dll" />
        </assemblies>
        <categories>
            <include name="C1" />
            <exclude name="C2"/>
        </categories>
    </test>
</nunit2>
```

Explicitly specify version on NUnit to run tests with.

Note, that in this case, the following property should be added before nunit2 task call.

```
<property name="teamcity.dotnet.nant.nunit2.version" value="NUnit-2.4.10" />
<nunit2> <!--....--> </nunit2>
```

## NUnit for MSBuild

This page describes how to use NUnit from MS build.

- To use NUnit prior to 3.0 from MSBuild, see [Working with TeamCity-provided NUnit task](#)
- To use NUnit 3.0 and above from MSBuild, see [Working with NUnit 3.0](#)

### Working with NUnit Task in MSBuild Build

 The information in this section is applicable if you are using NUnit prior to version 3.0. For later versions, refer to the [section below](#).

This section assumes that you already have an MSBuild build script with a configured `NUnit` task in it, and want TeamCity to track test reports without making any changes to the existing build script. Otherwise, consider adding [NUnit build runner](#) as one of the steps for your build configuration.

In this section:

- [Using NUnitTeamCity task in MSBuild Build Script](#)
- [Examples](#)

### Using NUnitTeamCity task in MSBuild Build Script

TeamCity provides a custom `NUnitTeamCity` task compatible with the `NUnit` task from [MSBuild Community tasks](#) project. If

you provide the `NUnitTeamCity` task in your build script, TeamCity will launch its own test runner based on the options specified within the task. Thus, you do not need to have any NUnit runner, because TeamCity will run the tests.

In order to correctly use the `NUnitTeamCity` task, perform the following steps:

1. Make sure the `teamcity_dotnet_nunitlauncher` system property is accessible on build agents. Build agents running Windows should automatically detect these properties as environment variables. If you need to set them manually, see defining [agent specific properties](#) for more information.
2. Configure your MSBuild build script with `NUnitTeamCity` task using the following syntax:

```
<UsingTask TaskName="NUnitTeamCity"  
AssemblyFile="$(teamcity_dotnet_nunitlauncher_msbuild_task)" />
```

```
<NUnitTeamCity Assemblies="@({assemblies_to_test})" />
```

The following attributes are supported by `NUnitTeamCity` task:

Property name	description
Platform	Execution mode on a x64 machine. Supported values are: x86, x64 and ANY.
RuntimeVersion	.NET Framework to use: v1.1, v2.0, v4.0, ANY. By default, the MSBuild runtime is used. Default is v2.0 for MSBuild 2.0 and 3.5. For MSBuild 4.0 default value is v4.0
IncludeCategory	As used in the <code>NUnit</code> task from <a href="#">MSBuild Community tasks</a> project.
ExcludeCategory	As used in the <code>NUnit</code> task from <a href="#">MSBuild Community tasks</a> project.
NUnitVersion	Version of NUnit to be used to run the tests. Supported NUnit versions: 2.2.10, 2.4.1, 2.4.6, 2.4.7, 2.4.8, 2.5.0, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3. For example, <code>NUnit-2.2.10</code> .  To use NUnit 3.0 and above, see the <a href="#">section below</a> .
Addins	List of third-party NUnit addins to be used. For more information on using NUnit addins, refer to <a href="#">NUnit Addins Support</a> page.
HaltIfTestFailed	True to fail task, if any test fails.
Assemblies	List of assemblies to run tests with.
RunProcessPerAssembly	Set true, if you want to run each assembly in a new process.



- Custom TeamCity `NUnit` task also supports additional attributes. For the list of available attributes refer to the [Using `NUnitTeamCity` task in MSBuild Build Script](#) section.
- If you need the TeamCity test runner to support third-party NUnit addins, please, refer to the [NUnit Addins Support](#) section for the details.

Example (part of the MSBuild build script):

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">  
  <UsingTask TaskName="NUnitTeamCity"  
    AssemblyFile="$(teamcity_dotnet_nunitlauncher_msbuild_task)"/>  
  
  <Target Name="SayHello">  
    <NUnitTeamCity Assemblies="!!!*put here item group of assemblies to run tests on*!!!" />  
  </Target>  
</Project>
```

Important Notes

- Be sure to replace `".` with `_` when using [System Properties](#) in MSBuild scripts. For example, use `teamcity_dotnet_nunitlauncher_msbuild_task` instead of `teamcity.dotnet.nunitlauncher.msbuild.task`
- TeamCity also provides [Visual Studio Solution Runner](#) for solution files of Microsoft Visual Studio 2005 and above. It allows you to use MSBuild-style wildcards for the assemblies to run unit tests on.

#### Examples

Run NUnit tests using specific NUnit runner version:

```
<Target Name="build_01">
  <!-- start tests for NUnit-2.2.10 -->
  <NUnitTeamCity Assemblies="@{(TestAssembly)}" NUNITVersion="NUnit-2.2.10"/>

  <!-- start tests for NUnit-2.4.6 -->
  <NUnitTeamCity Assemblies="@{(TestAssembly)}" NUNITVersion="NUnit-2.4.8"/>
</Target>
```

Run NUnit tests with custom addins with NUnit 2.4.6:

```
<Target Name="build">
  <NUnitTeamCity Assemblies="@{(TestAssembly)}" Addins="NUnitExtension.RowTest.AddIn.dll"
  NUNITVersion="NUnit-2.4.6"/>
</Target>
```

Run NUnit tests with custom addins with NUnit 2.4.6 in per-assembly mode:

```
<Target Name="build">
  <NUnitTeamCity Assemblies="@{(TestAssembly)}" Addins="NUnitExtension.RowTest.AddIn.dll"
  NUNITVersion="NUnit-2.4.6" RunProcessPerAssembly="True"/>
</Target>
```



To make a TeamCity independent build script, consider the following trick:

```
<NUnitTeamCity ... Condition="$(TEAMCITY_VERSION) != "" />
```

MSBuild Property `TEAMCITY_VERSION` is added to msbuild when started from TeamCity.

#### Working with NUnit 3.0

The information in this section is applicable if you are using NUnit 3.0 and above. For earlier versions of NUnit, refer to the [section above](#).

Starting from version 3.0, NUnit supports TeamCity natively, so there is no need to use a special task for MSBuild as it was done for the [earlier NUnit versions](#). The simplest way is to run the NUnit console via the standard [Exec task](#). For example:

```

<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="14.0" DefaultTargets="RunTests"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<Target Name="RunTests">
<Exec IgnoreExitCode="True" Command="nunit3-console.exe Tests.dll">
<Output TaskParameter="ExitCode" ItemName="exitCode" />
</Exec>
<Error Text="Error while running tests" Condition="@{exitCode} < 0" />
</Target>
</Project>

```

The NUnit console returns the number of failed tests as the positive exit code and, in case of the NUnit test infrastructure failure, as the negative exit code.

TeamCity controls the test execution progress, but the NUnit infrastructure exceptions may not allow TeamCity to collect the required information. That is why the `IgnoreExitCode="True"` attribute needs to be set, which will ignore the positive exit codes and will not interrupt the build due to several failed tests. The [Error](#) task will stop the build in case of the test infrastructure errors for the negative exit codes.

### Build Step (3 of 3): Tests ▾

<b>Runner type:</b>	MSBuild	Runner for MSBuild files
<b>Step name:</b>	Tests	
Optional, specify to distinguish this build step from other steps.		
<b>Build file path:</b> *	sample2.proj	
The specified path should be relative to the checkout directory.		
<b>MSBuild version:</b>	Microsoft Build Tools 2015	
<b>MSBuild ToolsVersion:</b>	14.0	
<b>Run platform:</b>	x86	
<b>Targets:</b>	<input type="text"/>	
Enter targets separated by space or semicolon.		
<b>Command line parameters:</b>	<input type="text"/>	
Enter additional command line parameters to MSBuild.exe.		
<b>.NET Coverage</b>		
<b>.NET Coverage tool:</b> ⓘ	<No .NET Coverage>	
Choose a .NET coverage tool.		
⚠ Test code coverage is supported only for NUnit tests run using TeamCity facilities. ⓘ		
JetBrains dotTrace		
<b>Run build step under dotTrace profiler:</b>	<input type="checkbox"/>	
JetBrains dotMemory Unit		
<b>Run build step under JetBrains dotMemory Unit:</b>	<input type="checkbox"/>	

Besides the project file, you can define the MSBuild version and platform, the target, you can use profiles and other settings.

 The MSBuild runner option [Reduce test failure feedback time](#) will not work out of the box in this case. To use this feature, configure the [NUnit](#) build step.

[Getting Started with NUnit](#) contains details and more examples.

#### MSBuild Service Tasks

For MSBuild, TeamCity provides the following service tasks that implement the same options as the [service messages](#):

- [TeamCitySetBuildNumber](#)
- [TeamCityProgressMessage](#)
- [TeamCityPublishArtifacts](#)
- [TeamCityReportStatsValue](#)
- [TeamCityBuildProblem](#)
- [TeamCitySetStatus](#)

#### TeamCitySetBuildNumber

[TeamCitySetBuildNumber](#) allows user to change BuildNumber:

```
<TeamCitySetBuildNumber BuildNumber="1.3_{build.number}" />
```

It is possible to use '{build.number}' as a placeholder for older build number.

#### TeamCityProgressMessage

[TeamCityProgressMessage](#) allows you to write progress message.

```
<TeamCityProgressMessage Text="Progress message text" />
```

#### TeamCityPublishArtifacts

[TeamCityPublishArtifacts](#) allows you to publish all artifacts taken from MSBuild item group

```
<ItemGroup>
  <Files Include="*.dll" />
</ItemGroup>
<TeamCityPublishArtifacts SourceFiles="@{Files->'%(FullPath)'}" Condition="
'$(TEAMCITY_VERSION)' != ''"/>
```

#### TeamCityReportStatsValue

[TeamCityReportStatsValue](#) is a handy task to publish statistic values

```
<TeamCityReportStatsValue Key="StatsValueType" Value="42" />
```

#### TeamCityBuildProblem

[TeamCityBuildProblem](#) task reports a build problem which actually fails the build. Build problems appear on the build results page and also affect build status text.

```
<TeamCityBuildProblem description="description" identity="identity"/>
```

- Mandatory `description` attribute is a human-readable text describing the build problem. By default `description` appears

ars in build status text.

- `identity` is an optional attribute and characterizes particular build problem instance. Shouldn't change throughout builds if the same problem occurs, e.g. the same compilation error. Should be a valid Java id up to 60 characters. By default `identity` is calculated based on `description`.

#### TeamCitySetStatus

`TeamCitySetStatus` is a task to change current build status text.

Prior to TeamCity 8.0, this task was also used for changing build status to failure. However since TeamCity 7.1 [TeamCityBuildProblem](#) task should be used for this purpose.

```
<TeamCitySetStatus Status="<status value>" Text="{build.status.text} and some aftertext" />
```

'`{build.status.text}`' is substituted with older status text.

Status can have `SUCCESS` value.

#### NUnit Addins Support

NUnit addin is an extension that plug into NUnit core and changes the way it operates. Refer to the [NUnit addins page](#) for more information.

This section covers description of NUnit addins support for:

- [NAnt build runner](#)
- [TeamCity NUnit console launcher](#)
- [MSBuild build runner](#)

#### NAnt Build Runner

To support NUnit addins for NAnt build runner you need to provide in your build script the `teamcity.dotnet.nant.nunit2.addins` property in the following format:

```
<property name="teamcity.dotnet.nant.nunit2.addins" value="<list of paths>" />
```

where `<list>` is the list of paths to NUnit addins separated by ','.

For example:

```
<property name="teamcity.dotnet.nant.nunit2.addins"  
value="..../tools/addins/MyFirst.AddIn.dll;MySecond.AddIn.dll" />
```

#### TeamCity NUnit Console Launcher

To support NUnit addins for the [console launcher](#) you need to provide the `'/addins:<list of addins separated with ;>'` commandline option.

For example:

```
 ${teamcity.dotnet.nunitlauncher}  
/addin:..../tools/addins/MyFirst.AddIn.dll;nunit-addins/MySecond.AddIn.dll
```

#### MSBuild

 This section is applicable to NUnit versions prior to 3.0.

To support NUnit addins for the MSBuild runner, specify the `Addins` property for the `NUnitTeamCity` task with the following format:

```
Addins="<list>"
```

where `<list>` is the list of addins separated by ';' or ','.

For example:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" DefaultTargets="build">
<ItemGroup>
<TestAssembly Include="$(MSBuildProjectDirectory)/MyTests.dll" />
</ItemGroup>
<Target Name="build">
<NUnitTeamCity Assemblies="@{TestAssembly}"
Addins="..../tools/addins/MyFirst.AddIn.dll;nunit-addins/MySecond.AddIn.dll" />
</Target>
</Project>
```

## TeamCity Addin for NUnit



TeamCity NUnit Addin supports NUnit prior to version 3.0. For later versions, refer to the [this section](#).

If you run NUnit tests via the [NUnit console](#) and want TeamCity to track the test results without having to launch the TeamCity test runner, the best solution is to use TeamCity Addin for NUnit. You can plug this addin into NUnit, and the tests will be automatically reported to the TeamCity server.

Alternatively, you can opt to use the [XML Report Processing](#) build feature, or manually configure reporting tests by means of [service messages](#).

To be able to review test results in TeamCity, do the following:

1. In your build, set the path to the TeamCity Addin to the system property `teamcity.dotnet.nunitaddin` (for MSBuild it would be `teamcity_dotnet_nunitaddin`), and add the version of NUnit at the end of this path. For example:
  - For NUnit 2.4.X, use  `${teamcity.dotnet.nunitaddin}-2.4.X.dll` (for MSBuild:  `$(teamcity_dotnet_nunitaddin)-2.4.X.dll`)  
Example for NUnit 2.4.7: NAnt:  `${teamcity.dotnet.nunitaddin}-2.4.7.dll`, MSBuild:  `$(teamcity_dotnet_nunitaddin)-2.4.7.dll`
  - For NUnit 2.5.0 alpha 4, use  `${teamcity.dotnet.nunitaddin}-2.5.0.dll` (for MSBuild:  `$(teamcity_dotnet_nunitaddin)-2.5.0.dll`)
2. Copy the .dll and .pdb TeamCity addin files to the NUnit addin directory.



Although you can copy these files once, it is highly recommended to configure your builds so that the TeamCity addin files are copied to the NUnit addin directory for each build, because these files could be updated by TeamCity.

The following example shows how to use the NUnit console runner with the TeamCity Addin for NUnit 2.4.7 (on MSBuild):

```
<ItemGroup>
<NUnitAddinFiles Include="$(teamcity_dotnet_nunitaddin)-2.4.7.*" />
</ItemGroup>

<Target Name="RunTests">
<MakeDir Directories="$(NUnitHome)/bin/addins" />
<Copy SourceFiles="@{NUnitAddinFiles}" DestinationFolder="$(NUnitHome)/bin/addins" />
<Exec Command="$(NUnitHome)/bin/NUnit-Console.exe ${NUnitFileName}" />
</Target>
```

## Important Notes

### NUnit 2.4.8 Issue

NUnit 2.4.8 has the following known issue: NUnit 2.4.8 runner tries to load an assembly according to the created `AssemblyName` object; however, the `addins` folder of NUnit 2.4.8 is not included in application probe paths. Thus NUnit 2.4.8 fails to load any addin in the console mode.

To solve the problem, we suggest you use any of the following workarounds:

- copy the TeamCity addin assembly both to the NUnit `bin` and `bin/addins` folders

- patch `NUnit-Console.exe.config` to include the addins to application probe paths. Add the following code into the `config/runtime` element:

```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <probing privatePath="addins"/>
</assemblyBinding>
```

See original blog post on this issue <http://nunit.com/blogs/?p=56>

#### Environment variables

If you need to configure environment variables for NUnit explicitly, specify an environment variable with the value reference of `%system.teamcity.dotnet.nunitaddin%`.

See [Configuring Build Parameters](#) for details.

## MSTest Support



The MSTest runner is merged into the [Visual Studio Tests runner](#).

TeamCity provides support for MSTest 2005-2015 testing framework via parsing of the MSTest results file (.trx file). The appropriate Microsoft Visual Studio edition installed on the build agent is required.

Due to specifics of MSTest tool, TeamCity does not support on-the-fly test reporting for MSTest. All test results are imported after the tests run has finished.

There are two ways to report test results to TeamCity:

- Add the [MSTest runner](#) as one of your build steps.
- Configure [XML Report Processing](#) via build feature or via [service message](#) to parse the .trx reports that are produced by your build procedure.

The easiest way to set up MSTest tests reporting in TeamCity is to add MSTest build runner as one of the steps to your build configuration and specify there all the required parameters.

Please, refer to [MSTest build runner](#) page for details.

If the tests are already run within your build script and MSTest generates .trx reports, you can configure [service messages](#) to parse the reports.

#### Autodetection of MSTest

Prior to TeamCity 9.1 the MSTest location was reported as system properties: `%system.MSTest.8.0%`, `%system.MSTest.9.0%`, `%system.MSTest.10.0%`, `%system.MSTest.11.0%`, `%system.MSTest.12.0%`, `%system.MSTest.14.0%` that referred to MSTest 2008, 2010, 2012, 2013, 2015 correspondingly.

Since TeamCity 9.1 system parameters of the `%system.MSTest.xx.yy%` format were changed to configuration parameters of the `%teamcity.dotnet.mstest.xx.yy%` format.

If system properties are required for the build, the `mstest-legacy-provider` plugin can be used.

TeamCity auto-detects MSTest based on the registry values that describe the Visual Studio installation path. If Visual Studio is installed in a non-standard location, or the registry key is corrupted, or the TeamCity agent has no access to the VisualStudio directory, TeamCity may not be able to detect MSTest. In this case, the corresponding configuration parameter of the `%teamcity.dotnet.mstest.xx.yy%` format must be added to the build manually. It should contain the full path including the `MSTest.exe` executable, e.g. the default path for MSTest 2013 is `C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\MSTest.exe`

#### See also:

[Concepts: Testing Frameworks](#)  
[Administrator's Guide: NUnit Support](#)

[Configuring .NET Code Coverage](#)

TeamCity supports .NET code coverage using NCover, PartCover, and dotCover coverage engines. Configuration via the TeamCity UI is supported for

- NUnit build runner
- NAnt runner: <nunit2> [NAnt task](#)
- MSBuild runner: <NUnit> and <NUnitTeamCity> [MSBuild tasks](#)

For the [.NET CLI \(dotnet\)](#) runner and with NUnit version 3.x the only supported coverage tool is [JetBrains dotCover](#).

If you use a test framework other than NUnit, you can configure coverage analysis manually using the JetBrains dotCover console runner and TeamCity service messages as described in [Manually Configuring Reporting Coverage](#).

Details on configuring code coverage can be found on the following pages:

- [JetBrains dotCover](#)
- [NCover](#)
- [PartCover](#)
- [Manually Configuring Reporting Coverage](#)

See also:

[Administrator's Guide: NUnit Support](#)

## JetBrains dotCover

TeamCity comes bundled with the console runner of [JetBrains dotCover](#). Since TeamCity 2017.1, in addition to the bundled version, it is possible to install another version of JetBrains dotCover Command Line Tools and/or change the defaults using the [Administration | Tools](#) page.

After choosing the appropriate option in the .Net coverage section of a build step, you will be able to collect code coverage for your .Net project and then view the coverage statistics and detailed coverage report inside the [TeamCity web UI](#).

If you have a license for dotCover and have it installed on a developer machine, TeamCity-collected coverage results can be downloaded and viewed inside Visual Studio with the help of the [TeamCity Visual Studio Add-in](#).

 .NET Framework 3.5 or newer must be installed on the agent machine. This is necessary for the bundled dotCover to work. Your project can depend on another .NET Framework version.

On this page:

- [dotCover Settings](#)
- [Compile and Test in Different Builds](#)
  - [Bundled dotCover Versions](#)

### dotCover Settings

Path to dotCover Home	Leave this field blank to use the default dotCover. The <a href="#">bundled version</a> is set as default prior to TeamCity 2017.1; after this version you can mark any of the <a href="#">additionally installed</a> versions as default. Alternatively, specify the path to the dotCover installed on a build agent.
Filters	Specify a new-line separated list of filters for code coverage. Use the <code>+ -:assembly=*&gt;;type=**&gt;;method=***&gt;</code> to include or exclude assemblies from covered assemblies:  For example, to run coverage on all MyDemoApp assemblies but not on MyDemoApp.*.Tests, specify the following assembly filters for coverage: <code>+ :MyDemoApp.*</code> <code>- :MyDemoApp.*.Tests</code>  See also <a href="#">this blog post</a> .

Attribute Filters	If you do not want to know the coverage data solution-wide, you can exclude the code marked with an attribute (for example, with <code>ObsoleteAttribute</code> ) from the coverage statistics. You only need to specify these attribute filters here in the following format: the filters should be a new-line separated list; the <code>-:attributeName</code> syntax should be used to exclude the code marked with the attributes from code coverage. Use the asterisk (*) as a wildcard if needed. Supported only for dotCover 2.0 or newer.
Additional dotCover.exe arguments	Provide a new-line separated list of additional commandline parameters to pass to dotCover.exe

Note that dotCover coverage engine reports statement coverage instead of line coverage.

#### Compile and Test in Different Builds

To build a consistent coverage report, dotCover has to be able to find source files under the build checkout directory which should be easy if you build binaries and collect coverage in the same build, or if you use different builds, but they use a [snapshot dependency](#) and the same agent as well as the same [VCS settings](#).

If you need to build binaries in one build and collect code coverage in another one using different [checkout settings](#), some additional properties are required. It is assumed that:

- Build configuration A compiles code with debugging information and creates an artifact with assemblies and .pdb files
- Build configuration B runs tests with dotCover enabled and has a [snapshot dependency](#) on A.

To display the source code in the [Code Coverage tab](#) of build results of B, you need to point B to the same [VCS root](#) as A to get your source code in an appropriate location ([the checkout root](#)) and add an [artifact dependency](#) on build from the same chain of A (for dotCover to get the paths to the sources from the .pdb files).

You also need to tell TeamCity where to find the source code.

To do this, perform the following:

1. Add the `teamcity.dotCover.sourceBase` configuration parameter with the value `%teamcity.build.checkoutDir%` to the compiling build configuration A.
2. Add the configuration parameter `dotNetCoverage.dotCover.source.mapping` to your test configuration B with the value `%dep.bta.teamcity.dotCover.sourceBase%=>%teamcity.build.checkoutDir%`, where `bta` is the actual [id](#) of your configuration A.

#### Bundled dotCover Versions

This section provides information on the versions of dotCover bundled with TeamCity 10+ versions. For information on the earlier TeamCity releases, see the [previous documentation version](#).

TeamCity Version	dotCover Version
TeamCity 2018.1	dotCover 2018.1

You can view the installed versions of dotCover on the [Server Administration | Tools](#) page. The bundled version is set as default, you can install other versions and change the default settings.

#### See also:

- [Administrator's Guide: Manually Configuring Reporting Coverage](#)  
[Troubleshooting: dotCover issues](#)

#### NCover

TeamCity supports code coverage with NCover (1.x and 3.x) for NUnit tests run via TeamCity NUnit test runner, which can be

configured in one of the following ways: web UI, command line, [NUnit](#)[TeamCity task](#), [NUnit task](#), [nunit2 task](#).

### Important Notes

- To launch coverage, NCover and NCoverExplorer should be installed on the agent where the coverage build will run.
- You don't need to make any modifications to your build script to enable coverage.
- You don't need to explicitly pass any of the NCover/NCoverExplorer arguments to the TeamCity NUnit test runner.
- NCover supports .NET Framework 2.0 and 3.5 started under x86 platform (NCover 3.x also supports x64 platform and works with .NET Framework 4.0). Make sure, you use have specified the same platform both for NCover and NUnit.

## Configuring NCover 1.x

Make sure your NUnit tests run under x86.

To configure NCover 1.x:

1. While creating/editing Build Configuration, go to the Build Steps page.
2. Add one of the build steps that support NCover ([.NET Process Runner](#), [MSBuild](#), [MSpec](#), [MSTest](#), [NAnt](#), [NUnit](#)), configure unit tests.
3. Select NCover (1.x) in .NET coverage tool.
4. Set up the NCover options - refer to the description of the available options below.

Option	Description
Path to NCover	Specify the path to NCover installed on the build agent, or use %system.ncover.v1.path% to refer to the auto-detected NCover on the build agent.
Path to NCoverExplorer	Specify the path to NCoverExplorer on the build agent.
Additional NCover Arguments	Type additional arguments to be passed to NCover.   <ul style="list-style-type: none"><li>• Do not enter the arguments that can be configured in the web UI.</li><li>• Do not specify the output path for the generated reports. It is configured automatically by TeamCity.</li></ul>
Assemblies to Profile	Specify new-line separated assembly names (without paths and extensions), or leave this field blank to profile all assemblies. Equivalent to //a NCover.Console option.
Exclude Attributes	Specify the classes and methods with defined .NET attribute(s) to be excluded from the coverage statistics. Equivalent to //ea NCover.Console option
Report Type	Select the report type. For the details, refer to <a href="#">NCoverExplorer documentation</a> .
Sorting	Select the preferred sorting option. For the details, refer to <a href="#">NCoverExplorer documentation</a> .
Additional NCoverExplorer Arguments	Specify additional arguments to be passed to NCoverExplorer. Do not enter here the output path for the reports, nor specify arguments, for which there are corresponding options in the UI.

## Configuring NCover 3.x

Make sure you use have specified the same platform both for NCover and NUnit.

To configure NCover 3.x:

1. While creating/editing Build Configuration, go to the Build Steps page.
2. Add one of the build steps that support NCover ([.NET Process Runner](#), [MSBuild](#), [MSpec](#), [MSTest](#), [NAnt](#), [NUnit](#)), configure unit tests.
3. Select NCover (3.x) in .NET coverage tool.
4. Set up the NCover options - refer to the description of the available options below.

Option	Description
--------	-------------

Path to NCover 3	Specify the path to NCover. Alternatively, use %system.ncover.v3.x86.path% or %system.ncover.v3.x64.path% to refer to the auto-detected NCover 3 on the build agent.
Run NCover under	Select the preferred platform to run the coverage under - x86 or x64. Make sure the selected platform agrees with the one used for NUnit tests.
NCover Arguments	Specify NCover arguments, i.e. assemblies to profile and coverage tool specific arguments. Do not enter here arguments, which can be specified in the UI, nor enter here output path for generated reports and NCover process parameters. Use //ias .* to get coverage of all assemblies.
NCover Reporting Arguments	Specify additional NCover reporting arguments, except for the output path. Use //or FullCoverageReport:Html:{teamcity.report.path} to get the report.

#### Reporting NCover Results Manually

If .NET code coverage is collected by the build script and needs to be reported inside TeamCity (for example, [Rake runner](#), or if you run tests via a test launcher other than TeamCity NUnit Test Launcher), there is a way to let TeamCity know about the coverage data. Please read more at [Manually Configuring Reporting Coverage](#).

#### PartCover

TeamCity supports code coverage with PartCover (2.2 and 2.3) for NUnit tests run via the TeamCity NUnit test runner, which can be configured in one of the following ways: via the web UI, [command line](#), [NUnitTeamCity task](#), [NUnit task](#), [nunit2 task](#).

##### Important Notes

- In order to launch coverage, PartCover should be installed on an agent where coverage builds will run.
- You don't need to make any modifications to your build script to enable coverage.
- You don't need to explicitly pass any of the PartCover arguments to the TeamCity NUnit test runner.

To configure PartCover:

1. While creating/editing Build Configuration, go to the Build Runner page.
  2. Select PartCover (2.2 or 2.3) as a .NET coverage tool.
  3. Select the .Net runtime platform and version.
-  Some versions of PartCover support .NET Framework 2.0 and 3.5 and can be started under x86 platform only.  
Make sure you use the appropriate configuration options.
4. Set up the PartCover options - find the description of the available options below.

Option	Description
Path to PartCover	Specify the path to PartCover installed on a build agent, or the corresponding <a href="#">system property</a> , if configured.
Additional PartCover Arguments	Specify additional PartCover arguments, excluding the ones that can be specified using the web UI. Do not specify here the output path for the generated reports, because TeamCity configures it automatically.
Include Assemblies	Explicitly specify the assemblies to profile, or use [*]* to include all assemblies.
Exclude Assemblies	Explicitly specify the assemblies to be excluded from coverage statistics. If you have specified [*]* to profile all assemblies, type [JetBrains*]* here to exclude TeamCity NUnit test runner sources.
Report XSLT	Write new-line delimited xslt transformation rules in the following format: file.xslt=>generatedFileName.h tm1. You can use the default PartCover xslt as file.xslt, or your own. The Xslt files path is relative to the build checkout directory.
	 Note, that default xslt files bundled with PartCover 2.3 are broken and you need to write your own xslt files to be able to generate reports.

#### Reporting PartCover Results Manually

If .NET code coverage is collected by the build script and needs to be reported inside TeamCity (for example, [Rake runner](#), or if

you run tests via a test launcher other than TeamCity NUnit Test Launcher), there is a way to let TeamCity know about the coverage data. Please read more at [Manually Configuring Reporting Coverage](#).

## Manually Configuring Reporting Coverage

If you run .Net tests using [NUnit](#), [MSTest](#), [MSpec](#) or [.NET Process Runner](#) runners or run NUnit tests via supported tasks of [MSBuild](#) or [NAnt](#) runners, you can turn on coverage collection in the TeamCity web UI for the specific runner.

For other cases, when the .Net code coverage is collected by the build script and needs to be reported inside TeamCity (for example, [Rake runner](#), or if you run NUnit tests via a test launcher other than TeamCity NUnit Test Launcher), there is a way to let TeamCity know about the coverage data.

First, make sure the build script actually collects the code coverage according to the coverage engine documentation.

Then, report the collected data to TeamCity:

Communication is done via [service messages](#).

First, the build script needs to let TeamCity know details on the coverage engine with the "dotNetCoverage" message.

Then, the build script can issue one or several "importData" messages to import the actual code coverage data files collected. As a result, TeamCity will display coverage statistics and an HTML report for the coverage.

### Configuring Code Coverage Engine

Use the following service message template:

```
#teamcity[dotNetCoverage <key>='<value>' <key1>='<value1>' ...]
```

where **key** is one of the following:

For dotCover (optional):

key	description
dotcover_home	The full path to the dotCover home folder to override the bundled dotCover.

For NCover 3.x:

key	description	sample value
ncover3_home	Full path to NCover installation folder.	The path to the NCover3 installation directory
ncover3_reporter_args	Arguments for the NCover report generator.	Set "//or FullCoverageReport:Html:{teamcity.report.path}" or another NCover commandline argument

For NCover 1.x:

key	description	sample value
ncover_explorer_tool	Path to NCoverExplorer.	Path to NCoverExplorer
ncover_explorer_tool_args	Additional arguments for NCover 1.x.	
ncover_explorer_report_type	Value for /report: argument.	1
ncover_explorer_report_order	Value for /sort: argument.	1

For PartCover:

key	description	value
partcover_report_xslts	Write xslt transformation rules one per line (use  n as separator) in the following format: file.xslt=>generatedFileName.html	file.xslt=>generatedFileName.html

## Importing Coverage Data Files

To pass xml report generated by a coverage tool to TeamCity, in your build script use the following service message:

```
##teamcity[importData type='dotNetCoverage' tool='<tool name>' path='<path to the results file>']
```

where `tool name` can be `dotcover`, `partcover`, `ncover` or `ncover3`.

 For `dotCover` you should send paths to the snapshot file that is generated by the `dotCover.exe cover` command.

## Java Testing Frameworks Support

TeamCity supports JUnit and TestNG by means of following build runners:

- Ant (when tests are run by `junit` and `testng` tasks directly within the script)
- Maven2 (when tests are run by Surefire/Failsafe Maven plugins, on-the-fly reporting is not available.)
- IntelliJ IDEA project: IntelliJ IDEA's JUnit and TestNG run configurations are supported. Note, that such run configurations should be shared and checked in to the version control.

## Configuring Java Code Coverage

TeamCity supports Java code coverage based on the IntelliJ IDEA coverage engine, [EMMA](#) open-source toolkit, and [JaCoCo](#). See details in the dedicated sections:

- [IntelliJ IDEA](#)
- [EMMA](#)
- [JaCoCo](#)

See also:

[Concepts: Code Coverage](#)

[Administrator's Guide: IntelliJ IDEA | EMMA](#)

## IntelliJ IDEA

The IntelliJ IDEA coverage engine in TeamCity is the same engine that is used within IntelliJ IDEA to measure code coverage. This coverage attaches to the JVM as a java agent and instruments classes on the fly when they are loaded by the JVM. In particular that means that classes are not changed on the disk and can be safely used for distribution packages.

The IntelliJ IDEA coverage engine currently supports Class, Method and Line coverage. There is no Branch/Block coverage yet.

 Make sure your tests run in the `fork=true` mode. Otherwise the coverage data may not be properly collected.

To configure code coverage using IntelliJ IDEA engine, follow these steps:

1. While creating/editing Build Configuration, go to the Build Step page.
2. Select the [Ant](#), [IntelliJ IDEA Project](#), [Gradle](#) or [Maven](#) build runner.
3. In the Code Coverage section, select IntelliJ IDEA as a coverage tool in the Choose coverage runner drop-down.
4. Set up the coverage options - refer to the description of the available options below.

Option	Description
Classes to instrument	Specify Java packages for which code coverage will be gathered. Use new-line delimited patterns that start with a valid package name and contain *. For example: <code>org.apache.*</code> .
Classes to exclude from instrumentation	Use newline-separated patterns for fully qualified class names to be excluded from the coverage, for example: <code>*Test</code> . Exclude patterns have priority over include patterns.

See also:

[Concepts: Build Runner | Code Coverage](#)

[Administrator's Guide: Configuring Java Code Coverage | EMMA](#)

## EMMA

[EMMA Integration Notes](#)

The following steps are performed when collecting coverage with EMMA:

1. After each compilation step (with `javac/javac2`), the build agent invokes EMMA to instrument the compiled classes and to store the location of the source files. As a result, the `coverage.em` file containing the classes metadata is created in the build checkout directory. The collected source paths of the java files are used to generate the final HTML report.

 All `coverage.*` files are removed in the beginning of the build, so you have to ensure that full recompilation of sources is performed in the build to have the actual `coverage.em` file.
2. Test run. At this stage, the actual runtime coverage information is collected. This process results in creation of the `coverage.ec` file. If there are several test tasks, data is appended to `coverage.ec`.
3. Report generation. When the build ends, TeamCity generates an HTML coverage report, creates the `coverage.zip` file with the report and uploads it to the server. It also generates and uploads the summary report in the `coverage.txt` file, and the original `coverage.ec(m)` files to allow viewing coverage from the TeamCity plugin for IntelliJ IDEA.

## Configuring Coverage with EMMA

To configure code coverage by means of EMMA engine, follow these steps:

1. While creating/editing Build Configuration, go to the Build Step page.
2. Select `Ant`, or `Ipr` build runner.
3. In the Code Coverage section, choose EMMA as a coverage tool in the drop-down.
4. Set up the coverage options - refer to the description of the available options below.

Option	Description
Include Source Files in the Coverage Data	<p>Check this option to include source files into the code coverage report (you'll be able to see sources on the Web).</p> <p> <b>Warning</b> Enabling this option can increase the report size and may slow down the creation of your builds. To avoid this situation, specify some EMMA properties (see <a href="http://emma.sourceforge.net/reference_single/reference.html#prop-ref.tables">http://emma.sourceforge.net/reference_single/reference.html#prop-ref.tables</a> for details).</p>
Coverage Instrumentation Parameters	<p>Use this field to specify the filters to be used for creating the code coverage report. These filters define classes to be exempted from instrumentation. For detailed description of filters, refer to <a href="http://emma.sourceforge.net/reference_single/reference.html#prop-ref.tables">http://emma.sourceforge.net/reference_single/reference.html#prop-ref.tables</a>.</p>

## Troubleshooting

No coverage, there is a message: EMMA: no output created: metadata is empty

Please make sure that all your classes (whose coverage is to be evaluated) are recompiled during the build. Usually this requires adding a "clean" task at the beginning of the build.

`java.lang.NoClassDefFoundError: com/vladium/emma/rt/RT`

This message appears when your build loads EMMA-instrumented class files in runtime, and it cannot find `emma.jar` file in classpath. For test tasks, like `junit` or `testng`, TeamCity adds `emma.jar` to classpath automatically. But for other tasks, this is not the case and you might need to modify your build script or to exclude some classes from instrumentation.

If your build runs a java task which uses your own compiled classes, you'll have to either add `emma.jar` to the classpath of the java task, or to ensure that classes used in your java task are not instrumented. Besides, you should run your java task with the `fork=true` attribute.

The corresponding `emma.jar` file can be taken from `buildAgent/plugins/coveragePlugin/lib/emma.jar`.

For a typical build, the corresponding include path would be `../../../../plugins/coveragePlugin/lib/emma.jar`

To exclude classes from compilation, use settings for EMMA instrumentation task. TeamCity UI has a field to pass these parameters to EMMA, labeled "Coverage instrumentation parameters". To exclude some package from instrumenting, use the following syntax: `-ix -com.foo.task.*,+com.foo.*,-*Test*`, where the package `com.foo.task.*` contains files for your custom task.

EMMA coverage results are unstable

Please make sure that your `junit` task has the `fork=true` attribute. The recommended combination of attributes is "`fork=true forkmode=once`".

See also:

## JaCoCo

TeamCity supports JaCoCo, a Java Code Coverage tool allowing you to measure a wide set of coverage metrics and code complexity.

JaCoCo is available for the following build runners: Ant, IntelliJ IDEA Project, Gradle and Maven.

 To ensure the coverage data is collected properly, make sure your tests run in (one or more) separate JVMs.

- Ant and IntelliJ Idea Project runners: this is the default setting for [TestNG](#), for [JUnit test task](#), set `fork=true`.
- Maven runner: set `forkCount` to a value [higher than 0](#).
- Gradle runner: this is the default setting for [Gradle tests](#).

On this page:

- [Enabling JaCoco coverage](#)
- [Importing JaCoCo coverage data to TeamCity](#)

### Enabling JaCoco coverage

TeamCity supports the java agent coverage mode allowing you to collect coverage without modifying build scripts or binaries. No additional build steps needed - just choose JaCoCo coverage in a build step which runs tests:

1. In the Code Coverage section, select JaCoCo as a coverage tool in the Choose coverage runner drop-down.
2. Set up the coverage options - refer to the description of the available options below.

Option	Description	Example
Classfile directories or jars	Newline-delimited set of path patterns in the form of + -:[path] to scan for classfiles to be analyzed. Libraries and test classfiles don't have to be listed unless their coverage is wanted.	+ :target/main/java/**
Classes to instrument	Newline-delimited set of classname patterns in the form of + -:[path]. Allows filtering out unwanted classes listed in "Classfile directories or jars" field. Useful in case test classes are compiled.	+ :com.package.core.* - :com.package.*Test*

 By default, in TeamCity the `jacoco.sources` property is set to `". "`, which means that TeamCity will scan whole checkout directory including all subdirectories for your sources. Check that your classfiles are compiled with debug information (including the source file info) to see with highlighted source code in the report.

The code coverage results can be viewed on the Overview tab of the Build Results page; detailed report is displayed on the dedicated Code Coverage tab.

### Importing JaCoCo coverage data to TeamCity

TeamCity can parse JaCoCo coverage data and generate a report using a service message of the following format:

```
# #teamcity[jacocoReport dataPath='<path to jacoco.exec file>']
```

Attribute name	Description	Default value	Example
dataPath	Space-delimited set of paths relative to the checkout directory to read the jacoco data file		<code>jacocoResults/jacoco.exec</code> <code>jacocoResults/anotherJacocoRun.exec</code>
includes	Space-delimited set of classname include patterns	*	<code>com.package.core.*</code> <code>com.package.api.*</code>

excludes	Space-delimited set of classname exclude patterns		com.package.test.* .*Test
sources	Space-delimited set of paths relative to the checkout directory to read sources from. Does not need to be listed by default.	.	src
classpath	Space-delimited set of path patterns in the form of + -:[path] to scan for classfiles to be analyzed. Libraries and test classfiles do not need to be listed unless their coverage is wanted.	+:**/*	+:target/main/java/**
reportDir	Path to the directory to store temporary files. The report will be generated as coverage.zip under this directory. Check that there is no existing directory with the same name.	A random directory under Agent's temp directory	jacocoReport

An example of a complete service message:

```
# #teamcity[jacocoReport dataPath='jacoco.exec' includes='com.package.core.*'
classpath='classes/lib/some.jar' reportDir='temp/jacocoReport']
```

## Running Risk Group Tests First

This section covers:

- Reordering Risk Tests for JUnit and TestNG
  - JUnit
  - TestNG
    - TestNG versions less than 5.14:
    - TestNG versions 5.14 or newer:
- Reordering Risk Tests for NUnit

### Reordering Risk Tests for JUnit and TestNG

Supported environments:

- Ant and IntelliJ IDEA Project runners
- JUnit and TestNG frameworks when tests are started with usual JUnit or TestNG tasks

 TeamCity also allows to implement tests reordering feature for a custom build runner.

You can instruct TeamCity to run some tests before others. You can do this on the build runner settings page. Currently there are two groups of tests that TeamCity can run first:

- recently failed tests, i.e. the tests failed in previous finished or running builds as well as tests having high failure rate (a so called blinking tests)
- new and modified tests, i.e. tests added or modified in changelists included in the running build



The recently added or modified tests in your [personal build](#) have the highest priority, and these tests run even before any other reordered test.

TeamCity allows you to enable both of these groups or each one separately.

TeamCity operates on test case basis, that is not the individual tests are reordered, but the full test cases. The tests cases are reordered only within a single test execution Ant task, so to maximize the feature effect, use a single test execution task per build.

Tests reordering works the following way:

JUnit

1. TeamCity provides tests that should be run first (test classes).

2. When a JUnit task starts, TeamCity checks whether it includes these tests.
3. If at least one test is included, TeamCity generates a new fileset containing included tests only and processes it before all other filesets. It also patches other filesets to exclude tests added to the automatically generated fileset.
4. After that JUnit starts and runs as usual.

 Some cases when automatic tests reordering will not work:

- if JUnit suites are created manually in test cases with help of suite() method
- if @RunWith annotation is used in JUnit4 tests

TestNGTestNG versions less than 5.14:

1. TeamCity provides tests that should be run first (test classes).
2. When a TestNG task starts, TeamCity checks whether it includes these tests.
3. If at least one test is included, TeamCity generates a new xml file with suite containing included tests only and processes it before all other files. It also patches other files to exclude tests added to the automatically generated file.
4. After that TestNG starts and runs as usual.

 Some cases when automatic tests reordering will not work:

- if <package/> element is used in the TestNG XML suite

TestNG versions 5.14 or newer:

1. TeamCity provides tests that should be run first (test classes).
2. When a TestNG starts, TeamCity injects custom listener which will reorder tests if needed.
3. Before starting tests TestNG asks listener to reorder tests execution order list. If some test requires reordering, TeamCity listener moves it to the start of the list.
4. After that TestNG runs tests in new order.

 Some cases when automatic tests reordering will not work:

- if <test/> or <suite/> element in the TestNG XML suite has "preserve-order" attribute set to "true"
- if TestNG suite file in YAML format

Reordering Risk Tests for NUnit

Supported build runners:

- [NAnt](#), [MSBuild](#), [NUnit](#), [Visual Studio 2003](#) build runners
- NUnit testing framework

Tests reordering only supports reordering of recently failed tests.

- 
- When "Run recently failed tests first" option is selected, the `Explicit` attribute will not work.
  - Test reordering is not supported for parametrized NUnit tests.

If risk tests reordering option is enabled, the feature for NUnit test runner works in the following way:

1. NUnit runs tests from the "risk" group using test name filter.
2. NUnit runs all other tests using inverse filter.



- Since tests run twice, thus risk test fixtures Set Up and Tear Down will be performed twice.
- Tests reordering feature applies to an NUnit task. That is, for NAnt and MSBuild runners, tests reordering feature will be initiated as many times as many NUnit tasks you have in your build script.

See also:

## Build Failure Conditions

In TeamCity you can adjust the conditions when a build should be marked as failed in the Failure Conditions section of the of the [Build Configuration Settings](#) page.

 To fail a build if sufficient disk space cannot be freed for the build, see the [Free disk space](#) build feature.

On this page:

- [Common build failure conditions](#)
- [Additional Failure Conditions](#)
  - [Fail build on metric change](#)
    - [Adding custom build metric](#)
  - [Fail build on specific text in build log](#)
  - [Stopping build immediately](#)

### Common build failure conditions

In the Common Failure Conditions, specify how TeamCity will fail builds by selecting appropriate options from in the Fail build if area:

- it runs longer than ... minutes: Enter a value in minutes to enable execution timeout for a build. If the specified amount of time is exceeded, the build is automatically canceled.
  - Unless set to 0, this build configuration setting overrides the [server-wide](#) default execution timeout specified in Administration -> Global settings.
  - The default value of 0 means that no limit is set by the build configuration. If there is a [server-wide](#) default execution timeout, this default will be used.This option helps to deal with builds that hang and maintains agent efficiency.
- the build process exit code is not zero: Check this option to mark the build as failed if the build process doesn't exit successfully.
- at least one test failed: Check this option to mark the build as failed if the build fails at least one test. If this option is not checked, the build can be marked successful even if it fails to pass a number of tests. Regardless of this option, TeamCity will run all build steps.
- an error message is logged by build runner: Check this option to mark the build as failed if the build runner reports an error while building.
- an out of memory or crash is detected (Java only): Check this option to mark the build as failed if a crash of the JVM is detected, or Java out of memory problems. If possible, TeamCity will upload crash logs and memory dumps as artifacts for such builds.

### Additional Failure Conditions

You can instruct TeamCity to mark a build as failed if some of its metrics has deteriorated in comparison with another build,e.g. code coverage, artifacts size, etc. For instance, you can mark build as failed if the code duplicates number is higher than in the previous build.

Another build failure condition causes TeamCity to mark build as failed when a certain text is present in the build log.

To add such build failure condition to your build configuration, click Add build failure condition and select from the list:

- [Fail build on metric change](#)
- [Fail build on specific text in build log](#)

 You can disable a build failure condition temporarily or permanently at any time, even if it is inherited from a build configuration template.

### Fail build on metric change

When using code examining tools in your build, like code coverage, duplicates finders, inspections and so on, your build generates various numeric metrics. For these metrics you can specify a threshold which, when exceeded, will fail a build.

In general there are two ways to configure this build fail condition:

- A build metric exceeds or is less than the specified constant value (threshold).  
e.g.: Fail build if build duration (secs) compared to constant value is more than\* 300. In this case a build will fail if it runs more than 300 seconds.
- A build metric has changed comparing to a specific build by a specified value.  
e.g.: Fail build if its build duration (secs) compared to a value from another build is more by at least 30 0 default units for this metric than the value in the Last successful build. In this case a build will fail if it runs 300 seconds longer than the last successful build. If [Branch specification](#) is configured, then [the following logic](#) is applied.

Values from the following builds can be used as the basis for comparing build metrics:

- last successful build
- last pinned build
- last finished build
- build with specified build number
- last finished build with specified tag.

By default, TeamCity provides the wide range of build metrics:

- artifacts size(bytes) - size of artifacts excluding [internal artifacts](#) under .teamcity directory
- build duration (secs)
- number of classes
- number of code duplicated
- number of covered classes
- number of covered lines
- number of covered methods
- number of failed tests
- number of ignored tests
- number of inspection errors
- number of inspection warnings
- number of lines of code
- number of methods
- number of passed tests
- number of tests
- percentage of block coverage
- percentage of class coverage
- percentage of line coverage
- percentage of method coverage
- percentage of statement coverage
- test duration (secs)
- total artifacts size (bytes) - size of all artifacts including [internal ones](#)

 Note that since TeamCity 9.0, the way TeamCity counts tests [has changed](#).  
Adding custom build metric

You can add your own build metric. To do so, you need to modify the TeamCity configuration file [<TeamCity Data Directory>/config/main-config.xml](#) and add the following section under "server" node there:

```
<build-metrics>
  <statisticValue key="myMetric" description="build metric for number of files"/>
</build-metrics>
```

So, if your build publishes the `myMetric` value, you can use it as a criterion for a build failure.

Fail build on specific text in build log

TeamCity can inspect all lines in build log for some particular text occurrence that indicates a build failure. Lines are matched without the time and block name prefixes which precede each message in the build log representation.

To configure this build failure condition, specify:

- a string or a [Java Regular Expression](#) whose presence/absence in the build log is an indicator of a build failure,
- a failure message to be displayed on the build results page when a build fails due to this condition.

Stopping build immediately

Since TeamCity 2017.1, you can now stop a build immediately on encountering a specified text in the build log or when a certain build metric, specified using the Fail build on metric change condition, is exceeded.

## Configuring Build Triggers

Once a build configuration is created, builds can be triggered manually by clicking the [Run button](#) or initiated automatically with the help of Triggers.

A build trigger is a rule which initiates a new build on certain events. The build is put into the [build queue](#) and is started when there are agents available to run it.

While creating/editing a build configuration, you can configure triggers using the Triggers sections of the Build Configuration Settings page by clicking Add new trigger and specifying the trigger settings. For configuration details on each trigger, refer to the corresponding sections. It is possible to disable a configured build trigger temporarily or permanently using the option in the last column of the Triggers list.

For each build configuration the following triggers can be configured:

- [VCS trigger](#): the build is triggered when changes are detected in the version control system roots attached to the build configuration.
- [Schedule trigger](#): the build is triggered at a specified time.
- [Finish Build trigger](#): the build is triggered after a build of the selected configuration is finished.
- [Maven Artifact Dependency trigger](#): the build is triggered if there is a content modification of the specified Maven artifact which is detected by the checksum change.
- [Maven Snapshot Dependency trigger](#): the build is triggered if there is a modification of the snapshot dependency content in the remote repository which is detected by the checksum change.
- [Retry build trigger](#): the build is triggered if the last build failed or failed to start.
- [Branch Remote Run Trigger](#): personal build is triggered automatically each time TeamCity detects new changes in particular branches of the VCS roots of the build configuration. Supports Git and Mercurial.
- [NuGet Dependency Trigger](#): starts a build if there is a NuGet package update detected in the NuGet repository.



Note that if you create a build configuration from a template, it inherits build triggers defined in the template, and

they cannot be edited or deleted. However, you can specify additional triggers or disable a trigger permanently or temporarily.

In addition to the triggers defined for the build configuration, you can also trigger a build by an [HTTP GET request](#), or manually by running a custom build.

### Configuring VCS Triggers

VCS triggers automatically start a new build each time TeamCity detects new changes in the configured [VCS roots](#). Only one VCS trigger can be added to a build configuration.

A new VCS trigger with the default settings triggers a build each time new changes are detected: the version control is polled for changes according to the [Checking for changes interval](#) of a VCS root honouring a [VCS commit hook](#) if configured. Newly detected changes appear as Pending Changes of a build configuration. If several check-ins are made during this time, only one build will be triggered. If you have several VCS roots attached to a build configuration, TeamCity will add the build to the queue only after the longest of the specified intervals.

After the last change is detected, there is a [quiet period](#) before the build is started.

The global default value for both options is 60 seconds and can be configured for the server on the [Administration | Global Settings](#) page.

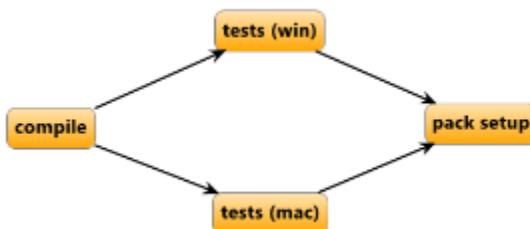
You can adjust a VCS trigger to your needs using the options described below:

- Trigger a build on changes in snapshot dependencies
- Per-check-in Triggering
- Quiet Period Settings
- Build Queue Optimization Settings
- VCS Trigger Rules
  - Trigger Rules Examples
  - General Syntax
- Branch Filter
- Trigger Rules and Branch Filter Combined
- Triggering a Build on Branch Merge

#### Trigger a build on changes in snapshot dependencies

If you have a [build chain](#) (i.e. a number of builds interconnected by [snapshot dependencies](#)), the triggers are to be configured in the final build in the chain. This is pack setup in the image below.

Let's take a build chain from the example: `pack setup--depends on--tests--depends on--compile`.



With the VCS Trigger set up in the `pack setup` configuration, the whole build chain is usually triggered when TeamCity detects changes in `pack setup`; changes in `compile` will trigger `compile` only and not the whole chain. If you want the whole chain to be triggered on a VCS change in `compile`, add a VCS trigger with the Trigger on changes in snapshot dependencies option enabled to the final build configuration of the chain, `pack setup`.

This will not change the order in which builds are executed, but will only trigger the whole build chain, if there is a change in any of snapshot dependencies. In this setup, no VCS triggers are required for the `compile` or `tests` build configurations.

If triggering rules are specified (described [below](#)), they are applied to all the changes (including changes from snapshot dependencies) and only the changes matching the rules trigger the build chain.

See also details at the [Build Dependencies](#) page.

#### Per-check-in Triggering

When this option is not enabled, several check-ins by different committers can be made; and once they are detected, TeamCity will add only one build to the queue with all of these changes.

If you have fast builds and enough build agents, you can make TeamCity launch a new build for each check-in ensuring that no other changes get into the same build. To do that, select the Trigger a build on each check-in option. If you select the Include

several check-ins in build if they are from the same committer option, and TeamCity will detect a number of pending changes, it will group them by user and start builds having single user changes only.

This helps to figure out whose change broke a build or caused a new test failure, should such issue arise.

### Quiet Period Settings

By specifying the quiet period you can ensure the build is not triggered in the middle of non-atomic check-ins consisting of several VCS check-ins.

A quiet period is a period (in seconds) that TeamCity maintains between the moment the last VCS change is detected and a build is added into the queue. If new VCS change is detected in the Build Configuration within the period, the period starts over from the new change detection time. The build is added into the queue only if there were no new VCS changes detected within the quiet period. Note that the actual quiet period will not be less than the maximum [Checking for changes interval](#) among the VCS roots of a build configuration, as TeamCity must ensure that changes were collected at least once during the quiet period.

The quiet period can be set to the default value (60 seconds, can be changed globally at the Administration | Global Settings page) or to a custom value for a build configuration.

**!** Note that when a build is triggered by a trigger with the VCS quiet period set, the build is put into the queue with fixed VCS revisions. This ensures the build will be started with only the specific changes included. Under certain circumstances this build can later become a [History Build](#).

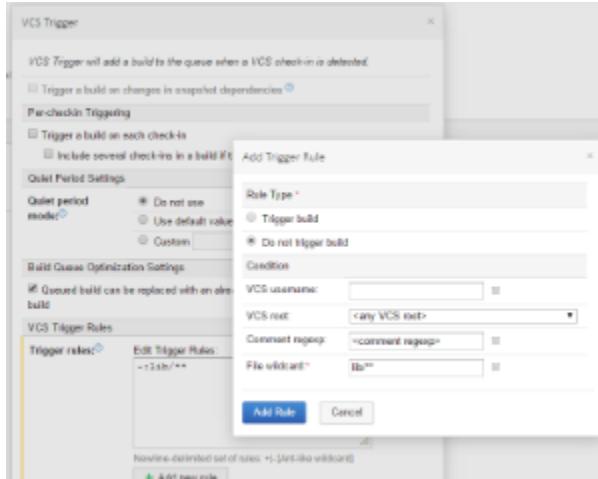
### Build Queue Optimization Settings

By default, TeamCity [optimizes the build queue](#): already queued build can be replaced with an already started build or a more recent queued build. To disable the default behavior, uncheck the box.

### VCS Trigger Rules

If no trigger rules specified, a build is triggered upon any detected change displayed for the build configuration. You can affect the changes detected by changing the VCS root settings and specifying [Checkout Rules](#).

To limit the changes that trigger the build, use the VCS trigger rules. You can add these rules manually in the text area (one per line), or use the Add new rule option to generate them.



Each rule is either an "include" (starts with "+") or an "exclude" (starts with "-").

### Trigger Rules Examples

```
+:.
-:**.html
-:user=techwriter;root=InternalSVN:/misc/doc/*.xml
-:lib/**
-:comment=minor:**
-:comment=^oops$:**
```

- `+:` includes all files
- `"-:**.html"` excludes all `.html` files from triggering a build.
- `"-:user=techwriter;root=InternalSVN:/misc/doc/*.xml"` excludes builds being triggered by `.xml` files checked in by the **VCS user** "tech writer" to the `misc/doc` directory of the VCS root named Internal SVN (as defined in the VCS Settings). Note that the path is absolute (starts with `/`), thus the file path is matched from the VCS root.
- `"-:lib/**"` prevents the build from triggering by updates to the "lib" directory of the build sources (as it appears on the agent). Note that the path is relative, so all files placed into the directory (by processing VCS root checkout rules) will not cause the build to be triggered.
- `"-:comment=minor:**"` prevents the build from triggering, if the changes check in comment contains word "minor".
- `"-:comment=^oops$:**"` no triggering if the comment consists of the only word "oops" (according to [Java Regular Expression](#) principle s `^` and `$` in pattern stand for string beginning and ending)

## General Syntax

The general syntax for a single rule is:

```
+|-[[:user=VCS_username;][root=VCS_root_id;][comment=VCS_comment_regexp]]:Ant_like_wildca
rd
```

Where:

- **Ant\_like\_wildcard** - A [wildcard](#) to match the changed file path. Only `"*"` and `"**"` patterns are supported, the `"?"` pattern is not supported. The file paths in the rule can be relative (not started with `'` or `'\'`) to match resulting paths on the agent or absolute (started with `'/'`) to match VCS paths relative to a VCS root. For each file in a change the most specific rule is found (the rule matching the longest file path). The build is triggered if there is at least one file with a matching "include" rule or a file with no matching "exclude" rules.
- **VCS\_username** - if specified, limits the rule only to the changes made by a user with the corresponding [VCS username](#).
- **VCS\_root\_id** - if specified, limits the rule only to the changes from the corresponding VCS root.
- **VCS\_comment\_regexp** - if specified, limits the rule only to the changes that contain specified text in the VCS comment. Use the [Java Regular Expression](#) pattern for matching the text in a comment (see examples below). The rule matches if the comment text contains a matched text portion; to match the entire text, include the `^` and `$` special characters.

**i** When entering rules, please note that as soon as you enter any "+" rule, TeamCity will remove the default "include all" setting. To include all the files, use `+:"` rule.

Also, rules are sorted according to path specificity. I.e. if you have an explicit inclusion rule for `/some/path`, and exclusion rule `-:user=some_user:.` for all paths, commits to the `/some/path` from `some_user` will be included unless you add a specific exclusion rule for this user and this path at once, like `-:user=some_user:/some/path/**`

## Branch Filter

When a VCS Root has branches [configured](#), the Branch filter setting appears in the trigger options.

The Branch filter setting limits a set of [logical branch names](#) (the branch names as they appear in the TeamCity UI) which trigger should apply to. The branch filter uses format and precedence similar to the [branch specification](#) (but it matches logical branch name and uses no parenthesis).

Details of the format:

```
+:logical branch name  
-:logical branch name
```

where `logical branch name` is the part of the branch name matched by the branch specification (i.e. displayed for a build in TeamCity UI), see [Working With Feature Branches](#).

It is possible to use the wildcard placeholder ('`*`') which matches one or more characters: `+|-:name*` will match the branch '`name1`' but will not match the branch '`name`', which will need to be added explicitly.

Other examples:

Only the default branch is accepted:

```
+:<default>
```

All branches except the default one are accepted:

```
+:  
-:<default>
```

Only branches with the `feature-` prefix are accepted:

```
+:feature-*
```

By default, the branch filter is set to `+:  
*`, which is the equivalent of an empty branch filter. In this case that a build will be triggered on every branch tracked by the branch specification.

If several rules match a single branch, the most specific (least characters matched by pattern) last rule apply.

#### Trigger Rules and Branch Filter Combined

Trigger rules and branch filter are combined by AND, which means that the build is triggered only when both conditions are satisfied.

For example, if you specify a comment text in the trigger rules field and provide the branch specification, the build will be triggered only if a commit has the special text and is also in a branch matched by branch filter.

#### Triggering a Build on Branch Merge

The VCS trigger is fully aware of branches and will trigger a build once a check-in is detected in a branch.

When changes are merged / fast-forwarded from one branch to another, strictly speaking there are no actual changes in the code. By default, the VCS trigger behaves in the following way:

- When merging/fast forwarding of two non-default branches: the changes in a build are calculated with regard to previous builds in the same branch, so if there is a build on same commit in a different branch, the trigger will start a build in another branch pointing to the same commit.
- If the default branch is one of the branches in the merging/fast-forwarding, the changes are always calculated against the default branch, if there is a build on same revision in the default branch, TeamCity will not run a new build on the same revision.

## Branch Remote Run Trigger

Branch Remote Run trigger automatically starts a new [Personal Build](#) each time TeamCity detects changes in particular branches of the VCS roots of the build configuration.

At the moment this trigger supports only Git and Mercurial VCSes.

For non-personal builds off branches, please see [Working with Feature Branches](#). When branch specification is configured for a VCS root, Branch Remote Run Trigger only processes branches not matched by the specification.

A trigger monitors branches with names that match specific patterns. Default patterns are:

for Git repositories — `refs/heads/remote-run/*`  
for Mercurial repositories — `remote-run/*`

These branches are regular version control branches and TeamCity does not manage them (i.e. if you no longer need the branch you would need to delete the branch using regular version control means).

By default TeamCity triggers a personal build for the user detected in the last commit of the branch. You might also specify TeamCity user in the name of the branch. To do that use a placeholder `TEAMCITY_USERNAME` in the pattern and your TeamCity username in the name of the branch, for example pattern `remote-run/TEAMCITY_USERNAME/*` will match a branch `remote-run/joe/my_feature` and start a personal build for the TeamCity user `joe` (if such user exists).



### Troubleshooting

At the moment there is no UI to show what's going on inside the trigger, so the only way to troubleshoot it is to look inside `teamcity-remote-run.log`. To see a more detailed log please enable `debug-vcs` logging preset at [Administration](#) | [Diagnostics](#) page.

In order to trigger a build branch should have at least one new commit comparing to the main branch.

Example: Run a personal build from a command line.

Git

```
% cd <your local git repo>
% git branch
* master
% git checkout -b my_feature
Switched to a new branch 'my_feature'
//code, commit; code, commit
% git push origin +HEAD:remote-run/my_feature
```

With the default pattern (`refs/heads/remote-run/*`) command `git branch -r` will list your personal branches. If you want to hide them, change the pattern to `refs/remote-run/*` and push your changes to branches like `refs/remote-run/my_feature`. In this case your branches are not listed by the above command, although you can see them anyway using `git ls-remote <url of git repository>`.

Mercurial

```
% cd <your local hg repo>
% hg branch
default
% hg branch remote-run/my_feature
marked working directory as branch remote-run/my_feature
//code, commit; code, commit
% hg push -b remote-run/my_feature --new-branch
```

## Limitations

If your build configuration has more than one VCS root which support branch remote-run, and you push changes to all of them, TeamCity will start several personal builds with changes from one VCS root each.

## See also:

[Administrator's Guide: Git | Mercurial](#)

## Configuring Schedule Triggers

The Schedule Trigger allows you to set the time when a build of the configuration will be run. The [Builds Schedule](#) page of the current project settings displays the configured build times. More than one Schedule trigger can be added to a build configuration.

On this page:

- [Triggering Conditions](#)
  - [Date and Time](#)
    - [Examples](#)
    - [Brief description of the cron format used](#)
  - [VCS Changes](#)
  - [VCS Trigger Rules](#)
    - [Trigger Rules Examples](#)
    - [General Syntax](#)
  - [Build Changes](#)
- [Additional Options](#)
  - [Enforce Clean Checkout](#)
  - [Trigger Build on All Enabled and Compatible Agents](#)
  - [Build Queue Optimization Settings](#)
  - [Branch Filter](#)
  - [Trigger Rules and Branch Filter Combined](#)

### Triggering Conditions

The settings in this section define time and other conditions for automatic build triggering. You can schedule a recurring build or set a specific date and time for it.

#### Date and Time

In addition to triggering builds daily or weekly at a specified time for a particular time zone, you can specify advanced time settings using [cron-like expressions](#). This format provides more flexible scheduling options.

TeamCity uses [Quartz](#) for working with cron expressions. See the examples below or consider using the [CronMaker](#) utility to generate expressions based on Quartz cron format.

#### Examples

	Each 2 hours at :30	Every day at 11:45PM	Every Sunday at 1:00AM	Every last day of month at 10:00AM and 10:00PM
Seconds	0	0	0	0
Minutes	30	45	0	0
Hours	0/2	23	1	10,22
Day-of-month	*	*	?	L

Month	*	*	*	*
Day-of-week	?	?	1	?
Year(Optional)	*	*	*	*

See also other examples.

Brief description of the cron format used

Cron expressions are comprised of six fields and one optional field separated with a white space. The fields are respectively described as follows:

Field Name	Values	Special Characters
Seconds	0-59	, - * /
Minutes	0-59	, - * /
Hours	0-23	, - * /
Day-of-month	1-31	, - * ? / L W
Month	1-12 of JAN-DEC	, - * /
Day-of-week	1-7 or SUN-SAT	, - * ? / L #
Year(Optional)	empty, 1970-2099	, - * /

For the description of the special characters, please refer to [Quartz CronTrigger Tutorial](#).

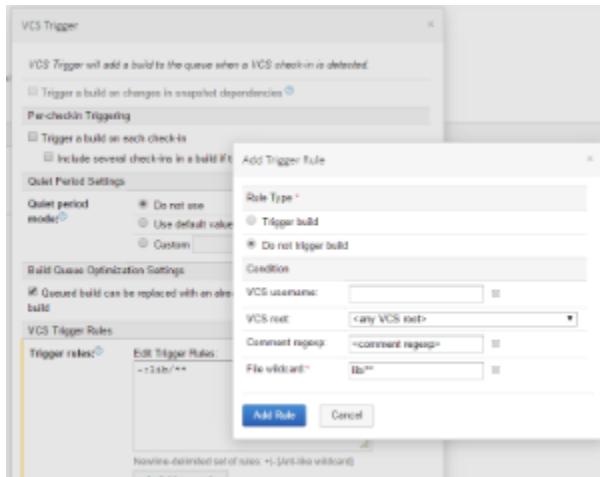
#### VCS Changes

You can restrict schedule trigger to start builds only if there are pending changes in your version control by selecting the corresponding option. The Trigger only if there are pending changes option considers newly detected pending changes only: if there were pending changes before the trigger was created, the build is not triggered.

#### VCS Trigger Rules

If no trigger rules specified, a build is triggered upon any detected change displayed for the build configuration. You can affect the changes detected by changing the VCS root settings and specifying [Checkout Rules](#).

To limit the changes that trigger the build, use the VCS trigger rules. You can add these rules manually in the text area (one per line), or use the Add new rule option to generate them.



Each rule is either an "include" (starts with "+") or an "exclude" (starts with "-").

#### Trigger Rules Examples

```
+:.
-:**.html
-:user=techwriter;root=InternalSVN:/misc/doc/*.xml
-:lib/**
-:comment=minor:**
-:comment=^oops$:**
```

- `+:. includes all files`
- `"-:**.html"` excludes all `.html` files from triggering a build.
- `"-:user=techwriter;root=InternalSVN:/misc/doc/*.xml"` excludes builds being triggered by `.xml` files checked in by the **VCS user** "tech writer" to the `misc/doc` directory of the VCS root named Internal SVN (as defined in the VCS Settings). Note that the path is absolute (starts with `/`), thus the file path is matched from the VCS root.
- `"-:lib/**"` prevents the build from triggering by updates to the "lib" directory of the build sources (as it appears on the agent). Note that the path is relative, so all files placed into the directory (by processing VCS root checkout rules) will not cause the build to be triggered.
- `"-:comment=minor:**"` prevents the build from triggering, if the changes check in comment contains word "minor".
- `"-:comment=^oops$:**"` no triggering if the comment consists of the only word "oops" (according to [Java Regular Expression](#) principle s `^` and `$` in pattern stand for string beginning and ending)

## General Syntax

The general syntax for a single rule is:

```
+|-[[:user=VCS_username;][root=VCS_root_id;][comment=VCS_comment_regexp]]:Ant_like_wildca
rd
```

Where:

- **Ant\_like\_wildcard** - A [wildcard](#) to match the changed file path. Only `"*"` and `"**"` patterns are supported, the `"?"` pattern is not supported. The file paths in the rule can be relative (not started with `'` or `'\'`) to match resulting paths on the agent or absolute (started with `'/'`) to match VCS paths relative to a VCS root. For each file in a change the most specific rule is found (the rule matching the longest file path). The build is triggered if there is at least one file with a matching "include" rule or a file with no matching "exclude" rules.
- **VCS\_username** - if specified, limits the rule only to the changes made by a user with the corresponding [VCS username](#).
- **VCS\_root\_id** - if specified, limits the rule only to the changes from the corresponding VCS root.
- **VCS\_comment\_regexp** - if specified, limits the rule only to the changes that contain specified text in the VCS comment. Use the [Java Regular Expression](#) pattern for matching the text in a comment (see examples below). The rule matches if the comment text contains a matched text portion; to match the entire text, include the `^` and `$` special characters.

**i** When entering rules, please note that as soon as you enter any "+" rule, TeamCity will remove the default "include all" setting. To include all the files, use "+:." rule.

Also, rules are sorted according to path specificity. I.e. if you have an explicit inclusion rule for `/some/path`, and exclusion rule `-:user=some_user:.` for all paths, commits to the `/some/path` from `some_user` will be included unless you add a specific exclusion rule for this user and this path at once, like `-:user=some_user:/some/path/**`

## Build Changes

The Schedule Trigger can watch a build in a different other build configuration and trigger a build if the watched build changes.

In the Build Changes section, select the corresponding box and specify the build configuration and the type of build to watch: last successful build, last pinned build, last finished build, or the last finished build with a specified tag.

TeamCity can [promote](#) the watched build if there is a dependency (snapshot or artifact) on its build configuration.

## Additional Options

### Enforce Clean Checkout

It is possible to force TeamCity to clean all files in the checkout directory before a build. This option can also be applied to snapshot dependencies. In this case, all the builds of the build chain will be forced to use [clean checkout](#). The option also enables rebuilding all dependencies (unless custom dependencies are provided via the custom build dialog or the schedule trigger promotes a build).

#### Trigger Build on All Enabled and Compatible Agents

Use this option to run a build simultaneously on all agents that are enabled and compatible with the build configuration. This option may be useful in the following cases:

- run a build for agent maintenance purposes (e.g. you can create a configuration to check whether agents function properly after an environment upgrade/update).
- run a build on different platforms (for example, you can set up a configuration, and specify for it a number of compatible build agents with different environments installed).

#### Build Queue Optimization Settings

By default, TeamCity [optimizes the build queue](#): already queued build can be replaced with an already started build or a more recent queued build. To disable the default behavior, uncheck the box.

#### Branch Filter

When a VCS Root has branches [configured](#), the Branch filter setting appears in the trigger options.

The Branch filter setting limits a set of [logical branch names](#) (the branch names as they appear in the TeamCity UI) which trigger should apply to. The branch filter uses format and precedence similar to the [branch specification](#) (but it matches logical branch name and uses no parenthesis).

Details of the format:

```
+:logical branch name  
-:logical branch name
```

where [logical branch name](#) is the part of the branch name matched by the branch specification (i.e. displayed for a build in TeamCity UI), see [Working With Feature Branches](#).

It is possible to use the wildcard placeholder ('\*') which matches one or more characters: +|-:[name\\*](#) will match the branch 'name1' but will not match the branch 'name', which will need to be added explicitly.

Other examples:

Only the default branch is accepted:

```
+:<default>
```

All branches except the default one are accepted:

```
+:  
-:<default>
```

Only branches with the `feature-` prefix are accepted:

```
+:feature-*
```

By default, the branch filter is set to `+:  
-`, which is the equivalent of an empty branch filter. In this case that a build will be triggered on every branch tracked by the branch specification.

If several rules match a single branch, the most specific (least characters matched by pattern) last rule apply.

The branch filter in the Schedule Trigger works as follows:

- if the option trigger build only if there are pending changes is turned ON, then the trigger will add a build to the queue for all branches matched by the trigger branch filter where pending changes exist
- if trigger build only if there are pending changes is turned OFF, then the trigger will add a build to the queue for all branches matched by the trigger branch filter regardless of presence of pending changes there

#### Trigger Rules and Branch Filter Combined

Trigger rules and branch filter are combined by AND, which means that the build is triggered only when both conditions are satisfied.

For example, if you specify a comment text in the trigger rules field and provide the branch specification, the build will be triggered only if a commit has the special text and is also in a branch matched by branch filter.

## Builds Schedule

The Builds Schedule page in the administration area of a specific project displays [schedule triggers](#) configured for [build configurations](#) belonging to this project.

Builds Schedule for the [Root Project](#) displays the list of triggers for the entire TeamCity server.

You can conveniently view the available schedule and plan your builds optimizing allocation of dependent hardware/software resources.

From this page it is also possible to [edit](#), disable or delete the triggers.

The Build Schedule page also displays the information for [paused build configurations](#).

#### Configuring Maven Triggers

The Triggers page of the Build Configuration Settings allows you to add the following Maven dependency triggers:

- [Maven Snapshot Dependency Trigger](#)
- [Maven Artifact Dependency Trigger](#)
  - Advanced Options
  - Version Ranges

#### Checksum Based Triggering

The trigger checks if the content of the dependency has actually changed by verifying its checksum from the repository against the locally stored version. Before triggering a build, TeamCity tries to determine the checksum of the required dependency by downloading the file digest (MD5/SHA-1) associated with that artifact.

If the checksum can be retrieved, and it matches a locally stored one, no build is triggered. If the checksum is different, a build is triggered.

If the checksum cannot be retrieved from the remote server, the dependency will be downloaded, TeamCity will calculate its checksum and follow the build triggering mechanism described above.

#### Maven Snapshot Dependency Trigger

Maven snapshot dependency trigger adds a new build to the queue when there is a real modification of the snapshot dependency content in the remote repository which is detected by the checksum change.

Dependency artifacts are resolved according to the POM and the server-side [Maven Settings](#).

**⚠** Note that since Maven deploys artifacts to remote repositories sequentially during a build, not all artifacts may be up-to-date at the moment the snapshot dependency trigger detects the first updated artifact. To avoid inconsistency, select the Do not trigger a build if currently running builds can produce snapshot dependencies check box when adding this trigger, which will ensure the build won't start while builds producing snapshot dependencies are still running.

**⚠** Simultaneous usage of snapshot dependency and dependency trigger for a build  
Assume build A depends on build B by both snapshot and trigger dependency. Then, after the build B finishes, build A will be added into the queue, only if build B is not a part of the build chain containing A.

### Maven Artifact Dependency Trigger

Maven artifact dependency trigger adds build to the queue when there is a real modification of the dependency content which is detected by the checksum change.

To add a trigger, specify the following parameters in the Add New Trigger dialog:

Parameter	Description
Group Id	Specify an identifier of a group the desired Maven artifact belongs to.
Artifact Id	Specify the artifact's identifier.
Version or Version range	Specify a version or version range of the artifact. The version range syntax is described in the <a href="#">section</a> below. SNAPSHOT versions can also be used.
Type	Define explicitly the type of the specified artifact. By default, the type is <code>jar</code> .
Classifier	(Optional) Specify the classifier of an artifact.
Maven repository URL	Specify a URL to the Maven repository. Note that this parameter is optional. If the URL is not specified, then: <ul style="list-style-type: none"> <li>For a Maven project the repository URL is determined from the POM and the server-side <a href="#">Maven Settings</a></li> <li>For a non-Maven project the repository URL is determined from the server-side <a href="#">Maven Settings</a> only</li> </ul>
Do not trigger a build if currently running builds can produce this artifact	Select this option to trigger a build only after the build that produces artifacts used here is finished.

### Advanced Options

Since TeamCity 9.0, the following advanced options have been added to the trigger:

Parameter	Description
Repository ID	Allows using authorization from the effective Maven settings
User settings selection	Allows selecting effective settings. The same as <a href="#">User Settings</a> of the Maven runner.

TeamCity determines the effective repository to be checked for the artifact updates and to trigger builds if changes are detected as follows:

- if a URL and Repository ID are set, authentication will be chosen from the effective settings (see below)
- if only a URL is set, the old behavior is preserved: a temporary repository ID is used ("`_tc_temp_remote_repo`")
- if URL is not set (regardless of the Repository ID), the artifact will be looked up in a repository available according to the effective settings.

TeamCity determines effective settings as follows:

- in the trigger settings a user can choose among the default, custom or uploaded Maven settings. See [Maven Server-Side Settings](#) for details.
- if no specific settings are configured for the trigger, [Maven](#) build step settings are used
- if no settings for the trigger are configured and there are no Maven build steps, the default [server Maven settings](#) will be used.

### Version Ranges

For specifying version ranges use the following syntax, as [proposed in the Maven documentation](#).

Note that Maven Artifact Dependency Trigger can be used not only for fixed-version artifacts but also for snapshots as a fine-grained alternative to the Maven Snapshots Dependency Trigger.

Range	Meaning
( ,1.0]	$x \leq 1.0$
1.0	"Soft" requirement on 1.0 (just a recommendation - helps select the correct version if it matches all ranges)
[1.0]	Hard requirement on 1.0
[1.2,1.3]	$1.2 \leq x \leq 1.3$
[1.0,2.0)	$1.0 \leq x < 2.0$
[1.5,)	$x \geq 1.5$
( ,1.0],[1.2,)	$x \leq 1.0$ or $x \geq 1.2$ . Multiple sets are comma-separated
( ,1.1),(1.1,)	This excludes 1.1, if it is known not to work in combination with this library
1.0-SNAPSHOT	The trigger will check the latest snapshot version for updates

## NuGet Dependency Trigger

The NuGet Dependency Trigger allows starting a new build if a NuGet packages update is detected in the NuGet repository.

### Requirements and limitations

For a TeamCity server running on Windows, .NET 4.0 is required.

For a TeamCity server running on Linux, the NuGet dependency trigger will reportedly work with the following limitations:

- filtering by Package Version Spec is not supported
- only HTTP package sources are supported
- NuGet feed version 1.0 is used, so case-sensitivity issues might occur
- the current trigger implementation on Linux might increase the server load
- authentication issues might occur

### Configuring NuGet Dependency Trigger

1. Select the NuGet version to use from the NuGet.exe drop-down list (if you have [installed NuGet beforehand](#)), or specify a custom path to `NuGet.exe`;
2. Specify the NuGet package source, if it is different from `nuget.org`;
3. Specify the credentials to access NuGet feed if required
4. Enter the package Id to check for updates.
5. Optionally, you can specify [package version range](#) to check for. If not specified, TeamCity will check for latest version.

You can also opt to trigger build if pre-release package version is detected by selecting corresponding check box. Note that this is only supported for NuGet version 1.8 or newer.

### See also:

[Administrator's Guide: NuGet](#)

## Configuring Finish Build Trigger

The Finish build trigger triggers a build of the current build configuration when a build of the selected build configuration is finished. If the Trigger after successful build only checkbox is enabled, a build is triggered only after a successful build of the selected configuration.

To monitor builds in other build configurations and trigger a build if these builds change, please see [this option](#) of the Schedule build trigger.



In most of the cases the Finish Build Trigger should be used with snapshot dependencies, i.e. the current build configuration where the trigger is defined should have a direct or an indirect snapshot dependency on the build

configuration selected in the trigger. If there is no snapshot dependency, the following limitations exist:

- it is likely that a build of the build configuration being triggered will not have the same revisions as the finished build even if both configurations have the same VCS settings
- if a build configuration with the Finish Build Trigger has an artifact dependency on the last finished build of the build configuration specified in the trigger settings, there is no guarantee that artifacts of a build which caused build triggering will be used, because, while the triggered build sits in the build queue, another build may finish
- the build triggered by the Finish Build Trigger will always be triggered in the default branch even if the finished build has some other branch

All these limitations do not apply if a build configuration with "Finish build trigger" has a snapshot dependency on the selected build configuration. In this case, the trigger will run build on the same revisions and will attach the build to the chain. It will also use consistent artifacts if they are produced by dependencies.

In a build configuration with branches, use the filter described below to limit the branches where finished builds will trigger new builds of the current configuration.

#### Branch Filter

When a VCS Root has branches [configured](#), the Branch filter setting appears in the trigger options.

The Branch filter setting limits a set of [logical branch names](#) (the branch names as they appear in the TeamCity UI) which trigger should apply to. The branch filter uses format and precedence similar to the [branch specification](#) (but it matches logical branch name and uses no parenthesis).

Details of the format:

```
+:logical branch name  
-:logical branch name
```

where `logical branch name` is the part of the branch name matched by the branch specification (i.e. displayed for a build in TeamCity UI), see [Working With Feature Branches](#).

It is possible to use the wildcard placeholder ('\*') which matches one or more characters: `+|-:name*` will match the branch 'name' but will not match the branch 'name', which will need to be added explicitly.

Other examples:

Only the default branch is accepted:

```
+:<default>
```

All branches except the default one are accepted:

```
+:  
-:<default>
```

Only branches with the `feature-` prefix are accepted:

```
+:feature-*
```

By default, the branch filter is set to `+:  
*`, which is the equivalent of an empty branch filter. In this case that a build will be triggered on every branch tracked by the branch specification.

If several rules match a single branch, the most specific (least characters matched by pattern) last rule apply.

## Configuring Dependencies

A build configuration can be made dependent on the artifacts or sources of builds of some other build configurations.

For [snapshot dependencies](#), TeamCity will run all dependent builds on the sources taken at the moment the build they depend on starts.

For [artifact dependencies](#), before a build is started, all artifacts this build depends on will be downloaded and placed in their configured target locations and then will be used by the build.

 The dependencies of the build can later be viewed on the build results page - the Dependencies tab. This tab also displays indirect dependencies, e.g. if a build A depends on a build B which depends on builds C and D, then these builds C and D are indirect dependencies for build A.

See also:

[Concepts: Dependent Build](#)

[Administrator's Guide: Accessing artifacts via HTTP | Snapshot Dependencies | Artifact Dependencies](#)

[External Resources: http://ant.apache.org/ivy/](#) (additional information on Ivy)

## Artifact Dependencies

This page details configuration of the TeamCity [Artifact Dependencies](#).

The Build Configuration Settings | Dependencies page, Artifact Dependencies section allows configuring the dependencies. It is possible to disable a configured dependency temporarily or permanently using the corresponding option in the last column of the Artifact Dependencies list.

- [Configuring Artifact Dependencies Using Web UI](#)
- [Configuring Artifact Dependencies Using Ant Build Script](#)
- [Build-level authentication](#)

### Configuring Artifact Dependencies Using Web UI

To add an artifact dependency to a build configuration:

1. When [creating/editing a build configuration](#), open the Dependencies page.
2. Click the Add new artifact dependency link and specify the following settings:

Option	Description
Depend on	Specify the build configuration for the current build configuration to depend on. A dependency can be configured on a previous build of the same build configuration
Get artifacts from	Specify the type of build whose artifacts are to be taken: last successful build, last pinned build, last finished build, build from the same chain (this option is useful when you have a snapshot dependency and want to obtain artifacts from a build with the same sources), build with a specific build number or the last finished build with a specified tag.   <ul style="list-style-type: none"><li>When selecting the build configuration, take your <a href="#">clean-up policy settings</a> into account. Builds are cleaned and deleted on a regular basis, thus the build configuration could become dependent on a non-existent build. When artifacts are taken from a build with a specific number, then the specific build will not be deleted during clean-up.</li><li>If both dependency by sources and dependency by artifacts on the last finished build are configured for a build configuration, then artifacts will be taken from the build with the same sources.</li></ul>
Build number	This field appears if you have selected build with specific build number in the Get artifacts from list. Specify here the exact <a href="#">build number</a> of the artifact.
Build tag	This field appears if you have selected last finished build with specified tag in the Get artifacts from list. Specify here the tag of the build whose artifacts are to be used. When resolving the dependency, TeamCity will look for the last successful build with the given tag and use its artifacts.
Build branch	This field appears if the dependency has a <a href="#">branch specified</a> in the VCS Root settings. Allows setting a <a href="#">branch</a> to limit source builds only to those with the branch. If not specified, the default branch is used. The logic branch name (shown in the UI for the build) is to be used. Patterns are not supported.
Artifacts Rules	A newline-delimited set of rules. Each rule must have the following syntax:

```
[+:-]SourcePath[!ArchivePath][=>DestinationPath]
```

Each rule specifies the files to be downloaded from the "source" build. The SourcePath should be relative to the artifacts directory of the "source" build. The path can either identify a specific file, directory, or use wildcards to match multiple files. [Ant-like wildcards](#) are supported.

Downloaded artifacts will keep the "source" directory structure starting with the first \* or ?.

DestinationPath specifies the destination directory on the agent where downloaded artifacts are to be placed. If the path is relative (which is recommended), it will be resolved against the build checkout directory. If needed, the destination directories can be cleaned before downloading artifacts. If the destination path is empty, artifacts will be downloaded directly to the checkout root.

#### Basic examples:

- Use `a/b/**=>lib` to download all files from `a/b` directory of the source build to the `lib` directory. If there is a `a/b/c/file.txt` file in the source build artifacts, it will be downloaded into the file `lib/c/file.txt`.
- At the same time, artifact dependency `**/*.*txt=>lib` will preserve the directories structure: the `a/b/c/file.txt` file from source build artifacts will be downloaded to `lib/a/b/c/file.txt`.

ArchivePath is used to extract downloaded [compressed](#) artifacts. Zip, 7-zip, jar, tar and tar.gz are supported. ArchivePath follows general rules for SourcePath: ant-like wildcards are allowed, the files matched inside the archive will be placed in the directory corresponding to the first wildcard match (relative to destination path)

For example: `release.zip!*.*.dll` command will extract all .dll files residing in the root of the `release.zip` artifact.

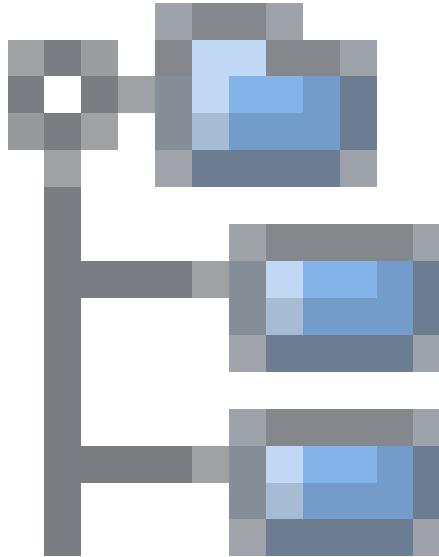
#### Archive processing examples:

- `release-*.*.zip!*.*.dll=>d1ls` will extract \*.dll from all archives matching the `release-*.*.zip` pattern to the `d1ls` directory.
- `a.zip/**=>destination` will unpack the entire archive saving the path information.
- `a.zip!/a/b/c/**/*.*.dll=>d1ls` will extract all .dll files from `a/b/c` and its subdirectories into the `d1ls` directory, without the `a/b/c` prefix.

`:+` and `-:` can be used to include or exclude specific files from download or unpacking. As `:+` prefix can be omitted: rules are inclusive by default, and at least one inclusive rule is required. The order of rules is unimportant. For each artifact the most specific rule (the one with the longest prefix before the first wildcard symbol) is applied. When excluding a file, DestinationPath is ignored: the file won't be downloaded at all. Files can also be excluded from archive unpacking. The set of rules applied to the archive content is determined by the set of rules matched by the archive itself.

### Exclusive patterns examples:

- `**/* .txt=>texts`  
`-:bad/exclude.txt`  
Will download all \*.txt files from all directories, excluding exclude.txt from the bad directory
- `+:release-* .zip!**/* .dll=>dlls`  
`-:release-0.0.1.zip!Bad.dll`  
Will download and unpack all dlls from release-\* .zip files to the dlls directory. The Bad.dll file from release-0.0.1.zip will be skipped
- `**/*.*=>target`  
`-:excl/**/*.*`  
`+:excl/must_have.txt=>target`  
Will download all artifacts to the target directory. Will not download anything from the excl directory, but the file called must\_have.txt



Click the  icon to invoke the Artifact Browser. TeamCity will try to locate artifacts according to the specified settings and show them in a tree. Select the required artifacts from the tree and TeamCity will place the paths to them into the input field.

The artifacts placed under the .teamcity directory are considered [hidden](#). These artifacts are ignored by wildcards by default.

If you want to include files from the .teamcity directory for any purpose, be sure to add the artifact path starting with .teamcity explicitly.

### Example of accessing hidden artifacts:

- `.teamcity/properties/* .properties`
- `.teamcity/*.*`

Clean destination paths before downloading artifacts	Check this option to delete the content of the destination directories before copying artifacts. It will be applied to all inclusive rules
--	--

At any point you can launch a build with [custom artifact dependencies](#).

#### Configuring Artifact Dependencies Using Ant Build Script

This section describes how to download TeamCity build artifacts inside the build script. These instructions can also be used to download artifacts from outside of TeamCity.

To handle artifact dependencies between builds, this solution is more complicated than configuring dependencies in the TeamCity UI but allows for greater flexibility. For example, managing dependencies this way will allow you to start a personal build and verify that your build is still compatible with dependencies.

To configure dependencies via Ant build script:

1. Download Ivy.

 TeamCity itself acts as an Ivy repository. You can read more about the Ivy dependency manager here: <http://ant.apache.org/ivy/>.

2. Add Ivy to the classpath of your build.

3. Create the `ivyconf.xml` file that contains some meta information about TeamCity repository. This file is to have the following content:

```
<ivysettings>
<property name='ivy.checksums' value="/" />
<caches defaultCache="${teamcity.build.tempDir}/.ivy/cache"/>
<statuses>
  <status name='integration' integration='true'/>
</statuses>
<resolvers>
  <url name='teamcity-rep' alwaysCheckExactRevision='yes' checkmodified='true'>
    <ivy
      pattern='http://YOUR_TEAMCITY_HOST_NAME/httpAuth/repository/download/[module]/[revision]/teamcity-ivy.xml' />
    <artifact
      pattern='http://YOUR_TEAMCITY_HOST_NAME/httpAuth/repository/download/[module]/[revision]/[artifact]([.ext])' />
  </url>
</resolvers>
<modules>
  <module organisation='.*' name='.*' matcher='regexp' resolver='teamcity-rep' />
</modules>
</ivysettings>
```

4. Replace `YOUR_TEAMCITY_HOST_NAME` with the host name of your TeamCity server.

5. Place `ivyconf.xml` in the directory where your `build.xml` will be running.

6. In the same directory create the `ivy.xml` file defining which artifacts to download and where to put them, for example:

```

<ivy-module version="1.3">
  <info organisation="YOUR_ORGANIZATION" module="YOUR_MODULE"/>
  <dependencies>
    <dependency org="org" name="BUILD_TYPE_EXT_ID" rev="BUILD_REVISION">
      <include name="ARTIFACT_FILE_NAME_WITHOUT_EXTENSION"
ext="ARTIFACT_FILE_NAME_EXTENSION" matcher="exactOrRegexp"/>
    </dependency>
  </dependencies>
</ivy-module>

```

Where:

- YOUR\_ORGANIZATION replace with the name of your organization.
- YOUR\_MODULE replace with the name of your project or module where artifacts will be used.
- BUILD\_TYPE\_EXT\_ID replace with the [external ID](#) of the build configuration whose artifacts are downloaded.
- BUILD\_REVISION can be either a build number or one of the following strings:
  - latest.lastFinished
  - latest.lastSuccessful
  - latest.lastPinned
  - TAG\_NAME.tcbuildtag - last build tagged with the TAG\_NAME tag
- ARTIFACT\_FILE\_NAME\_WITHOUT\_EXTENSION file name or regular expression of the artifact without the extension part.
- ARTIFACT\_FILE\_NAME\_EXTENSION the extension part of the artifact file name.

7. Modify your `build.xml` file and add tasks for downloading artifacts, for example (applicable for Ant 1.6 and later):

```

<target name="fetchArtifacts" description="Retrieves artifacts for TeamCity"
xmlns:ivy="antlib:org.apache.ivy.ant">
  <taskdef uri="antlib:org.apache.ivy.ant" resource="org/apache/ivy/ant/antlib.xml"/>
  <classpath>
    <pathelement location="${basedir}/lib/ivy-2.0.jar"/>
    <pathelement location="${basedir}/lib/commons-httpclient-3.0.1.jar"/>
    <pathelement location="${basedir}/lib/commons-logging.jar"/>
    <pathelement location="${basedir}/lib/commons-codec-1.3.jar"/>
  </classpath>
  </taskdef>
  <ivy:configure file="${basedir}/ivyconf.xml" />
  <!--<ivy:cleancache />-->
  <ivy:retrieve pattern ="${basedir}/[artifact].[ext]"/>
</target>

```

-  • commons-httpclient, commons-logging and commons-codec are to be in the classpath of Ivy tasks.  
• To clean the Ivy cache directory before retrieving dependencies, uncomment the `<ivy:cleancache />` element in the example above.

Artifacts repository is protected by a basic authentication. To access the artifacts, you need to provide credentials to the `<ivy:configure/>` task. For example:

```

<ivy:configure file ="${basedir}/ivyconf.xml"
  host="TEAMCITY_HOST"
  realm="TeamCity"
  username="USER_ID"
  passwd="PASSWORD"/>

```

where `TEAMCITY_HOST` is hostname or IP address of your TeamCity server (without port and servlet context). As `USER_ID/PASSWORD` you can use either username/password of a regular TeamCity user (the user should have corresponding permissions to access artifacts of the source build configuration) or system properties `teamcity.auth.userId/teamcity.auth`

.password.

#### Build-level authentication

The system properties `teamcity.auth.userId` and `teamcity.auth.password` store automatically generated build-unique values which can be used to authenticate on TeamCity server. The values are valid only during the time the build is running. This generated user has limited permissions which allow build-related operations. The primary intent for the user is to use the authentication to download artifacts from other TeamCity builds within the build script.

Using the properties is preferable to using real user credentials since it allows the server to track the artifacts downloaded by your build. If the artifacts were downloaded by the build configuration artifact dependencies or using the supplied properties, the specific artifacts used by the build will be displayed at the Dependencies tab on the build results page. In addition, the builds which were used to get the artifacts from, can be configured to have different clean-up logic.

See also:

[Concepts: Dependent Build](#)

#### Snapshot Dependencies

By setting a [snapshot dependency](#) of a build (e.g. build B) on other build's (build A's) sources, you can ensure that build B will start only after the one it depends on (build A) is run and finished. We call build A a dependency build, whereas build B is a dependent build.

The Dependencies page of the build configuration settings displays the configured dependencies and since TeamCity 2017.1, the Snapshot dependencies section of the page allows previewing the build chain and its configuration. The preview shows

builds of the chain; the builds with automatic triggering configured are marked with the  icon:

When adding a new snapshot dependency, the following options need to be specified:

Option	Description
Depend on	Specify the build configuration for the current build configuration to depend on.

Do not run new build if there is a suitable one	<p>If the option is enabled, TeamCity will not run a dependency build, if another running or finished dependency build with the appropriate sources revision exists. See also <a href="#">#Suitable Builds</a> below.</p> <p>However, when a dependent build is triggered, the dependency build will also be put into the queue. Then, when the changes for the build chain are collected, this dependency build will be removed from the queue and the dependency will be set to a suitable finished build.</p> <div style="border: 1px solid #f0c987; padding: 10px; margin-top: 10px;">  Note: if there is more than one snapshot dependency on some build configuration, then for builds reusing to work, all of them must have the "Do not run new build if there is a suitable one" option enabled.         </div>
Only use successful builds from suitable ones	A new triggered build will only "use" successfully finished suitable builds as dependencies. If the latest finished "suitable" build is failed, it will be re-run.
Run build on the same agent	<p>When enabled, and B snapshot-depends on A, then builds of B are run on the same agent where the build of A from the same build chain was run.</p> <div style="border: 1px solid #f0c987; padding: 10px; margin-top: 10px;">  Before starting a build chain having run on the same agent dependencies, TeamCity forms groups of builds combined by run on the same agent dependency, and for each starting build participating in such a group TeamCity chooses agents which can be used by any build of the group. Thus this option makes sense for <a href="#">composite builds</a> too, even though composite build does not occupy an agent, it still can form a group of builds combined by such dependencies. For instance, composite build B having run on the same agent dependencies on A and C will cause both A and C use the same agent.         </div>
On failed dependency/ On failed to start/canceled dependency	<p>If a dependency fails, you can manage the status of the dependent build by selecting one of the following options:</p> <ul style="list-style-type: none"> <li>• Run build, but add problem: the dependent build will be run and the problem will be added to it, changing its status to failed (if problem was not muted earlier)</li> <li>• Run build, but do not add problem: the dependent build will be run and no problems will be added</li> <li>• Make build failed to start: the dependent build will not run and will be marked as "Failed to start"</li> <li>• Cancel build: the dependent build will not run and will be marked as "Canceled".</li> </ul>

## Suitable Builds

A "suitable" build in terms of snapshot dependencies is a build which can be used instead a queued dependency build within a [build chain](#). That is, a queued build which is a part of a build chain can be dropped and the builds depending on it can be made dependent on another queued, running or already finished "suitable" build. This behavior only works when the Do not run new build if there is a suitable one option of a corresponding snapshot dependency is selected.

For a build to be considered "suitable", it should comply with all of the conditions below:

- use the same sources snapshot as the entire queued build chain being processed. If the build configurations have the same VCS settings, this basically means the one with the same sources revision. If the VCS settings are different (VCS roots or checkout rules), then "same sources snapshot" revisions means revisions taken simultaneously at some moment in time.
- be successful (if "Only use successful builds from suitable ones" snapshot dependency option is set)
- be a usual, not a [personal build](#)
- have no customized parameters (see also [TW-23700](#))
- have no VCS settings preventing effective revision calculation, see [below](#)
- there is no other build configuration snapshot-depending on the current one with "Do not run new build if there is a suitable one" option set to "off"
- the running build is not "hanging"
- settings of the build configuration were not changed since the build (that is, the build was run with the current build configuration settings). This also includes no changes to the parameters of all the parent projects of the build configuration. You can check if the settings were changed between several builds by comparing `.teamcity/settings/digest.txt` file in the [hidden build's artifacts](#)
- if there is also an artifact dependency in addition to snapshot one, the suitable build should have artifacts
- all the dependency builds (the builds the current one depends on) are "suitable" and are appropriately merged

Some settings in VCS roots can effectively disable builds reusing. These settings are:

- Subversion: Checkout, but ignore changes mode
- CVS: Checkout by tag mode
- Perforce: Stream or Client connection settings, or label is specified in Label/revision to checkout option
- Starteam: checkout mode option set to view label or promotion date

See also:

[Concepts: Dependent Build](#)

## Build Dependencies Setup

This page is intended to give you the general idea on how dependencies work in TeamCity based on an example. For the dependencies description, please see [Dependent Build](#).

In this section:

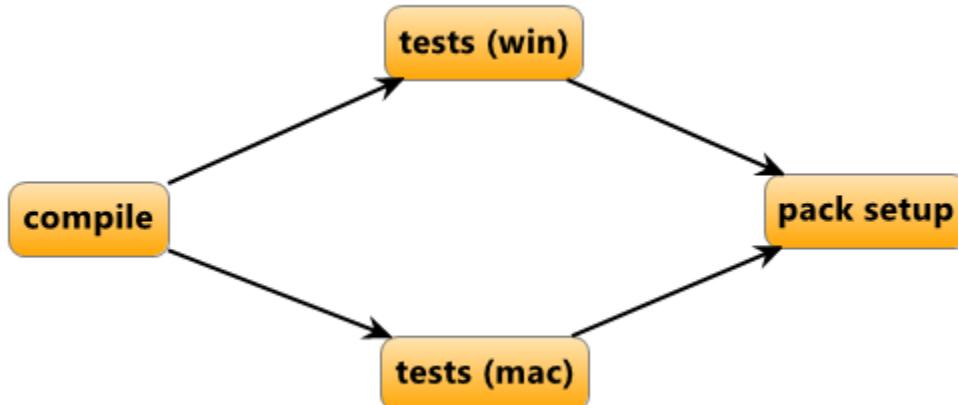
- [Introduction](#)
- [Basics](#)
- [Artifact Dependencies](#)
- [Snapshot Dependencies](#)
  - When to Create Build Chain
  - Build Chains in TeamCity UI
  - How Snapshot Dependencies Work
  - Example 1
    - What Happens When Build A is Triggered
    - What Happens When Build B is Triggered
  - Example 2
  - Advanced Snapshot Dependencies Setup
    - Reusing builds
    - Run build on the same agent
    - Build behavior if dependency has failed
    - Trigger on changes in snapshot dependencies
    - Parameters in dependent builds
- [Miscellaneous Notes on Using Dependencies](#)

### Introduction

In many cases it is convenient to use the output of one build in another, as well as to run a number of builds sequentially on the same sources. Consider a typical example: you have a cross-platform project that has to be tested under Windows and macOS before you get the production build. The best workflow for this simple case will be to:

1. Compile your project
2. Run tests under Windows and macOS simultaneously on the same sources
3. Build a release version on the same sources, of course, if tests have passed under both OSs.

This can be easily achieved by configuring dependencies between your build configurations in TeamCity that would look like this:



Where `compile`, `tests (win)`, `tests (mac)` and `pack setup` are build configurations, and naturally the tests depend on the compilation, which means they should wait till the compilation is ready.

## Basics

Generally known as the "build pipeline", in TeamCity a similar concept is referred to as a "[build chain](#)". Before getting into details on how this works in TeamCity, let's clarify the legend behind diagrams given here (including the one in the introduction):

	A build configuration.
	<p><b>Snapshot dependency</b> between 2 build configurations. Note that the arrow shows the sequence of triggering build configurations, the <a href="#">build chain</a> flow, meaning that B is executed before A. However, the dependencies are configured in the opposite direction (A snapshot-depends on B). The arrows are drawn this way because <a href="#">in the TeamCity UI</a> you can find the visual representation of build chains which are always displayed according to the build chain flow.</p> <p>Typically, when adding a snapshot dependency, you also add an artifact dependency with the "build form the same chain" option from the same configuration to transfer the previous build results and use them in the build as well.</p>
	<p><b>Artifact dependency</b>. The arrow shows the artifacts flow, the dependency is configured in the opposite direction.</p>

As you noticed, there are 2 types of dependencies in TeamCity: artifact dependencies and snapshot dependencies. In two words, the first one allows using the output of one build in another, while the second one can trigger builds from several build configurations in a specific order, but on the same sources.

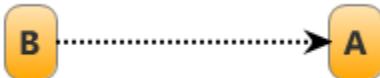
These two dependencies are often configured together because an artifact dependency doesn't affect the way builds are triggered, while a snapshot dependency itself doesn't reuse artifacts, and sometimes you may need only one of those.

Dependencies are configured on the dedicated page of a build configuration settings.

Now, let's see what you can do with artifact and snapshot dependencies, and how exactly they work.

### Artifact Dependencies

An artifact dependency allows reusing the output of one build (or a part of it) in another.



If build configuration A has an artifact dependency on B, then the artifacts of B are downloaded to a build agent before a build of A starts. Note that you can flexibly adjust [artifact rules](#) to configure which artifacts should be taken and where exactly they should be placed.

If for some reason you need to store artifact dependency information together with your codebase and not in TeamCity, you can configure [Ivy Ant tasks](#) to get the artifacts in your build script.

If both snapshot and artifact dependency are configured, and the Build from the same chain option is selected in the artifact dependency settings, TeamCity ensures that artifacts are downloaded from the same-sources build.

## Snapshot Dependencies

A snapshot dependency is a dependency between two build configurations that allows launching builds from both build configurations in a specific order and ensure they use the same sources snapshot (sources revisions corresponding to the same moment).

When you have a number of builds interconnected by snapshot dependencies, they form a [build chain](#).

### When to Create Build Chain

The most common use case for creating a [build chain](#) is running the same test suite of your project on different platforms. For example, you may need to have a release build and want to make sure the tests run correctly on different platforms and environments. For this purpose, you can instruct TeamCity to run an integration build first, and after that to run a release build, if the integration one was successful.

Another case is when your tests take too long to run, so you have to extract them into a separate build configuration, but you also need to make sure they use the same sources snapshot.

### Build Chains in TeamCity UI

Once you have snapshot dependencies defined and at least one [build chain](#) was triggered, the Build Chains tab appears on the Project home page and on the home pages of the related build configurations, providing a visual representation of all build chains and a way to re-run any chain step manually, using the same set of sources pulled originally.

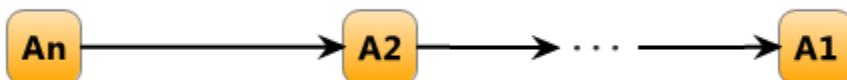


[Learn more](#)

### How Snapshot Dependencies Work

To get an idea of how snapshot dependencies work, think of module dependencies, because these concepts are similar. However, let's start with the basics.

Let's assume, we have a [build chain](#):



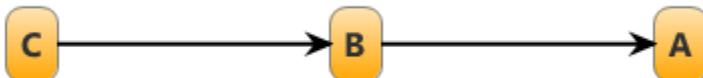
Here are the main rules:

1. If a build of A1 is triggered, the whole build chain A1...AN is added to the [build queue](#), but not vice versa! - if build AN is triggered, it doesn't affect anyhow the build chain, only AN is run.
2. Builds run sequentially starting from AN to A1. Build A(k-1) won't start until build Ak finishes successfully.
3. All builds in the chain will use the same sources snapshot, i.e. with explicit specification of the sources revision, that is calculated at the moment when the build chain is added to the queue.

Now let's go into details and examples.

#### Example 1

Let's assume we have the following [build chain](#) with no extra options - plain snapshot dependencies.



#### What Happens When Build A is Triggered

1. TeamCity resolves the whole build chain and queues all builds - A, B and C. TeamCity knows that the builds are to run in a strict order, so it won't run build A until build B is successfully finished, and it won't run build B until build C is successfully finished.
2. When the builds are added to the queue, TeamCity starts checking for changes in the entire build chain and synchronizes them - all builds have to start with the same sources snapshot.

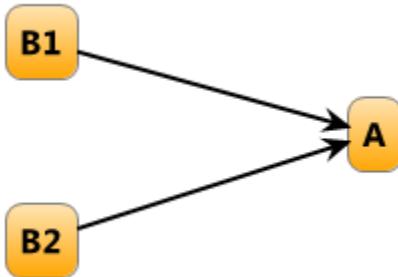
**i** Note that if the build configurations connected with a snapshot dependency share the same set of VCS roots, all builds will run on the same sources. Otherwise, if the VCS roots are different, changes in the VCS will correspond to the same moment in time.

- Once build C has finished, build B starts, and so on. If build C failed, TeamCity won't further execute builds from the chain by default, but this behavior is [configurable](#).

What Happens When Build B Is Triggered

The same process will take place for build chain B->C. Build A won't be affected and won't run.

Example 2



When the final build A is triggered, TeamCity resolves the build chain and queues all builds - A, B1 and B2. Build A won't start until both B1 and B2 are ready.

In this case it doesn't matter which build - B1 or B2 - starts first. As in the first example, when all builds are added to the queue, TeamCity checks for changes in the entire build chain and synchronizes them.

Advanced Snapshot Dependencies Setup

Reusing builds

All builds belonging to the [build chain](#) are placed in the [queue](#). But, instead of enforcing the run of all builds from a build chain, TeamCity can check whether there are already "suitable" builds, i.e. finished builds that used the required sources snapshot. The matching queued builds will not be run and will be [dropped from the queue](#), and TeamCity will link the dependency to the "suitable" builds. To enable this, select "Do not run new build if there is a suitable one" when configuring snapshot dependency options.

Another option that allows you to control how builds are re-used is called "Only use successful builds from suitable ones" and it may help when there's a suitable build, but it isn't successful. Normally, when there's a failed build in a chain, TeamCity doesn't proceed with the rest of the chain. However, with this option enabled, TeamCity will run this failed build on these sources one more time. When is this helpful? For example, when the build failure was caused by a problem when connecting to a VCS.

Run build on the same agent

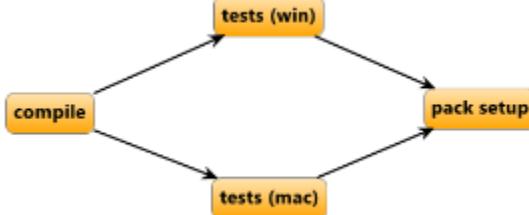
This option was designed for the cases when a build from the build chain modifies system environment, and the next build relies on that system state and thus has to run on the same build agent.

Build behavior if dependency has failed

It is possible to [configure](#) the final build behavior if its dependency has failed.

Trigger on changes in snapshot dependencies

The VCS build trigger has another [option](#) that alters triggering behavior for a build chain. With this options enabled, the whole build chain will be triggered even if changes are detected in dependencies, not in the final build. Let's take a build chain from the example: pack setup--depends on--tests--depends on--compile.



With the VCS Trigger set up in the `pack setup` configuration, the whole build chain is usually triggered when TeamCity detects changes in `pack setup`; changes in `compile` will trigger `compile` only and not the whole chain. If you want the whole chain to be triggered on a VCS change in `compile`, add a VCS trigger with the Trigger on changes in snapshot dependencies op

tion enabled to the final build configuration of the chain, pack setup.

This will not change the order in which builds are executed, but will only trigger the whole build chain, if there is a change in any of snapshot dependencies. In this setup, no VCS triggers are required for the compile or tests build configurations.

#### Changes from Dependencies

For a build configuration with snapshot dependencies, you can enable showing of changes from these dependencies transitively. The setting is called "Show changes from snapshot dependencies" and is available in the advanced options of the "Version Control Settings" step of the build configuration administration pages.

Enabling this setting affects pending changes of a build configuration, builds changes in builds history, the change log and issue log. Changes from dependencies are marked with . For example:

dmitry.neverov	
(jetbrains.git) TW-17252	handle git caches correctly  11 files
(mercurial) Wording	2 files
(mercurial) TW-17252	handle mercurial caches correctly  6 files
Pavel Sher (pavel.sher)	
(perforce) fix changelog styles	2 files
(perforce) disable title + newline	3 files
(perforce) fix bug which reproduces after server restart + test	2 files

With this setting enabled, the Schedule Trigger with a "Trigger build only if there are pending changes" option will consider changes from dependencies too.

#### Parameters in dependent builds

TeamCity provides the ability to use properties provided by the builds the current build depends on (via a snapshot or artifact dependency). When build A depends on build B, you can pass properties from build B to build A, i.e. properties can be passed only in the direction of the build chain flow and not vice versa.

For the details on how to use parameters of the previous build in chain, refer to the [Dependencies Properties page](#).

#### Miscellaneous Notes on Using Dependencies

##### Build chain and clean-up

By default, TeamCity preserves builds that are a part of a chain from clean-up, but you can switch off the option. Refer to the [Clean-Up](#) description for more details.

##### Artifact dependency and clean-up

Artifacts may not be [cleaned](#) if they were downloaded by other builds and these builds are not yet cleaned up. For a build configuration with configured artifact dependencies, you can specify whether the artifacts downloaded by this configuration from other builds can be cleaned or not. This setting is available on the [cleanup policies](#) page.

#### Configuring Build Parameters

Build Parameters provide you with flexible means of sharing settings and a convenient way of passing settings into the build.

On this page:

- [Types of Build Parameters](#)
- [Parameter name restrictions](#)
- [Defining Build Parameters in Build Configuration](#)
- [Using Build Parameters in Build Configuration Settings](#)
  - [Where References Can Be Used](#)
- [Using Build Parameters in VCS Labeling Pattern and Build Number](#)
- [Using Build Parameters in the Build Scripts](#)

#### Types of Build Parameters

Build parameters are name-value pairs, defined by a user or provided by TeamCity, which can be used in a build.

There are three types of build parameters:

- Environment variables (defined using "env." prefix) are passed into the spawned build process as environment
- System properties (defined using "system." prefix) are passed into the build scripts of the supported runners (e.g. Ant, MSBuild) as build-tool specific variables
- Configuration parameters (no prefix) are not passed into the build and are only meant to share settings within a build configuration. They are the primary means for customizing a build configuration which is based on a [template](#) or uses a [meta-runner](#).

There is a set of [predefined parameters](#) provided by TeamCity and administrators can also add custom parameters.

The parameters can be defined at different levels (in order of precedence):

- a specific build (via [Run Custom Build](#) dialog)
- Build Configuration settings (the Parameters page of Build Configuration settings) or [Build Configuration Template](#)
- Project settings (the Parameters page of the Project settings). These affect all the Build Configurations and Templates of the project and its subprojects.
- an agent (the `<Agent home>/conf/buildAgent.properties` file on the agent)

Any textual setting can reference a parameter which makes the string in the format of `%parameter.name%` be substituted with the actual value at the time of build.

If there is a reference to a parameter which is not defined, it is considered an [implicit agent requirement](#) so the build will only run on the agents with the parameter defined.

### Parameter name restrictions

Since 2018.1, the name of a configuration parameter should satisfy the following requirements:

- it should only contain the following characters: [a-zA-Z0-9.\_-\*]
- it should not start with numbers

TeamCity will show warning on edit parameters page if a parameter name does not satisfy these requirements.

Also, since 2018.1 references to parameters with names not satisfying the above restrictions, no longer create an [implicit requirement](#).

### Defining Build Parameters in Build Configuration

On the Parameters page of Build Configuration settings you can define the required system properties and environment variables to be passed to the build script and environment when a build is started. Note that you can redefine them when launching a [Custom Build](#).

Build Parameters defined in a Build Configuration are used only within this configuration. For other ways, refer to [Project and Agent Level Build Parameters](#).

Any user-defined build parameter (system property or environment variable) can reference other parameters by using the following format:

```
%[env|system].property_name%
For example: system.tomcat.libs=%env.CATALINA_HOME%/lib/*.jar
```

### Using Build Parameters in Build Configuration Settings

In most Build Configuration settings you can use a reference to a Build Parameter instead of using the actual value. Before starting a build, TeamCity resolves all references with the available parameters. If there are references that cannot be resolved, they are left as is and a warning will appear in the build log.

To make a reference to a build parameter, use its name enclosed in percentage signs, e.g.: `%teamcity.build.number%`

Any text appearing between percentage signs is considered a reference to a property by TeamCity. If the property cannot be found in the build configuration, the reference becomes an [implicit agent requirement](#) and such build configuration can only be run on an agent with the property defined. The agent-defined value will be used in the build.

If you want to prevent TeamCity from treating the text in the percentage signs as reference to a property, use two percentage signs. Every occurrence of "%%" in the values where property references are supported will be replaced to "%" before passing the value to the build. e.g. if you want to pass "%Y%m%d%H%M%S" into the build, change it to "%%Y%%m%%d%%H%%M%%S"

### Where References Can Be Used

Group of settings	References notes
-------------------	------------------

Build Runner settings, artifact specification	any of the properties that are passed into the build
User-defined properties and Environment variables	any of the properties that are passed into the build
Build Number format	only <a href="#">Predefined Server Build Properties</a>
VCS root and checkout rules settings	any of the properties that are passed into the build
VCS label pattern	system.build.number and <a href="#">Server Build Predefined Properties</a>
Artifact dependency settings	only <a href="#">Predefined Server Build Properties</a>

If you reference a build parameter in a build configuration, and it is not defined there, it becomes an [agent requirement](#) for the configuration. The build configuration will be run only on agents that have this property defined.

Password fields can also contain references to parameters, though in this case you cannot see the reference as it is masked as any password value.

For details on using and overriding parameters from dependencies, please refer to [this section](#).

## Using Build Parameters in VCS Labeling Pattern and Build Number

In Build number pattern and VCS labeling pattern, you can use `%[env|system].property_name%` syntax to reference the properties that are known on the server-side. These are [server](#) and [reference](#) predefined properties and properties defined in the settings of the build configuration on the [Parameters](#) page.

For example, VCS revision number: `%build.vcs.number%`.

## Using Build Parameters in the Build Scripts

All build parameters starting with "env." prefix (environment variables) are passed into the build's process environment (omitting the prefix).

All build parameters starting with "system." prefix (system properties) are passed to the supported build script engines and can be referenced there just by the property name (without "system." prefix):

- For [Ant](#), [Maven](#) and [NAnt](#) use  `${<property name>}`
- For [MSBuild](#) (Visual Studio 2005/2008 Project Files) use  `$(<property name>)`. Note that MSBuild does not support names with dots ("."), so you need to replace "." with "\_" when using the property inside a build script.
- For [Gradle](#): the TeamCity system properties can be accessed as Gradle properties (similar to the ones defined in the `gradle.properties` file) and are to be referenced as follows:
  - a) name allowed as Groovy identifier (the property name does not contain dots):

```
println "Custom user property value is ${customUserProperty}"
```

b) name not allowed as Groovy identifier (the property name contains dots, e.g. `build.vcs.number.1`): `project.ext["build.vcs.number.1"]`



Using `teamcity["<property name>"]` is not recommended, although still supported.

When TeamCity starts a build process, the following precedence of the build parameters is used (those on top have higher priority):

- parameters from the `teamcity.default.properties` file.
- [pre-defined](#) parameters.
- parameters defined in the Run Custom Build dialog.
- parameters defined in the Build Configuration.
- parameters defined in the Project (the parameters defined for a project will be inherited by all its subprojects and build configurations. If required, you can redefine them in a build configuration).
- parameters defined in a template (if any).
- parameters defined in the agent's `buildAgent.properties` file.
- environment variables of the Build Agent process itself.

The resultant set of parameters is also saved into a file which can be accessed by the build script. See `teamcity.build.properties.file` system property or `TEAMCITY_BUILD_PROPERTIES_FILE` environment variable description in [Predefined Build Parameters#Agent Build Properties](#) for details.

See also:

[Administrator's Guide: Project and Agent Level Build Parameters](#) | [Predefined Build Parameters](#) | [Configuring Agent Requirements](#)

## Predefined Build Parameters

TeamCity provides a number of [build parameters](#) which are ready to be used in the settings of a build configuration or in build scripts.

On this page:

- [Server Build Properties](#)
- [Configuration Parameters](#)
  - [Dependencies Properties](#)
    - [Overriding Dependencies Properties](#)
  - [VCS Properties](#)
  - [Branch-Related Parameters](#)
  - [Other Parameters](#)
- [Agent Properties](#)
- [Agent Environment Variables](#)
  - [Java Home Directories](#)
    - [Detecting Java on Agent](#)
      - [Defining Custom directory to Search for Java](#)
      - [Defining Java-related Environment Variables](#)
- [Agent Build Properties](#)

The predefined build parameters can originate from several scopes:

- [#Server Build Properties](#) - the parameters generated by TeamCity on the server-side in the scope of a particular build. An example of such property is a build number.
- [#Agent Properties](#) - the parameters provided by an agent on connection to the server. The parameters are not specific to any build and characterize the agent environment (for example, the path to .Net framework). These are mainly used in [agent requirements](#).
- [#Agent Build Properties](#) - the parameters provided on the agent side in the scope of a particular build right before the build start. For example, a path to a file with a list of changed files.

All these parameters are finally passed to the build.

There is also a special kind of server-side build parameters that can be used in references while defining other parameters, but which are not passed into the build. See [Configuration Parameters](#) below for the list of such properties.



The most up-to-date list of parameters can be obtained in the TeamCity web UI while defining a text value supporting parameters: either click on icon to the right of the text field, or enter "%" in the text field.

### Server Build Properties

System properties can be referenced using `%system.propertyName%`.

System Property Name	Environment Variable Name	Description
teamcity.version	TEAMCITY_VERSION	The version of TeamCity server. This property can be used to determine the build is run within TeamCity.
teamcity.projectName	TEAMCITY_PROJECT_NAME	The name of the project the current build belongs to.
teamcity.buildConfName	TEAMCITY_BUILDCONF_NAME	The name of the Build Configuration the current build belongs to.
teamcity.buildType.id	none	The <a href="#">unique id</a> used by TeamCity to reference the Build Configuration the current build belongs to
teamcity.configuration.properties.file	none	Full name (including path) of the file containing all the build properties in alphabetical order.
build.is.personal	BUILD_IS_PERSONAL	Is set to <code>true</code> if the build is a <a href="#">personal one</a> . Is not defined otherwise.

build.number	BUILD_NUMBER	The build number assigned to the build by TeamCity using the build number format. The property is assigned based on the <a href="#">build number format</a> .
teamcity.build.id	none	The internal unique id used by TeamCity to reference builds.
teamcity.auth.userId	none	A generated username that can be used to <a href="#">download artifacts</a> of other build configurations. Valid only during the build. <a href="#">Details</a>
teamcity.auth.password	none	A generated password that can be used to download artifacts of other build configurations. Valid only during the build. <a href="#">Details</a>
build.vcs.number.<VCS root ID>	BUILD_VCS_NUMBER_<VCS root ID>	The latest VCS revision included in the build for the root identified. See <a href="#">Configuring VCS Roots</a> for the <VCS root ID> description. If there is only a single root in the configuration, the <code>build.vcs.number</code> property (without the VCS root ID) is also provided.



Please note that this value is a VCS-specific (for example, for SVN the value is a revision number while for CVS it is a timestamp)

## Configuration Parameters

These are the parameters that other properties can reference (only if defined on the Parameters page), but that are not passed to the build themselves.

You can get the full set of such server properties by adding the `system.teamcity.debug.dump.parameters` property to a build configuration and examining the "Available server properties" section in the build log.

Among these properties are the following:

- Dependencies Properties
  - Overriding Dependencies Properties
- VCS Properties
- Branch-Related Parameters
- Other Parameters
  - Detecting Java on Agent
    - Defining Custom directory to Search for Java
    - Defining Java-related Environment Variables

## Dependencies Properties

These are properties provided by the builds the current build depends on (via a snapshot or an artifact dependency).

In the [dependent build](#), dependencies properties have the following format:

```
dep.<btID>.<property name>
```

- <btID> — is the **ID** of the build configuration to get the property from. Only the configurations the current one has snapshot or artifact dependencies on are supported. Indirect dependencies configurations are also available (e.g. A depends on B and B depends on C - A will have C's properties available).
- <property name> — the name of the [build parameter](#) of the build configuration with the given ID.

When using build parameters of type "Password", referencing them from a dependency such as `%dep.<btID>.password%` will not retrieve the actual value. This is done for security reasons to prevent dependencies from accessing the value, thus restricting the possibility of unauthorized access to it.

## Overriding Dependencies Properties

It is possible to redefine the [build parameters](#) in the snapshot-dependency builds when the current build starts. For example, build configuration A depends on B and B depends on C; when triggering A, there is the ability to change parameters in any of its dependencies using the following format:

```
reverse.dep.<btID>.<property name>
```

 Note that if a parameter is redefined in B, but only A is triggered, no parameters change occurs.

It is also possible to change parameter in all dependencies at once using the syntax:

```
reverse.dep.*.<property name>
```

The `reverse.dep.` parameters are processed on queuing of the build where the parameters are defined. As the parameter's values should be known at that stage, they can only be defined either as [build configuration parameters](#) or in the [custom build dialog](#). Setting the parameter during the build has no effect.

Pushing a new parameter into the build will supersede the "[Do not run new build if there is a suitable one](#)" snapshot dependency option and may trigger a new build if the parameter is set to a non-default value.

Note that the values of the `reverse.dep.` parameters are pushed to the dependency builds "as is", without reference resolution. %-references, if any, will be resolved in the context of the build where the parameters are pushed to. `<property name>` is the name of the property to set in the noted build configuration. To set system property, `<property name>` should contain "system." prefix.

#### VCS Properties

These are the settings of VCS roots attached to the build configuration.

VCS properties have the following format:

```
vcsroot.<VCS root ID>.<VCS root property name>
```

- `<VCS root ID>` — is the VCS root ID as described on the [Configuring VCS Roots page](#).
- `<VCS root property name>` — the name of the VCS root property. This is VCS-specific and depends on the VCS support. You can get the available list of properties as described [above](#).

If there is only one VCS root in a build configuration, the `<VCS root ID>.` part can be omitted.

Properties marked by the VCS support as `secure` (for example, passwords) are not available as reference properties.

#### Branch-Related Parameters

When TeamCity starts a build in a build configuration where [Branch specification](#) is configured, it adds a branch label to each build. This logical branch name is also available as a configuration parameter:

```
teamcity.build.branch
```

To distinguish builds started on a default and a non-default branch, there is an additional boolean configuration parameter available since 7.1.5 which allows differentiating these cases:

```
teamcity.build.branch.is_default=true|false
```

For Git & Mercurial, TeamCity provides additional parameters with the names of VCS branches known at the moment of the build start. Note that these may differ from the logical branch name as per branch specification configured. This VCS branch is available form a configuration parameter with the following name:

```
teamcity.build.vcs.branch.<VCS root ID>
```

Where `<VCS root ID>` is the VCS root ID as described on the [Configuring VCS Roots page](#).

#### Other Parameters

Parameter Name	Description
teamcity.build.triggeredBy	a human-friendly description of how the build was triggered
teamcity.build.triggeredBy.username	if the build was triggered by a user, the username of this user is reported. When a build is triggered not by a user, this property is not reported.

## Agent Properties

Agent-specific properties are defined on each build agent and vary depending on its environment. Aside from standard properties (for example, `teamcity.agent.jvm.os.name` or `teamcity.agent.jvm.os.arch`, etc. — these are provided by the JVM running on agent) agents also have properties based on installed applications. TeamCity automatically detects a number of applications including the presence of .NET Framework, Visual Studio and adds the corresponding system properties and environment variables. A complete list of predefined agent-specific properties is provided in the [table](#) below.

If additional applications/libraries are available in the environment, the administrator can manually define the property in the `<agent home>/conf/buildAgent.properties` file. These properties can be used for setting various build configuration options, for defining build configuration requirements (for example, existence or absence of some property) and inside build scripts. For more information on how to reference these properties, see the [Defining and Using Build Parameters in Build Configuration](#) page.

In the TeamCity Web UI, the actual properties defined on the agent can be reviewed by going to the Agents tab at the top navigation bar | <Agent> | <Agent> page | the Agent Parameters tab:

Predefined Property	Description
<code>teamcity.agent.name</code>	The name of the agent as specified in the <code>buildAgent.properties</code> agent configuration file. Can be used to set a requirement of build configuration to run (or not run) on particular build agent.
<code>teamcity.agent.work.dir</code>	The path of <a href="#">Agent Work Directory</a> .
<code>teamcity.agent.work.dir.freeSpaceMb</code>	Free space available in the <a href="#">Agent Work Directory</a> .
<code>teamcity.agent.home.dir</code>	The path of <a href="#">Agent Home Directory</a> .
<code>teamcity.agent.tools.dir</code>	The path to the <a href="#">Tools</a> directory on the Agent
<code>teamcity.agent.jvm.os.version</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.user.country</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.user.home</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.user.timezone</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.user.name</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.user.language</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.user.variant</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.file.encoding</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.file.separator</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.path.separator</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
<code>teamcity.agent.jvm.specification</code>	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)

teamcity.agent.jvm.version	The corresponding JVM property (see <a href="#">JDK help</a> for properties description)
teamcity.agent.jvm.java.home	See the section <a href="#">below</a> for details
teamcity.agent.os.arch.bits	Since TeamCity 10.0 The agent's OS bitness
DotNetFramework<version>[_x86 x64]	This property is defined if the corresponding version(s) of .NET Framework runtime is installed. (Supported versions are 1.1, 2.0, 3.5, <a href="#">4.0</a> - 4.6.1)
DotNetFramework<version>[_x86 x64]_Path	This property value is set to the corresponding framework runtime version(s) path(s)
DotNetFrameworkSDK<version>[_x86 x64]	This property is defined if the corresponding version(s) of .NET Framework SDK is installed. (Supported versions are 1.1, <a href="#">2.0</a> )
DotNetFrameworkSDK<version>[_x86 x64]_Path	This property value is the path of the corresponding framework SDK version.
DotNetFrameworkTargetingPack<version>_Path	This property value is the path to the corresponding Reference assemblies (AKA Targeting Pack) location. (Supported versions are 2.0 - 4.6.1)
WindowsSDK<version>	This property is defined if the corresponding version of Windows SDK is installed. (Supported versions are 6.0, 6.0A, <a href="#">7.0</a> , 7.0A, <a href="#">7.1</a> , 8.0, 8.0A, 8.1, 8.1A, 10)
WindowsSDK<version>_Path	This property value is the path of the corresponding version of Windows SDK.
VS[2003 2005 2008 2010 2012 2013 2015 2017]	This property is defined if the corresponding version(s) of Visual Studio is installed
VS[2003 2005 2008 2010 2012 2013 2015 2017]_Path	This property value is the path to the directory that contains <code>devenv.exe</code>
teamcity.dotnet.nunitlauncher<version>	This property value is the path to the directory that contains the standalone NUnit test launcher, <code>NUnitLauncher.exe</code> . The version number refers to the version of .NET Framework under which the test will run. The version equals the version of .NET Framework and can have a value of 1.1, 2.0, or 2.0vsts.
teamcity.dotnet.nunitlauncher.msbuild.task	The property value is the path to the directory that contains the MSBuild task dll providing the NUnit task for MSBuild, Visual Studio ( <code>sln</code> ).
teamcity.dotnet.msbuild.extensions2.0	The property value is the path to the directory that contains MSBuild 2.0 listener and tasks assemblies.
teamcity.dotnet.msbuild.extensions4.0	The property value is the path to the directory that contains MSBuild 4.0 listener and tasks assemblies.
teamcity.agent.ownPort	The <a href="#">agent port</a> used by the TeamCity server to connect to the agent
teamcity.agent.protocol	The <a href="#">protocol</a> used for data transfers between the agent and the server
teamcity.agent.cpuBenchmark	<a href="#">CPU benchmarking</a> result for the agent
teamcity.agent.hardware.cpuCount	The number of processors in the build agent system
teamcity.agent.hostname	The name of the build agent host



- Make sure to replace `"."` with `"_"` when using properties in MSBuild scripts; e.g. use `teamcity_dotnet_nunitlauncher_msbuild_task` instead of `teamcity.dotnet.nunitlauncher.msbuild.task`
- `_x86` and `_x64` property suffixes are used to designate the specific version of the framework.
- `teamcity.dotnet.nunitlauncher` properties cannot be hidden or disabled.

## Agent Environment Variables

An agent can define some environment variables. These variables can be used in build scripts as usual environment variables.

## Java Home Directories

When a build agent starts, first the installed JDK and JRE are detected; when they are found, the Java-related environment variables are defined as described in [the section below](#).

 The environment variables are defined only if they are not already present in the environment: if a started agent already has the Java-related environment variables set, they are not redefined.

### Detecting Java on Agent

The installed Java is searched for in the ALL locations listed below. Then, every discovered Java is launched to verify that it is a valid Java installation, and the Java version and bitness are determined based on the output.

The following locations are searched (a number of locations is common for all operating systems; some of them are OS-specific):

#### For All OS

- If defined, a custom directory on the agent is searched for Java installations. Defining a custom directory to search for Java is described [below](#).
- The `agent.tools` directory, `<Agent Home Directory>/tools`, is checked for containing a jre or jdk. By default, the subdirectories of `/tools` are not scanned. To search the subdirectories, define `teamcity.agent.java.search.path =%agent.tools.NAME%/INNER_PATH` in the `buildAgent.properties` file.  
 For Unix and Mac OS, remember to [set the executable bit](#) on the files for TeamCity to be able to launch the discovered Java.
- It is checked whether the `JAVA_HOME`, `JDK_HOME`, `JRE_HOME` variables are defined
- The OS-specific locations, listed in the next section, are checked
- The `PATH` environment variables are searched and the discovered directories are checked for containing Java

#### OS-specific locations

##### Windows

- The Windows Registry is searched for the Java installed with the Java installer
- `C:\Program Files` and `C:\Program Files (x86)` directories are searched for `Java` and `JavaSoft` subdirectories
- the `C:\Java` directory is searched

##### Unix

The following directories are searched for Java subdirectories:

- `/usr/local/java`
- `/usr/local`
- `/usr/java`
- `/usr/lib/jvm`
- `/usr`

##### Mac OS

The following directories are searched:

- `/System/Library/Frameworks/JavaVM.framework/Versions/<Java Version>/Home`
- `/Library/Java/JavaVirtualMachines/Versions/<Java Version>/Home`
- `/Library/Java/JavaVirtualMachines/<Java Version>/Contents/Home`

#### Defining Custom directory to Search for Java

You can define a custom directory on an agent to search for Java installations in by adding the `teamcity.agent.java.search.path` property to the `buildAgent.properties` file.

You can define a list of directories separated by an OS-dependent character.

#### Defining Java-related Environment Variables

For each major version `V` of java, the following variables can be defined:

- `JDK_1V`
- `JDK_1V_x64`
- `JRE_1V`
- `JRE_1V_x64`

The JDK variables are defined when the JDK is found, the JRE variables are defined when the JRE is found but the JDK is not. The `_x64` variables point to 64-bit java only; the variables without the `_x64` suffix may point to both 32-bit or 64-bit

installations but 32-bit ones are preferred.

If several installations with the same major version and the same bitness but different minor version/update are found, the latest one is selected.

In addition, the following variables are defined:

- `JAVA_HOME` - for the latest JDK installation (but 32-bit one is preferred)
- `JDK_HOME` - the same as `JAVA_HOME`
- `JRE_HOME` - for the latest JRE or JDK installation (but 32-bit one is preferred), defined even if JDK is found.

The `JRE_HOME` and `JDK_HOME` variables may point to different installations; for example, if JRE 1.7 and JDK 1.6 but no JDK 1.7 installed - `JRE_HOME` will point to JRE 1.7 and `JDK_HOME` will point to JDK 1.6.

All variables point to the java home directories, not to binary files. For example, if you want to execute `javac` version 1.6, you can use the following path:

In a TeamCity build configuration:

```
%env.JDK_16%/bin/javac
```

In a Windows bat/cmd file:

```
%JDK_16%\bin\javac
```

In a unix shell script:

```
$JDK_16/bin/javac
```

#### Agent Build Properties

These properties are unique for each build: they are calculated on the agent right before build start and are then passed to the build.

System Property Name	Environment Variable Name	Description
<code>teamcity.build.checkoutDir</code>	<code>none</code>	<b>Checkout directory</b> used for the build.
<code>teamcity.build.workingDir</code>	<code>none</code>	<b>Working directory</b> where the build is started. This is a path where TeamCity build runner is supposed to start a process. This is a runner-specific property, thus it has different value for each new step.
<code>teamcity.build.tempDir</code>	<code>none</code>	Full path of the build temp directory automatically generated by TeamCity. The directory will be cleaned after the build.
<code>teamcity.build.properties.file</code>	<code>TEAMCITY_BUILD_PROPERTIES_FILE</code>	Full name (including path) of the file containing all the <code>system.*</code> properties passed to the build. "system." prefix stripped off. The file uses <b>Java properties</b> file format (for example, special symbols are backslash-escaped).
<code>teamcity.build.changedFiles.file</code>	<code>none</code>	Full path to a file with information about changed files included in the build. This property is useful if you want to support running of new and modified tests in your tests runner. This file is only available if there were changes in the build; it is not available for history builds.

See also system properties related to [Risk Tests Reordering in Custom Test Runner](#)

[Project and Agent Level Build Parameters](#)

In addition to defining build parameters in Build Configuration settings, you can define them on the project or build agent level.

- Project Level Build Parameters
- Agent Level Build Parameters

## Project Level Build Parameters

TeamCity allows you to define build parameters for a project, all its subprojects and build configurations in one place: Project Settings -> Parameters tab.

Note that if a build parameter P is defined in a build configuration and a build parameter with the same name exists on the project level, the following heuristics applies:

Case 1: Project A, Build Configuration from project A.

Parameters defined in the build configuration have priority over the parameters with the same names defined on project level.

Case 2: Project A, Template T from project A, build configuration from project A inherited from template T.

Parameters of the build configuration have priority over the parameters with the same name defined in project A, and project-level parameters have priority over parameters with the same name defined in the template.

Case 3: Project A1, Project A2, Template T from project A1, build configuration from project A2 inherited from template T.

Parameters of project A2 (the one build configuration belongs to) have priority over the parameters with the same names defined in the template.

You can also define parameters for only those build configurations of the project that use the same VCS root. To do that, create a text file named `teamcity.default.properties`, and check it into the VCS root. Ensure that the file appears directly in the **build working directory** by specifying the appropriate **checkout rules**. The name and path to the file can be customized via the `teamcity.default.properties` property of a build configuration.

The properties defined this way are not visible in the TeamCity web UI, but are passed directly to the build process.

## Agent Level Build Parameters

To define agent-specific properties, edit the Build Agent's `buildAgent.properties` file (<agent home>/conf/buildAgent.properties). Refer to the [Agent-Specific Properties](#) page for more information.

When defining system properties and environment variables in the `teamcity.default.properties` or `buildAgent.properties` file, use the following format:

```
[env|system].<property_name>=<property_value>
```

For example: `env.CATALINA_HOME=C:\tomcat_6.0.13`

See also:

[Administrator's Guide: Configuring Build Parameters | Defining and Using Build Parameters in Build Configuration | Predefined Build Parameters](#)

[Typed Parameters](#)

When adding a [build parameter](#) (system property, environment variable or configuration parameter), you can extend its definition with a specification that will regulate parameter's control presentation and validation.

This specification is the parameter's "meta" information that is used to display the parameter in the [Run Custom Build](#) dialog. It allows making a custom build run more user-friendly and usable by non-developers.

Consider a simple example. You have a build configuration in which you have a monstrous-looking build parameter that regulates if a build has to include a license or not; can be either true or false; and by default is false. It may be clear for a build engineer, which build parameter regulates license generation and which value it is to have, but it may not be obvious to a regular user.

Using the build parameter's specification you can make your parameters more readable in the Run Custom Build dialog.

On this page:

- [Adding Parameter Specification](#)
- [Manually Configuring Parameter Specification](#)
- [Copying Parameter Specification](#)
- [Modifying Parameter Specification via REST API](#)

#### Adding Parameter Specification

To add specification to a build parameter, click the Edit button in the Spec area when editing/adding a build parameter.

All parameters specifications support a number of common properties, such as:

- Label: some text that is shown near the control in the Run Custom Build dialog.
- Description: some text that is shown below the control containing an explanatory note of the control use.
- Display: If hidden is specified, the parameter will not be shown in the Run Custom Build dialog, but will be sent to a build; if prompt is specified, TeamCity will always require a review of the parameter value when clicking the Run button (won't require the parameter if build is triggered automatically); if normal is selected, the parameter will be shown as usual.
- Read-only: if the box is checked, it will be impossible to override the parameter with a different value
- Type : Currently you can present parameters in following forms:
  - a simple text field with the ability to validate its value using regular expression;
  - a checkbox;
  - a select control;
  - a password field.

The table below provides more details on each control type.

Type	Description
Text	The default. Represents a usual text string without any extra handling
Checkbox	True/false option represented by a check box
Select	"Select one" or "select many" control to set the value to one of predefined settings
Password	This is designed to store passwords or other secure data in TeamCity settings. TeamCity makes the value of the password parameter never appear in the TeamCity Web UI: it affects the settings screens and the Run Custom Build dialog where password fields appear. Also, the value is replaced in the build's Parameters tab and build log. The value is stored scrambled in the configuration files under TeamCity Data Directory. Please note that build log value hiding is implemented with simple search-and-replace, so if you have a trivial password of "123", all occurrences of "123" will be replaced, potentially exposing the password. Setting the parameter to type password does not guarantee that the raw value cannot be retrieved. Any project administrator can retrieve it and also any developer who can change the build script can in theory write malicious code to get the password.

Depending on the specification's type, there are additional settings.

Text	Allowed value - choose the allowed value. For the Regex option, specify Pattern, a Java-style regular expression to validate the field value, as well as a validation message.
------	--

Checkbox	<p>Checked value/Unchecked value: Specify values for the parameter to have depending on the checkbox's state.</p> <p>It is recommended to specify the checked value and keep the default value and the unchecked value of the parameter the same.</p> <p>Depending on the way the build is triggered, the checkbox behavior will be as follows:</p> <ul style="list-style-type: none"> <li>- if the build is triggered by an automatic trigger or by clicking the Run button (without the <a href="#">run custom build dialog</a>), the default value is used</li> <li>- if the build is triggered via the <a href="#">run custom build</a> dialog without changing anything, the 'unchecked value' is used (if not empty); and if it is empty, the default value is used</li> <li>- if the build is triggered via the <a href="#">run custom build</a> dialog with the box checked, the 'checked value' is used (if not empty); and if it is empty, the 'true' value is used</li> </ul>
Select	<p>Check the Allow multiple box to enable multiple selection. In the Items field specify a newline-separated list of items. Use the following syntax <code>label =&gt; value or value</code>.</p>

#### Manually Configuring Parameter Specification

Alternatively, you can manually configure a specification using a specially formatted string with the syntax similar to the one used in service messages (`typeName key='value'`).

For example, for text: `text label='some label' regex='some pattern'`.

#### Copying Parameter Specification

If you start editing a parameter that has a specification, you can see a link to its raw value in the "Edit parameter" dialog. Click it to view the specification in its raw form (in the service message format). To use this specification in another build configuration, just copy it from here, and paste in another configuration.

#### Modifying Parameter Specification via REST API

You can also view/edit typed parameters specification via [REST API](#).

See also:

[Administrator's Guide: Configuring Build Parameters | Defining and Using Build Parameters in Build Configuration | Predefined Build Parameters](#)

## Configuring Agent Requirements

By specifying [Agent Requirements](#) for build configuration you can control on which agents the configuration will be run.

To add a requirement, click corresponding link and specify the following options:

Parameter Type	Specify the type of the parameter: system property, environment variable, or configuration parameter. For details on the types of parameters available in TeamCity, please refer to <a href="#">Configuring Build Parameters</a> section.
Parameter Name	Specify the mandatory property or environment variable name.

Condition	Select condition from the drop-down list.
	<p><b>i</b> Some notes on how conditions work:</p> <ul style="list-style-type: none"> <li>• equals: This condition will be true if an empty value is specified and the specified property exists and its value is an empty string; or if a value is specified and the property exists with the specified value.</li> <li>• does not equal: This condition is true if an empty value is specified and the property exists and its value is NOT empty; or if a specific value is specified and either the property doesn't exist, or the property exists and its value does not equal the specified value.</li> <li>• does not contain: This condition will be true if the specified property either does not exist or exists and does not contain the specified string.</li> <li>• is more than, is not more than, is less than, is not less than: These conditions only work with numbers.</li> <li>• matches, does not match: This condition will be true if the specified property matches/does not match the specified <a href="#">Regular Expression</a> pattern.</li> <li>• version is more than, version is not more than, version is less than, version is not less than: compares versions of a software. Multiple formats are supported including ". "-delimited, leading zeroes, common suffixes like "beta", "EAP". If the version number contains alphabetic characters, they are compared as well, for instance, 1.1e &lt; 1.1g.</li> </ul>
Value	Is shown for the conditions which require the value to match against (like "equals"). The value supports parameters resolution, but only the parameters which do not originate from the agents are supported in the field.

Note that the [Agent Requirements](#) page also displays the list of compatible and incompatible build agents for this build configuration, which is updated each time you modify the list of requirements. Possible reasons why build agent may be incompatible with this build configuration are [described separately](#).

See also:

[Concepts: Agent Requirements](#) | [Build Agent](#) | [Build Configuration](#)

## Triggering a Custom Build

A build configuration usually has [build triggers](#) configured which automatically start a new build each time the conditions are met, like scheduled time, or detection of VCS changes, etc.

Besides triggering a build automatically, TeamCity allows you to run a build manually whenever you need, and customize this build by adding properties, using specific changes, running the build on a specific agent, etc.

On this Page:

- Run Custom Build dialog
  - General Options
  - Dependencies
  - Changes
    - Include changes
    - Build branch
    - Use settings
  - Parameters
  - Comment and Tags
- Promoting Build

There are several ways of launching a custom build in TeamCity:

- Click the ellipsis on the Run button, and specify the options in the Run Custom Build dialog described [below](#).
- To run a custom build with specific changes, open the build results page, go to the [Changes tab](#), expand the required change, click the Run build with this change and proceed with the [options](#) in the Run Custom Build dialog.
- Use [HTTP request](#) or [REST API request](#) to TeamCity to trigger a build.
- Promote a build - see the section [below](#).

### Run Custom Build dialog

#### General Options

Select an agent you want to run the build on from the drop-down list. Note that for each agent in the list, TeamCity displays its

current state and estimates when the agent will become idle if it is running a build at the moment. Besides the possibility to run a build on a particular agent from the list, you can also use one of the following options:

- fastest idle agent — default option; if selected, TeamCity will automatically choose an agent to run a build on based on calculated estimates.
- the fastest agent in <a certain> pool - if selected, TeamCity will run a build on an agent from a specified pool
- if a [cloud integration](#) is configured, you can select to run a build on an agent from a certain cloud image. If no available cloud agents of this type exist, TeamCity will also attempt to start a new one.
- run a build on <a specified> agent
- All enabled compatible agents :

Use this option to run a build simultaneously on all agents that are enabled and compatible with the build configuration. This option may be useful in the following cases:

- run a build for agent maintenance purposes (e.g. you can create a configuration to check whether agents function properly after an environment upgrade/update).
- run a build on different platforms (for example, you can set up a configuration, and specify for it a number of compatible build agents with different environments installed).

On the General options you can also specify whether

- this particular build will be run as a [personal](#) one
- this particular build will be put at the top of the [build queue](#)
- all files in the [build checkout directory](#) will be cleaned before this build.
  - Since TeamCity 2017.1, if snapshot dependencies are configured, this option can be applied to snapshot dependencies. In this case, all the builds of the build chain will be forced to use clean checkout. The option also enables rebuilding all dependencies (unless custom dependencies are provided via this dialog or the schedule trigger promotes a build).

#### Dependencies

This tab is available only for builds that have dependencies on other builds.

You can enforce rebuilding of all dependencies and select a particular build whose artifacts will be taken. By default, the last 20 builds are displayed. To increase the number of builds displayed in the drop-down to 50, use the `teamcity.runCustomBuild.buildsLimit=50` [internal property](#).

#### Changes

This tab is available only if you have permissions to access VCS roots for the build configuration.

The tab allows you to specify a particular change to be included to the build.

The build will use the change's revision to checkout the sources. That is, all the changes up to the selected one will be included into the build.

Note that TeamCity displays only the changes detected earlier for the current build configuration VCS roots. If the VCS root was detached from the build configuration after the change occurred, there is no ability to run the build on such a change. A limited number of changes is displayed. If there is an earlier change in TeamCity that you need to run a build on, you can locate the change in the Change Log and use the Run build with this change action.

#### Include changes

The Include changes drop-down allows selecting the changes in the VCS roots attached to the configuration to run the build on.

- Latest changes at the moment the build is started: TeamCity will automatically include all changes available at the moment.
- <Last change to include>: When you select a change in the drop-down list, TeamCity runs the build with the selected change and all changes that were made before it. The build run with the changes earlier than the latest available is marked as a [History Build](#).

#### Build branch

The Build branch drop-down, available if you have branches in your build configuration (or in snapshot dependencies of this build configuration), allows choosing a branch to be used for the custom build.

#### Use settings

If your project has [versioned settings](#) enabled, you can tell TeamCity to run a build:

- with the settings defined for the project, either the current settings on the server or the settings from VCS
- with the project settings currently defined on the server
- with the settings loaded from the VCS revision calculated for the build.

If changes are selected in the step above, the revision of the project settings corresponding to the selected changes will be loaded.

To define which settings to take, use one of the corresponding options from the Use settings drop-down (the option here will override the [project-level setting](#)).

#### Parameters

These settings are available only if you have permissions to change system properties and environment variables for the build configuration.

This tab allows adding new parameters/properties/variables, and editing or deleting them, as well as redefining values of the [predefined ones](#).

When adding/editing/deleting properties and variables, note the following:

- For a predefined property/variable, only the value is editable.
- Only newly added properties/variables can be deleted. You cannot delete predefined properties.

#### Comment and Tags

Add an optional comment as well as one or more [tags](#) to the build. You can also add a custom build to [favorites](#) by checking the corresponding box in this section.

 A greater build number does not mean more recent changes and the last build in the builds history does not reflect the state of the latest project sources: builds in the builds history are sorted by their start time, not by changes they include.

### Promoting Build

Build promotion refers to triggering a custom build with an overridden [artifact or snapshot dependency](#), i.e. manual launching of a build with dependencies configured, but using a build different from the build specified in the dependency.

To promote a build, open the build results page of the dependency build and click Actions | Promote.

For example, your build configuration A is configured to take artifacts from the last successful build of configuration B, but you want to run a build of configuration A using artifacts of a different build of configuration B (not the last successful build), so you promote an earlier build of B.

Build promotion affects only a single run of the dependent build. Once you click Promote, a build of the dependent build configuration which uses the artifacts of the specified build is queued. Any further runs of the dependent build configuration will use artifacts as configured (last successful, last pinned etc.), unless you use another promotion.

More details are available in the [related blog-post](#).

See also:

Concepts: [Build Queue](#) | [Dependent Build](#) | [Personal Build](#)  
Administrator's Guide: [Configuring Build Triggers](#)

## Copy, Move, Delete Build Configuration

To copy, move or delete a build configuration, use the Actions menu on the right of the build configuration settings pages.

- [Copy and Move Build Configuration](#)
- [Delete Build Configuration](#)

### Copy and Move Build Configuration

Build configurations can be copied and moved to another project by project administrators:

- A copy duplicates all the settings of the original build configuration, but no data related to builds is preserved. The copy is created with empty build history and no statistics. You can copy a build configuration into the same or another project.
- When moving a build configuration between projects, TeamCity preserves its settings and associated data, as well as its build history and dependencies.

On copying, TeamCity automatically assigns a new [ID](#) to the copy. It is also possible to change the ID manually.

Selecting the Copy associated user, agent and other settings option makes sure that all the settings like notification rules or agent's compatibility are exactly the same for the copied and original build configurations for all the users and agents affected.

If the build configuration uses VCS Roots or is associated with a template which is not accessible in the target project (does not belong to the target project or one of its parent projects), the copies of these VCS roots and the template will be created in the target project. (see also related issue [TW-28550](#))

 When running TeamCity in the [Professional mode](#) with the maximum allowed number of build configurations (100 build configurations and prior to TeamCity 2017.2 - 20) unless you purchased additional Build Agent licenses), the Copy option will not be displayed for build configurations.

## Delete Build Configuration

When you delete a build configuration, TeamCity will remove its .xml configuration file. After the deletion, there is a [configurable](#) timeout - since TeamCity 2018.1, it is 5 days by default - before the builds of the deleted configuration are removed during the build history clean-up.

 If you attempt to delete a build configuration which has [dependent build configurations](#), TeamCity will warn you about it. If you proceed with deletion, the dependencies will no longer function.

## Ordering Projects and Build Configurations

By default, TeamCity displays projects, their subprojects, build configurations, and templates in the alphabetical order. Project administrators can apply custom ordering to subprojects and build configurations on the Project Settings page for the parent project and use it as the default order.

To enable reordering, use the corresponding button above the list.

Individual users can still manually tweak the display using the up-down button in the [Configure Visible Projects](#) pop-up on the Projects Overview page.

## Archiving Projects

If a project is not in an active development state, you can archive it using the Actions menu in the top right of the project settings page. On the Archive project screen, you can choose whether to cancel the builds which are already in the queue using the corresponding option (enabled by default).

When archived:

- All the subprojects of the current project are archived as well.
- All project's build configurations are automatically [paused](#).
- Automatic checking for changes in the project's [VCS roots](#) is not performed if the VCS roots are not used in other non-archived projects.
- As part of pausing, automatic build triggering is disabled. However, builds of the project can be triggered manually or automatically as a part of a build chain.
- All data (settings, build results, artifacts, build logs, etc.) of the project's build configurations are preserved - you can still edit settings of the archived project or its build configurations.
- Archived projects do not appear in most user-facing projects lists and in IDEs including the list of build configurations for remote run.

By default, permissions to archive projects are given to project and system administrators.

If the parent project of an archived subproject is displayed in the project list (e.g. on the [Projects Overview](#) page), the archived subprojects are displayed with the corresponding note.

## Dearchiving Projects

If you need to dearchive a project, you can do it using the Actions menu in the top right of the project settings page or the More menu next to the project on the Root project settings page.

See also:

[Concepts: Project | Build Configuration](#)

## Ordering Build Queue

This page lists the options to manage the queue builds manually. Automatic build queue optimization is detailed in the [separate section](#).

On this page:

- Manually Reordering Builds in Queue
  - Moving Builds to Top
- Managing Build Priorities
- Removing Builds From Build Queue
- Limiting Maximum Size of Build Queue

## Manually Reordering Builds in Queue

To reorder builds in the [Build Queue](#), you can simply drag them to the desired position.

## Moving Builds to Top

- For any queued build, do one of the following:
  - on the [Build Queue](#) page, click the arrow button next to the build sequence number  to move the build to the top of the queue.
  - click the build number or build status link anywhere in the UI, and, on the [queued build](#) page, click the Actions menu in the top right. Select the Move to top action. The queue position will change. For a composite build, the whole build chain will be moved to the top of the queue.
- For a running composite build which has dependencies that have not yet started, click the build number or build status link anywhere in the UI, and, on the [running build](#) page, click the Actions menu in the top right. Select the Move queued dependencies to top action. All queued dependencies of this build will be moved to the top of the queue.

## Managing Build Priorities

By default, builds are placed in the build queue in the order they are triggered: most recently triggered build is added to the bottom of the queue. It is possible to change build's priorities so that builds are inserted into the build queue at a position depending on their defined priority and the wait time of the currently queued builds.

In TeamCity you can control build priorities by creating Priority Classes. A priority class is a set of build configurations with a specified priority (the higher the number, the higher the priority. For example, priority=2 is higher than priority=1). The higher priority a configuration has, the higher place it gets when added to the Build Queue.

To access these settings, on the Build Queue tab, click the Configure Build Priorities link in the upper right corner of the page.



Note that only users with the System Administrator role can manage build priority classes.

By default, there are two predefined priority classes: Personal and Default, both with priority=0.

- All personal builds ([Remote Run](#) or [Pre-tested Commit](#)) are assigned to the Personal priority class once they are added to the build queue. Note that you can change the priority for personal builds here.
- The Default class includes all the builds not associated with any other class. This allows to create a class with priority lower than default and place some builds to the bottom of the queue.

To create a new priority class:

1. Click Create new priority class.
2. Specify its name, priority (in the range -100..100) and additional description. Click Create.
3. Click the Add configurations link to specify which build configurations should have priority defined in this class.



This setting is taken into account only when a build is added to the queue. The builds with higher priority will have more chances to appear at the top of the queue; however, you shouldn't worry that the build with lower priority won't ever run. If a build spent long enough in the queue, it won't be outrun even by builds with higher priority.

## Removing Builds From Build Queue

To remove build(s) from the Queue, check the Remove box next to the selected build and confirm the deletion. If a build to be removed from the queue is a part of a build chain, TeamCity shows the following message below comment field: "This build is a part of a build chain". Refer to the [Build Chain](#) description for details.

Also you can remove all your personal builds from the queue at once from the Actions menu.

Since TeamCity 10.0, it is possible to remove several builds of [paused build configurations](#) from the queue.

## Limiting Maximum Size of Build Queue

It is possible to limit the maximum number of builds in the queue. By default, the limit is set to 3000 builds. The default value

can be changed using the `teamcity.buildTriggersChecker.queueSizeLimit` internal property.

When the queue size reaches the limit, TeamCity will pause **automatic build triggering**. Automatic build triggering will be re-enabled once the queue size gets below limit. While triggering is paused, a warning message is shown to all of the users.

- While automatic triggering is paused, it is still possible to add builds to the queue **manually**.

See also:

Concepts: Build Queue

## Muting Test Failures

TeamCity provides a way to "mute" any of the currently failing tests so they will not affect build status for future builds. Muting will only affect builds which fail on "at least one test failed" **build failure condition** but not others: if any of the other failure conditions apply to your build (non-zero exit code, etc.), it will still fail even if failing tests are muted.

This feature is useful when some tests fail for some known reason, but it is currently not possible to fix them. For example, the responsible developer is on vacation, or you are waiting for the system administrators to fix the environment, or the test is failing intentionally, for example, if the required functionality is not yet written (TDD). In these cases you can mute such failures and avoid unnecessary disturbance of other developers.

When a test is muted, it is still run in the future builds, but its failure does not affect the build status. The test can be unmuted manually on a specific date or after a successful run. Also, tests can be muted only in a single build configuration or in all the build configurations of a specific TeamCity project.

Your build script might need adjustment to make the build green when there are failing but muted tests. Make sure that the build does not fail because of other build failure conditions (e.g. "Fail if build process exit code is not zero") in case the only errors encountered were tests failures. See also the related issue [TW-16784](#).

### How to mute tests

The Mute/unmute problems in project permission required to mute tests is granted to Project administrator and System administrator by default.

You can mute a test failure from:

- The Projects page
- The Project overview page
- The Build Configuration home page
- The Current Problems tab

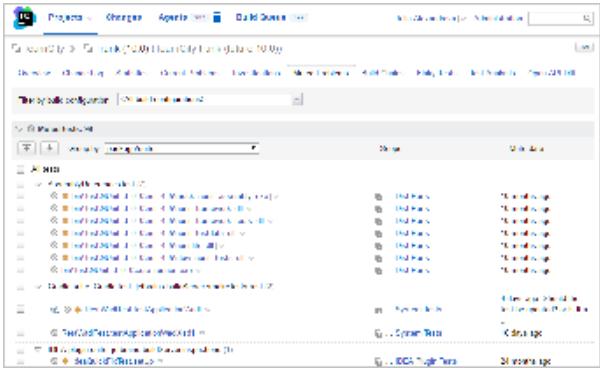
On the build results page you can select several test failures (or all) to be muted or unmuted: select the tests, click the Investigate.. button and specify the mute options.

Note that you can start investigation of the problem simultaneously with muting the failure. When muting a test failure, you can specify conditions when it should be unmuted: on a specified date or when it is fixed. Alternatively, you can unmute it manually.

On the build results page you can view the list of muted test failures, their stacktraces and details about the mute status:

The screenshot shows the TeamCity build results page for a build labeled '149 (14072) (24 Mar 10 09:45)'. The 'Changes' tab is selected. A list of muted test failures is displayed under the 'Test Result' section. One entry is highlighted: 'Tests passed: 11 failed: 1' with a note '1 test failure muted'. The 'Details' column shows the failure was reported by 'Oleg Kulikov' on '24 Mar 10 09:45'. Below this, a 'Snapshot dependencies' section shows '1 test failure muted' with a note 'Julia Balashova muted 15 days ago'.

From the Project Home page you can navigate to the Muted Problems tab to view all problems muted in all build configurations within project:



## Changing Build Status Manually

### Overview

A user with appropriate permissions can change the status of a build manually, i.e. make it either failed or successful (issue [TW-2529](#)).

The corresponding action is available in the Actions menu on the [build results](#) page.

### Marking build as successful

You may want to make build successful to:

- Change the last successful build anchor when using [Build failure conditions](#), i.e. if your last build failed because of an incorrect value of a metric, and this new value is valid, you may mark this build with a successful anchor.
- Allow using an incorrectly failed build with good artifacts in the "last successful" dependencies.
- For a running [personal build](#), you can mark the current failures as non-relevant to allow pre-tested commit to pass (if the user has permission to do this).

The "Mark as successful" action is not available for Failed to Start Builds.

### Marking build as failed

You may want to mark a build as failed when:

- The build has some problem which didn't affect the final [build status](#).
- There is a known problem with the build, and it should be ignored by your QA team.
- You've mistakenly marked the build as successful manually.

### Permissions

By default, the permission to change the build status is granted to Project Administrator.

## Customizing Statistics Charts

To help you track the condition of your projects and individual build configurations over time, TeamCity gathers statistical data across all their history and displays it as visual charts. This page describes how to modify the pre-defined project-level charts.

Refer to a separate page to add [custom charts](#) on the project or build configuration level.

- [Modifying Pre-defined Project-level Charts](#)
  - [Disabling Charts of Particular Type on Project Level](#)
  - [Showing Charts Only for Specific Build Configurations on Project Level](#)

### Modifying Pre-defined Project-level Charts

By default, the Statistics tab on the project level shows charts for all build configurations in the current project, which have coverage, duplicates or inspections data. However, you can disable charts of a particular or specify build configurations to be used in the charts.

To modify pre-defined project level charts, you need to configure the `<TeamCity Data Directory>/config/projects/<project_name>/pluginData/plugin-settings.xml` file. In this file a similar format is used for all types of pre-defined graphs:

Chart Type	XML Tag Name

Code Coverage	coverage-graph
Code Duplicates (Java and .NET)	duplicates-graph
Code Inspections	inspections-graph

#### Disabling Charts of Particular Type on Project Level

To disable charts of particular type for a project, use the following syntax:

```
<coverage-graph enabled="false"/>
```

In this example, all code coverage charts will be removed from the Statistics page.

#### Showing Charts Only for Specific Build Configurations on Project Level

To show the code coverage chart related only to a particular build configuration, use the following syntax:

```
<coverage-graph enabled="true">
  <build-type id="myConf1"/>
  <build-type id="myConf2"/>
</coverage-graph>
```

where myConf1 and myConf2 values are [build configuration IDs](#).

However, note that build configurations specified should contain code coverage data for the charts to be shown. If the data is available, two charts will be shown (one for each specified build configuration).

See also:

Concepts: [Code Coverage](#) | [Code Inspection](#) | [Code Duplicates](#)  
User's Guide: [Statistic Charts](#)  
Extending TeamCity: [Build Script Interaction with TeamCity](#)

## Configuring Artifacts Storage

- [Built-in Artifacts Storage](#)
- [External Artifacts Storage](#)
  - [Amazon S3 Support](#)
  - [Permissions](#)
  - [Other External Artifact Storage Plugins](#)
    - [Azure Artifact Storage](#)
    - [Google Cloud Artifact Storage](#)

The [Project Settings | Artifacts Storage](#) tab displays artifact storages configured in this project as well as the storages inherited from parents. By default, the built-in TeamCity artifacts storage is displayed and marked as active. You can activate a different storage using the corresponding link.

### Built-in Artifacts Storage

TeamCity stores [artifacts](#) produced by builds on the file system accessible by the TeamCity server. The default location is `<Team city Data Directory>/system/artifacts` but this location can be [redefined](#).

### External Artifacts Storage

TeamCity provides a pluggable API to enable external storage for TeamCity build artifacts. Support for different storages can be implemented as an external plugin to TeamCity: the details are provided in the [external storage implementation guide](#).



Note that when an external storage for artifacts is enabled, the TeamCity [internal](#) artifacts (including build logs) will still be published to the TeamCity server and stored in the TeamCity Data Directory in the built-in artifacts storage.

The same applies to the metadata about artifacts mappings, which will be published to the [artifacts directory](#) of the TeamCity Data directory. When restoring from a backup, make sure they are restored for the external artifact plugin to work properly.

## Amazon S3 Support

Since TeamCity 2018.1 TeamCity comes bundled with [Amazon S3 Artifact Storage](#) plugin which allows storing build artifacts in an Amazon S3 bucket.

It is possible to replace the TeamCity built-in artifacts storage with [AWS S3](#) at the project level. When S3 artifact storage is configured, it:

- allows uploading to, downloading and removing artifacts from S3
- handles resolution of artifact dependencies as well as clean-up of artifacts
- displays artifacts located externally in the TeamCity web UI.

To enable external artifact storage in an AWS S3 bucket

1. Navigate to the [Project Settings | Artifacts Storage](#) tab. The built-in TeamCity artifacts storage is displayed by default and marked as active.
2. Click [Add new storage](#). S3 Storage is selected as the storage type (provided there are no other external storage plugins installed).
3. Provide an optional name for your storage.
4. Select the AWS environment and provide the required settings (if any).
5. Provide your AWS Security Credentials.
6. Specify an existing S3 bucket to store artifacts.
7. Save your settings.
8. The configured S3 storage will appear on the Artifacts storage page. Make it active using the corresponding link.

Now new artifacts produced by builds of this project with its subprojects and build configurations will be stored in the specified AWS S3 bucket.

## Permissions

The plugin requires the following S3 permissions:

- build agent: `ListBucket`, `PutObject` when pre-signed URLs option is disabled
- server: `DeleteObject`, `ListAllMyBuckets`, `GetBucketLocation`, `GetObject`

## Other External Artifact Storage Plugins

### Azure Artifact Storage

[Azure Artifact Storage](#) is an experimental plugin by JetBrains which allows replacing the TeamCity built-in artifacts storage by Azure Blob storage.

### Google Cloud Artifact Storage

Google Cloud Artifact Storage is implemented as a [plugin](#) by JetBrains.

## Licensing Policy

This page covers:

- Licensing Overview
- Editions
  - Number of Build Configurations
  - Number of Agents
- Managing Licenses
- Valid TeamCity Versions
- License Expiration
- Ways to Obtain a License
- Upgrading From Previous Versions
  - Upgrading from TeamCity 5.x and later
  - Upgrading from TeamCity 4.x to TeamCity 5.0 and later
  - Upgrading from TeamCity 3.x to TeamCity 4.0

- [Upgrading from TeamCity 1.x-2.x to TeamCity 4.0](#)
- [Upgrading with IntelliJ IDEA 6.0 License Key](#)

Pricing and new licenses/upgrades purchase are available via the [official web site](#). If you have any questions on the licensing terms, obtaining or upgrading license key and related, please [contact JetBrains sales department](#). You can review TeamCity license agreement on the [official web site](#) or in the footer of the installed TeamCity server web UI.

## Licensing Overview

JetBrains offers several licensing options that allow you to scale TeamCity to your needs.

This section illustrates the main differences between the TeamCity server [editions](#) and provides general information on the TeamCity [Build Agent](#) license.

For detailed information, refer to the sections below.

Professional Server	Enterprise Server
no license key is required, free	a license key is required, <a href="#">price options</a>
100 build configurations (20 prior to TeamCity 2017.2)	unlimited number of build configurations
full access to all product features	free 1-year subscription to upgrades
support via <a href="#">community forum</a>	priority email <a href="#">support</a>
3 build agents included, buy more as necessary	from 3 to 100 build agents included, buy more as necessary

If you need more build agents that are included with your TeamCity server edition, you can purchase additional build agent licenses.

Build Agent License
connects 1 additional build agent
if using Professional edition, adds 10 additional build configurations
a license key is required, <a href="#">price options</a>

## Editions

There are two editions of TeamCity: Professional and Enterprise.

The editions are equal in all the features except for the maximum number of build configurations allowed.

The same TeamCity distribution and installation are used for both editions. You can switch to the Enterprise edition by entering the appropriate license key. All the data is preserved when the edition is switched.

The current edition in use is noted in the footer of every TeamCity web UI page and on the Administration > Licenses page as well as in teamcity-server.log on the server startup.

The Professional edition does not require any license key and can be used free of charge. The only functional difference from the Enterprise edition is a limitation of the maximum number of [build configurations](#). The limit is 100 build configurations (prior to TeamCity 2017.2 it was 20). It can be extended by 10 with each agent license key added. You can install several servers with the Professional license.

The Enterprise edition requires a license key, has no limit on the number of build configurations and entitles you to TeamCity [support](#) from JetBrains for the maintenance period of the license.



An additional server in a [high availability set-up](#) uses the license from the main server and does not require a separate license.

Each TeamCity edition comes bundled with 3 or more [build agents](#). To use more agents than the bundled number, separate build agent license keys can be entered. Additional agents can be added to both editions.

Besides the Professional and Enterprise licenses, there are two more license types:

- Evaluation — has an expiration date and provides an unlimited number of agents and build configurations. To obtain the evaluation license, use the link on [TeamCity download page](#). The evaluation license can be obtained only once for each

major TeamCity version. A second evaluation license key from the site is not accepted by the same major version of TeamCity server. If you need to extend/repeat the evaluation, please [contact](#) our sales department. Each [EAP](#) (preview, not stable) release of TeamCity comes bundled with a 60-day evaluation license.

- Open Source — this is a special type of license granted for open source projects, it is time-based, and provides an unlimited number of agents. Refer to the details on [the page](#)

The TeamCity Licensing Policy does not impose any limitations on the number of instances for any of the IDE plugins or the Windows Tray Notifiers.

## Number of Build Configurations

The Enterprise edition has no limit on the number of build configurations.

The Professional edition allows 100 [build configurations](#) per server since TeamCity 2017.2 (20 in earlier versions). Since TeamCity 8.0, each build agent license key gives you 10 more build configurations in Professional edition in addition to one more agent. All build configurations are counted (i.e. including those in archived projects).

## Number of Agents

TeamCity Professional edition comes bundled with 3 [build agents](#). More build agents can be added by purchasing additional agent license keys.

A server license key might include more agent licenses than the default 3. The number of agents stated for the server license on [jetbrains.com](#) site notes the total number of agents which will be available. Separate agent license keys can be used with either TeamCity edition (Enterprise and Professional). For more information about purchasing agent licenses, refer to [the product page](#).

The number of agent licenses limits the number of agents which can be [authorized](#) in TeamCity. The license keys are not bound to specific agents, they just limit the maximum number of functional agents. The licensing makes no difference between local (installed on the TeamCity server machine) and remote agents.

When there are more authorized agents than the agent licenses available, the server stops to start any builds and displays a warning message to all users in the web browser.

## Managing Licenses

You can enter new license keys and review the currently used ones (including the license issue date and maintenance period) on the Administration > Licenses page of the TeamCity web UI. By default, only users with the System Administrator role can access the page. Adding or removing licenses on the page is applied immediately.

A single license can only be used on a single running server. If you create a copy of the server and run two servers at the same time, you should ensure each license key is used on a single server only. You can use the Evaluation (limited time) license to run a server for testing/non production purposes. The licenses are not bound to specific server instance, machine, etc. The only limitation is that a license cannot be used on several servers at the same time.

When you already own license(s) and buy more licenses, you can [request](#) JetBrains sales to make the new licenses co-termed with those already purchased, so that all the licenses have equal maintenance expiration date. The cost of the licenses is then lowered proportionally.

When buying many licenses, you are welcome to [contact](#) our sales for available volume discounts.

## Valid TeamCity Versions

TeamCity licenses are perpetual for the TeamCity versions they cover. This means that you can run a covered TeamCity version with existing licenses for unlimited time and the licenses will stay valid for this TeamCity version.

Each TeamCity license (including Enterprise Server and Agent) has a maintenance period (generally 1 year). The license key is valid with any TeamCity version released within the maintenance period. Licenses valid for the major/minor release (changes in the first two release numbers) are also considered valid for the corresponding bugfix updates (changes in the third release number).

Before you [upgrade](#) to a newer TeamCity version, please check the validity of the existing licenses with the new version. If the new TeamCity server effective [release date](#) is not covered by the maintenance period of some of the licenses, the corresponding licenses will not be valid with the TeamCity version and would need a [renew](#). Generally, license renew costs about 50% of the new license price in case the new license date is the same as the end of maintenance of the license being renewed.

When a new version is available, TeamCity displays a notification in the web UI and warns you if any of your license keys are incompatible with this new version. A notification on the new TeamCity version is also displayed in the Global Configuration Items of the [Server Health](#) report, visible to system administrators. System administrators can use the link in the "Some Licenses are incompatible" message to quickly navigate to the [Licenses](#) page, where all incompatible licenses will have a warning icon. The information about the license keys installed on your server is secure as it is not sent over the Internet.

Regular upgrades are recommended as new releases contain lots of fixes (and of course new features).

Please note that TeamCity [email support](#) covers only the [recent TeamCity versions](#) and can be provided only to customers with not expired maintenance period of the enterprise server license.

## License Expiration

If an Enterprise license key is removed from the server, or an evaluation license expires, or a TeamCity server is upgraded to a version released out of the maintenance window of the available Enterprise license, TeamCity automatically switches to the Professional mode.

If the number of build configurations or the number of authorized agents exceeds the limits imposed by the valid licenses, the server stops to start any builds (pauses the build queue) and displays a warning message to all users in the web browser.

Build Agent Licenses work the same way as the Server Licenses. If you upgrade the server to the version which is not covered by the agent license maintenance window, then this agent license will expire.

Once sufficient valid license keys are entered which cover the server configuration, the builds begin to be started again.

## Ways to Obtain a License

The following ways to switch your server into the Enterprise mode exist:

- [buy](#) an Enterprise Server license;
- request a 60-days evaluation license on the [download page](#) (see details [above](#));
- use a TeamCity [EAP release](#) (not stable, but comes bundled with a 60-day nonrestrictive license);
- use TeamCity for open-source projects only and [request an open-source license](#).

## Upgrading From Previous Versions

### Upgrading from TeamCity 5.x and later

Each license has a maintenance period (typically one year since the purchase date). The license is suitable for any TeamCity version released within the maintenance period. Please check the maintenance period of your licenses before upgrading.

### Upgrading from TeamCity 4.x to TeamCity 5.0 and later

Licenses for previous versions of TeamCity needs upgrading, see details at [Licensing and Upgrade](#) section on the official site.

### Upgrading from TeamCity 3.x to TeamCity 4.0

Owners of TeamCity 3.x Enterprise Server Licenses upgrade to TeamCity 4.x Enterprise Edition free of charge. TeamCity 3.x Build Agent Licenses are compatible with both Professional and Enterprise editions of TeamCity 4.0.

### Upgrading from TeamCity 1.x-2.x to TeamCity 4.0

Any TeamCity 1.x-2.x license purchased before December, 05, 2008 can be used as one TeamCity 4.0 Build Agent license for both Professional and Enterprise editions of TeamCity 4.0. Additionally, TeamCity 1.x-2.x customers qualify for one TeamCity Enterprise Server License free of charge. To request your Enterprise Server License, please contact [sales department](#) with one of your TeamCity 1.x-2.x licenses.

### Upgrading with IntelliJ IDEA 6.0 License Key

Any IntelliJ IDEA 6.0 license purchased between July 12, 2006 and January 15, 2007 can be used as one TeamCity 4.0 Build Agent license. Additionally, IntelliJ IDEA customers with such licenses qualify for one TeamCity Enterprise Server license free of charge.

To check TeamCity upgrade availability for your IntelliJ IDEA licenses and to request your Enterprise Server license, please contact [sales department](#) with one of your IntelliJ IDEA licenses purchased within the above period.

See also:

Concepts: Build Agent  
Licensing: [Licensing & Upgrade](#)

## Third-Party License Agreements

The following are two alphabetical lists of third-party libraries distributed with TeamCity. See also the [TeamCity Home/license](#) [s](#) directory.

## Core libraries

Product	License
Acegi Security	Apache
Apache Ant	Apache
Apache Commons libraries	Apache
ApacheDS	Apache
Apache HttpComponents	Apache
Apache Ivy	Apache
Apache Jakarta Project	Apache
Apache Log4j 2	Apache
Apache log4net	Apache
Apache Lucene	Apache
Apache Olingo	Apache 2.0
Apache ServiceMix	Apache 2
Apache Tomcat	Apache
Apache Xerces2 Java Parser	Apache
AWS SDK for Java	Apache 2
Babel-runtime	MIT
Behaviour	BSD
Byteman	GNU LGPL
CassiniDev	Ms-PL
CodeMirror	MIT-style
Core4j	Apache 2.0
cglib	Apache
CVS client library	CDDL
CyberNekoHTML Parser	Apache-style
DHTML Tip Message	
EhCache	Apache
Expat XML Parser Toolkit	MIT
Flot	MIT
FormattedDataSet API	3-Clause BSD
Font Awesome	SIL OFL 1.1
FreeMarker	BSD-style
Ganymed SSH-2 for Java	BSD-style
google-gson	Apache
Guava: Google Core Libraries	Apache
Highlight.js	BSD-like

HSQldb	BSD
Jackson	Apache
Jakarta-ORO	Apache
JAMon	BSD-like
JAXB reference implementation	CDDL v1.0
JDK by Oracle	Oracle
Java Mail	CDDL v1.0
Java Native Access (JNA)	LGPL
Java Service Wrapper, version 3.2.3	Java Service Wrapper License
IntersectionObserver polyfill	W3C
JCIFS	GNU LGPL
JDom	JDom
Jersey	CDDL v1.0, CDDL v1.1
JFreeChart	GNU LGPL
JHighlight	CDDL
jMock	jMock
JNIWrapper	JNIWrapper
jQuery	MIT
jQuery Flot plugin	MIT
jQuery UFD plugin	MIT
Joda Time	Apache
JSch	BSD-Style
Json.NET	MIT
JUnit	CPL
just-throttle	MIT
J2SSH Maverick	GPL
Logstash Log4J Layout	Apache 2.0
Managed Stack Explorer	Ms-PL
Maragogype	Apache
Maven	Apache
Metrics	CPL-1
Microsoft.Web.Infrastructure	MVC-3-EULA
Microsoft TFS SDK for Java	MIT
Missing Link Ant Tasks	Apache
Mono.Cecil	MIT/X11
NanoContainer	NanoContainer
NUnit	NUnit

OData4j	Apache 2.0
opencsv	Apache
pack:tag	LGPL
Paul Johnson's MD5	BSD
Perf4J	Apache 2
PicoContainer	PicoContainer
PocketHTTP	Mozilla
PocketXML-RPC	Mozilla
Polymer Project	3-Clause BSD
PostgreSQL Data Base Management System	PostgreSQL License
Prototype	MIT
pty4j	EPL
Raphael	MIT
Rome	Apache
RouteMagic	Ms-PL
Script.aculo.us	MIT License
SilverStripe Unobtrusive Javascript Tree Control	BSD
Shaj	Apache
Slf4j	MIT
Smack	Apache
Spring Framework	Apache
Code snippets from Subclipse	EPL
SVNKit	TMate Open Source
Swagger	Apache
typica	Apache
Tom Wu's jsbn	BSD
Trove High Performance Collections for Java	GNU LGPL
Underscore.js	MIT
UserAgentUtils, version 1.12	user-agent-utils
Waffle	EPL
WebActivator	MS-PL
Winp, version 1.7 (patched)	MIT
Xerces Java Parser	Apache
XML Pull Parser	Extreme! Lab, Indiana University License
XML-RPC.NET	MIT X11
XStream	BSD
XZ	XZ license

yavijava	3-Clause BSD
YUI Compressor	BSD

## JavaScript-related libraries

Product	License
@atlaskit/logo	Apache-2.0
@babel/runtime	MIT
@jetbrains/babel-runtime	MIT
@jetbrains/icons	Apache-2.0
@jetbrains/ring-ui	Apache-2.0
babel-runtime	MIT
batch-processor	MIT
callbag-debounce	MIT
callbag-pipe	MIT
callbag-subject	MIT
callbag-subscribe	MIT
change-emitter	MIT
classnames	MIT
combokeys	Apache-2.0
conic-gradient	MIT
core-js	MIT
dom-helpers	MIT
dom4	MIT
element-resize-detector	MIT
fbjs	MIT
history	MIT
hoist-non-react-statics	BSD-3-Clause
intersection-observer	W3C
invariant	MIT
irregular-plurals	MIT
is-plain-object	MIT
isobject	MIT
just-debounce-it	MIT
just-throttle	MIT
lodash-es	MIT
lodash.every	MIT
lodash.filter	MIT

lodash.isequal	MIT
lodash.isfunction	MIT
lodash.isstring	MIT
lodash.keys	MIT
lodash.pick	MIT
lodash.some	MIT
lodash.union	MIT
moment	MIT
normalizr	MIT
object-assign	MIT
plur	MIT
process	MIT
prop-types	MIT
react	MIT
react-dom	MIT
react-is	MIT
react-lifecycles-compat	MIT
react-measure	MIT
react-redux	MIT
react-virtualized	MIT
recompose	MIT
redux	MIT
redux-query-sync	Unlicense
redux-thunk	MIT
regenerator-runtime	MIT
reselect	MIT
resize-observer-polyfill	MIT
resolve-pathname	MIT
shallowequal	MIT
sniffr	MIT
styled-components	MIT
stylis	MIT
stylis-rule-sheet	MIT
svg-baker-runtime	MIT
svg-sprite-loader	MIT
symbol-observable	MIT
url-search-params	MIT

util-deprecate	MIT
uuid	MIT
value-equal	MIT
warning	BSD-3-Clause
whatwg-fetch	MIT
why-did-you-update	MIT

## Acknowledgements

This product includes software developed by Spring Security Project (<http://acegisecurity.org>). (Acegi Security)  
 This product includes software developed by The Apache Software Foundation (<http://www.apache.org/>). (Apache Ant, Ivy, Jakarta, Log4j, Maven2, Tomcat, and Xerces2 Java Parser)  
 This product includes software developed by the DOM Project (<http://www.jdom.org/>). (JDom)  
 This product includes software developed by the Visigoth Software Society (<http://www.visigoths.org/>). (FreeMarker)  
 This product includes software from the Cryptix project <http://www.cryptix.org/>  
 This product includes software from the following Codehaus hosted projects: [Woodstox XML Processor](#), [StAX API](#).

Behaviour JavaScript Copyright © 2005 Ben Nolan and Simon Willison.

CyberNeko Copyright © 2002-2005, Andy Clark. All rights reserved.  
 Expat XML Parser Toolkit Copyright © 1998, 1999, 2000 Thai Open Source Software Center Ltd.  
 Ehcache Copyright 2003-2008 Luck Consulting Pty Ltd  
 Highlight.js Copyright © 2006 Ivan Sagalaev. All rights reserved.  
 HSQLDB Copyright © 1995-2000 by the Hypersonic SQL Group, © 2001-2005 by The HSQL Development Group. All rights reserved.  
 JAMon Copyright © 2002, Steve Souza ([admin@jamonapi.com](mailto:admin@jamonapi.com)).  
 Java Service Wrapper Copyright © 1999, 2006 Tanuki Software, Inc.  
 JDOM Copyright © 2000-2004 Jason Hunter & Brett McLaughlin. All rights reserved.  
 JFreeChart Copyright © 2000-2007, by Object Refinery Limited and Contributors.  
 JMock Copyright © 2000-2007, [jMock.org](http://jmock.org). All rights reserved.  
 Maverick Copyright © 2001 [Infohazard.org](http://Infohazard.org).  
 NanoContainer Copyright © 2003-2004, NanoContainer Organization. All rights reserved.  
 Paul Johnson's MD5 Copyright © 1998 - 2002, Paul Johnston & Contributors. All rights reserved.  
 Portions of PostgreSQL Copyright © 1996-2005, The PostgreSQL Global Development Group or Portions Copyright © 1994, The Regents of the University of California.  
 Raphaël © 2008 Dmitry Baranovskiy  
 Rome is Copyright © 2004 Sun Microsystems, Inc.  
 Script.aculo.us Copyright © 2005 Thomas Fuchs (<http://script.aculo.us>, <http://mir.aculo.us>).  
 Shaj is Copyright Cenqua Pty Ltd.  
 SilverStripe Unobtrusive Javascript Tree Control Copyright © 2006-7, SilverStripe Limited - [www.silverstripe.com](http://www.silverstripe.com).  
 Portions Copyright © 2002 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov or Copyright © 2000-2002 Philip A. Craig  
 Portions Copyright © 2002-2007 Charlie Poole or Copyright © 2002-2004 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov or Copyright © 2000-2002 Philip A. Craig  
 Smack is Copyright © 2002-2007 Jive Software.  
 SVNKit Copyright © 2004-2006 TMate Software. All rights reserved.  
 Tom Wu's isbn Copyright © 2003-2005 Tom Wu. All Rights Reserved.  
 Trove Copyright © 2001, Eric D. Friedman All Rights Reserved.  
 Underscore.js © 2011 Jeremy Ashkenas, DocumentCloud Inc.  
 UserAgentUtils Copyright © 2013, Harald Walker ([bitwalker.eu](http://bitwalker.eu)) All Rights Reserved.  
 XML Pull Parser: Copyright © 2002 Extreme! Lab, Indiana University. All rights reserved. This product includes software developed by the Indiana University Extreme! Lab (<http://www.extreme.indiana.edu/>).  
 XML-RPC.NET Copyright © 2006 Charles Cook.  
 XStream Copyright © 2003-2006, Joe Walnes, Copyright © 2006-2015, XStream Committers. All rights reserved.  
 JBoss Byteman Copyright © 2008-9, Red Hat Middleware LLC, and individual contributors by the @authors tag. See the copyright.txt in the distribution for a full listing of individual contributors

## Integrating TeamCity with Other Tools

In this section:

- [Integrating TeamCity with VCS Hosting Services](#)
- [Integrating TeamCity with Issue Tracker](#)
- [Mapping External Links in Comments](#)
- [External Changes Viewer](#)
- [Integrating TeamCity with Docker](#)

# Integrating TeamCity with VCS Hosting Services

If you have an organization account in [GitHub](#), [GitHub Enterprise](#) or [Bitbucket Cloud](#), you can connect TeamCity to these source code hosting services making it easier for the organization users to create new projects, [Git](#) or [Mercurial](#) VCS roots, [GitHub](#) or [Bitbucket](#) issue tracker, which are now supported out of the box.

It is also possible to connect TeamCity to [Visual Studio Team Services](#) making it really simple to set up projects which use VSTS repositories or issue tracker.

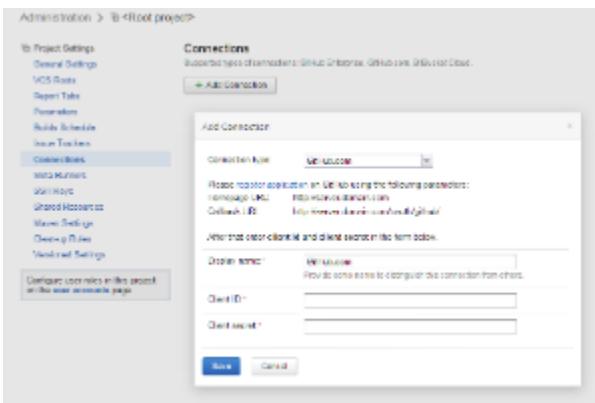
On this page:

- [Configuring Connections](#)
  - [Connecting to GitHub](#)
  - [Connecting to Bitbucket](#)
  - [Connecting to Visual Studio Team Services](#)
- [Creating Entities from URL in TeamCity](#)

## Configuring Connections

Connections are created on the project level, a configured connection is accessible in the current project and in all of its subprojects. If you use global VCS hosting services like GitHub or Bitbucket Cloud, it makes sense to configure a single connection for the Root project. Or your organization administrator can create a parent project and a configure connection to GitHub there once, and the users will see a list of GitHub repositories URLs in the TeamCity Web UI, which will make setting up subprojects a lot simpler for them.

Connections are configured on the Project Administration | Connections page.



You need to register your TeamCity application in your VCS hosting service using the information provided by TeamCity, enter the access details provided by the service in the TeamCity form, and log in to your VCS hosting service from TeamCity to authorize the TeamCity application in the VCS. See details below.

## Connecting to GitHub

You need to configure a connection to your GitHub repository to [create project from URL](#), [create VCS root from URL](#), [create a Git VCS root](#) or [create GitHub issue tracker](#).

To configure a GitHub connection:

1. On the Project Administration page, use the Connections menu item and click the Add Connection button.
2. Select GitHub as the connection type.  
The page that opens provides the parameters to be used when registering your TeamCity application in GitHub service.
3. Click the register application link. The GitHub page opens.  
You need to register TeamCity as an [OAuth application](#) in GitHub. The following steps are performed in your GitHub account:
  - a. Log into your GitHub account. On the Register a new OAuth application page specify the name and an optional description, the homepage URL and the callback URL as provided by TeamCity.
  - b. Click Register application. The page is updated with Client ID and the client secret information for your TeamCity application.
4. Continue configuring the connection in TeamCity: on the Add Connection page that is open, specify the Client ID and the client secret.
5. Save your settings.
6. The connection is configured, and now a small GitHub icon becomes active in several places where a repository URL can be specified: [create project from URL](#), [create VCS root from URL](#), [create Git VCS root](#), [create GitHub issue tracker](#). Click the icon, log in to GitHub and authorize TeamCity. The authorized application will be granted full control of private repositories and the Write repository hooks permission.

## Connecting to Bitbucket

You need to configure a connection to your public Bitbucket repository to [create project from URL](#), [create VCS root from URL](#), [create Mercurial VCS root](#), or [create a Bitbucket issue tracker](#).

To configure a Bitbucket connection:

1. On the Project Administration page, use the Connections menu item and click the Add Connection button.
2. Select Bitbucket as the connection type.  
The page that opens provides the parameters to be used when registering an OAuth consumer on Bitbucket Cloud. Click the register application link.  
You need to create an [OAuth consumer](#) on Bitbucket Cloud. The following steps are performed in your Bitbucket account:
  - a. Log into your Bitbucket account, click your avatar and select Bitbucket settings from the menu. The Account page appears.
  - b. Click OAuth from the menu bar. On the Add OAuth consumer page specify the name and an optional description, the callback URL and the URL as provided by TeamCity.
  - c. Specify the set of permissions: TeamCity requires "read" access to your account and your repositories.
  - d. Save your settings.
  - e. On the page that opens, in the OAuth consumers section, click the name of your TeamCity application to display the key and the secret.
3. Continue configuring the connection in TeamCity: on the Add Connection page that is open, specify the key and secret.
4. Save your settings.
5. The connection is configured, and now a small Bitbucket icon becomes active in several places where a repository URL can be specified: [create project from URL](#), [create VCS root from URL](#), [create Mercurial VCS root](#), [create Bitbucket issue tracker](#). Click the icon, log in to Bitbucket and authorize TeamCity. TeamCity will be granted access to your public repositories. For private repositories you'll still have to sign in to Bitbucket as it doesn't provide non-expiring access tokens. See the [related discussion](#).

## Connecting to Visual Studio Team Services

Since TeamCity 2017.1, you can configure a connection to your Visual Studio Team Services to [create project from URL](#), [create VCS root from URL](#), [create TFS VCS root](#), or [create Team Foundation Work Items](#) tracker.

To configure a connection to Visual Studio Team Services:

1. On the Project Administration page, use the Connections menu item and click the Add Connection button.
2. Select Visual Studio Team Services as the connection type.  
The page that opens provides the parameters to be used when connecting TeamCity to Visual Studio Team Services.
3. Log in to your Visual Studio Team Services account to create a personal access token with All scopes as described in the [Microsoft documentation](#).
4. Continue configuring the connection in TeamCity: on the Add Connection page that is open, specify
  - the server URL in the `https://{{account}}.visualstudio.com` format or your Team Foundation Server web portal as `https://{{server}}:8080/tfs/`.
  - your personal access token.
5. Save your settings.
6. The connection is configured, and now a small Visual Studio Team Services icon becomes active in several places where a repository URL can be specified: [create project from URL](#), [create VCS root from URL](#), [create TFS VCS root](#), [create Team Foundation Work Items](#) tracker. Click the icon, log in to Visual Studio Team Services and authorize TeamCity. TeamCity will be granted full access to all of the resources that are available to you.  
Since TeamCity 2017.2 EAP1, when configuring Commit Status Publisher for Git repositories hosted in TFS/VSTS, the personal access token can be filled out automatically if a VSTS project connection is configured.



It is possible to configure several VSTS connections. In this case the server URL will be displayed next to the VSTS icon to distinguish the server in use.

## Creating Entities from URL in TeamCity

Now creating entities from a URL in TeamCity is extremely easy: on clicking the GitHub, Bitbucket or VSTS icon, the list of repositories available to the current user is displayed (note that only public Bitbucket repositories will be available via the configured connection):

Parent project: \* <Root project>

Repository URL: \* A VCS repository URL. Supported formats: http(s)://, svn://, git://

Username: Optional. Provide username if access to repository requires authentication.

Password: Optional. Provide password if access to repository requires authentication.

**Found 2 repositories**

- TeamCityDotNetContrib ([https://bitbucket.org/pavel\\_sher/teamcity.dotnetcontrib](https://bitbucket.org/pavel_sher/teamcity.dotnetcontrib))
- TestGitRepo ([https://bitbucket.org/pavel\\_sher/testgitrepo](https://bitbucket.org/pavel_sher/testgitrepo))

**Proceed** **Cancel**

You can select the URL and proceed with the configuration.

## Integrating TeamCity with Issue Tracker

TeamCity can be integrated with your issue tracker to provide a comprehensive view of your development project. TeamCity detects issues mentioned in the comments to version control changes, turning them into links to your issue tracker in the TeamCity Web UI.

The integration is configured at the project level: the Project Administrator permissions are required. You can configure integration if you have multiple projects on both the TeamCity and the issue tracker server or if you are using different issue-trackers for different projects.

Enabling integration for the project also enables it for all its subprojects; if the configuration settings are different in a subproject, its settings have priority over the project's settings.

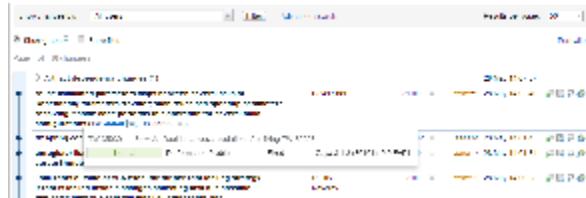
In this section:

- Dedicated Support for Issue Trackers
- Recommendations on Using Issue Tracker Integration
- Enabling Issue Tracker Integration
  - Requirements
  - Configuring connection
- Integrating TeamCity with Other Issue Trackers

### Dedicated Support for Issue Trackers

TeamCity supports JIRA, Bugzilla, YouTrack and since TeamCity 10.0 GitHub, Bitbucket Cloud and TFS trackers out of the box. The [Supported Platforms and Environments](#) page lists supported versions.

When an integration is configured, TeamCity automatically transforms an issue ID (=issue key in JIRA, work item id in TFS) mentioned in the VCS commit comment into a link to the corresponding issue and the basic issue details are displayed in the TeamCity Web UI when hovering over the icon next to the issue ID (e.g. on the [Changes](#) tab of the build results).



Issues fixed in the build can also be viewed on the [Issues](#) tab of the build results.

On the build configuration home page, you can review all the issues mapped to the comments at the [Issue Log](#) tab. You can filter the list to a particular range of builds and view issues mentioned in comments with their states.



## Recommendations on Using Issue Tracker Integration

To get maximum benefit from the issue tracker integration, do the following:

- When committing changes to your version control, always mention the issue id (issue key) related to the fix in the comment to the commit.
- Resolve issues when they are fixed (the time of resolve does not really matter).
- Use Issue Log of a build configuration to get issues related to builds; turn on the "Show only resolved issues" option to only display the issues fixed in the builds.

## Enabling Issue Tracker Integration

### Requirements

The information about the issues is retrieved by the TeamCity server using the credentials provided and then is displayed to TeamCity users.

This has several implications:

- The TeamCity server needs direct access to the issue tracker. (Also, TeamCity does not yet support proxy for connections to issue trackers).
- The user configured in the connection to the issue tracker has to have sufficient permissions to view the issues that can be mentioned in TeamCity. Also, TeamCity users will be able to view the issue details in TeamCity for all the issues that the configured user has access to.

### Configuring connection

To enable integration, you need to create a connection to your issue tracker on the Project Settings | Issue Trackers page. The settings described below are common for all issue trackers:

Connection type	Select the type of your issue tracker from the list.
Display name	A symbolic name that will be displayed in the TeamCity UI for the issue tracker.
Server URL (Repository URL)	The Issue tracker URL
Username/Password (Authentication)	Credentials to log in to the issue tracker, if it requires authorization.

Additional authentication information or/and the details on how to specify strings to be recognized by TeamCity and converted to your tracker issue links is provided in the corresponding sections:

- YouTrack
- JIRA
- Bugzilla
- GitHub
- Bitbucket Cloud
- TFS

## Integrating TeamCity with Other Issue Trackers

To integrate TeamCity with other issue trackers, configure TeamCity to turn any issue tracker issue ID mentions in change comments into links. See [mapping external links in comments](#) for details.

Dedicated support for an issue tracker can also be added via a custom [issue tracker integration plugin](#).

See also:

[Concepts: Supported Issue Trackers](#)  
[Administrator's Guide: Mapping External Links in Comments](#)  
[Extending TeamCity: Issue Tracker Integration Plugin](#)

## Bugzilla

### Converting Strings into Links to Issues

When [enabling issue tracker integration](#) in addition to general settings, you need to specify which strings are to be recognized as references to issues in your tracker.

For Bugzilla, you need to specify the Issue Id Pattern: a [Java Regular Expression](#) pattern to find the issue ID in the text. The matched text (or the first group if there are groups defined) is used as the issue number. The most common case seems to be `#(\d+)` - this will extract 1234 as issue ID from text "Fix for #1234".

### Requirements

If the username and password are specified, you need to have Bugzilla XML-RPC interface switched on. This is not required if you use anonymous access to Bugzilla without the username and password.

### Known Issues

There are several known issues in Bugzilla regarding XMLs generated for the issues, which makes it hard to communicate with it. However this can usually be fixed by tweaking the Bugzilla configuration.

- If you see a path/to/bugzilla.dtd not found error, this means that the issue XML contains the relative path to the `bugzil1a.dtd` file, and not the URL. To fix that, set the server URL in Bugzilla.
- Sometimes you may see a `SAXParseException` saying that Open quote is expected for attribute "type\_id" associated with an element type "flag". This happens because the generated XML does not correspond to the bundled `bugzilla.dtd`. To fix it, make the `type_id` attribute `#IMPLIED` (optional) in the `bugzilla.dtd` file. The issue and the workaround are described in detail [here](#).

See also:

[Concepts: Supported Issue Trackers](#)  
[Administrator's Guide: Integrating TeamCity with Issue Tracker](#)

## JIRA

### Authentication

When configuring authentication for a JIRA Cloud server, specify your JIRA username, not the email. The username can be found in the user profile.

### Converting Strings into Links to Issues

When [enabling issue tracker integration](#) in addition to general settings, you need to specify which strings should be recognized as references to issues in your tracker.

For JIRA, you need to provide a space-separated list of [Project keys](#). Since TeamCity 2017.1, you can also load all project keys automatically: check the corresponding box and test the connection to your JIRA server. If the connection is successful, the Project keys field will be automatically populated. Newly created projects in JIRA will be detected by TeamCity and the project keys list will be automatically synchronized.

For example, if a project key is WEB, an issue key like WEB-101 mentioned in a VCS comment will be resolved to a link to the corresponding issue.

See also:

[Concepts: Supported Issue Trackers](#)  
[Administrator's Guide: Integrating TeamCity with Issue Tracker](#)

## YouTrack

## Converting Strings into Links to Issues

When enabling issue tracker integration, in addition to general settings, you need to specify which patterns are to be recognized as references to issues in your tracker.

For YouTrack, you need to provide a space-separated list of Project IDs. You can also load all project ids automatically: check Use all YouTrack ids automatically and test connection to your YouTrack server. If the connection is successful, the Project IDs field will be automatically populated. Newly created projects in YouTrack will be detected by TeamCity and the project IDs list will be automatically synchronized.

For example, if a project id is TW, an issue id like TW-18802 mentioned in a VCS comment will be resolved to a link to the corresponding issue.

## Enhancing Integration with YouTrack

YouTrack provides native TeamCity integration which enhances the set of available features. For example:

- YouTrack is able to fill "Fixed in build" field with a specific build number.
- YouTrack allows you to apply commands to issues by specifying them in a comment to a VCS change commit.

To use these features, [configure YouTrack](#).

See also:

[Concepts: Supported Issue Trackers](#)

[Administrator's Guide: Integrating TeamCity with Issue Tracker](#)

## GitHub

Since TeamCity 10.0 TeamCity integrates with GitHub, and the integration with GitHub issue tracker can be set up separately, or as a part of TeamCity integration with [GitHub source code hosting service](#).

 If you were using the TeamCity-GitHub [third-party plugin](#) prior to TeamCity 10.0, you can safely remove it: the built-in TeamCity integration will detect the existing connection to GitHub issue tracker and pick up your settings automatically.

When setting up [integration](#) with GitHub, in addition to the repository URL and other general settings, you need to configure authentication and specify the issue ID pattern.

## Authentication

TeamCity allows you to select whether you want to connect to GitHub anonymously or to be authenticated via username/password or an Access token.

## Converting Strings into Links to Issues

You also need to specify which strings should be recognized as references to issues in your tracker. For GitHub, you need to use the regex syntax, e.g. #(\d+).

TeamCity will resolve the issue number mentioned in a VCS comment and will display a link to this issue in the Web UI (e.g. on the [Changes Page](#), [Issues tab](#) of the [build results page](#)).

## Bitbucket Cloud

Since TeamCity 10.0 TeamCity integrates with Bitbucket Cloud, and the integration with its issue tracker can be set up separately, or as a part of TeamCity integration with [Bitbucket Cloud](#) as a source code hosting service.

When setting up [integration](#) with the Bitbucket issue tracker, in addition to the repository URL and other general settings, you need to configure authentication and specify the issue ID pattern.

 For Bitbucket Cloud team accounts, it is possible to use the team name as the username, and the API key as the password.

## Authentication

TeamCity allows you to select whether you want to connect to Bitbucket issue tracker anonymously or to be authenticated via username/password.

## Converting Strings into Links to Issues

You also need to specify which strings should be recognized as references to issues in your tracker. For Bitbucket, you need to use the regex syntax, e.g. #(\d+).

TeamCity will resolve the issue number mentioned in a VCS comment and will display a link to this issue in the Web UI (e.g. on the [Changes Page](#), [Issues tab](#) of the [build results](#) page).

## Team Foundation Work Items

Since TeamCity 10.0, Team Foundation Work Items tracking is integrated with TeamCity. Supported versions are Microsoft Visual Studio Team Foundation Server 2010-2017, and Visual Studio Team Services.

TFS work items support can be configured on the [Issue trackers](#) page for a project. If a project has a [TFVC](#) root configured, TeamCity will suggest configuring the issue tracker as well.

## Integration

By default, the integration works the same way as the other issue tracker integrations: you need to mention the work item ID in the comment message, so the work items can be linked to builds and the links will be displayed in various places in the TeamCity Web UI.

Additionally, if your changeset has related work items, TeamCity can retrieve information about them even if no comment is added to the changeset. Besides, custom states for resolved work items are supported by TeamCity.

## Settings

Option	Description
Display Name	Specify the tracker connection name
Server URL	Team Foundation Server URL in the following format:  TFS 2010+: <code>http[s]://&lt;host&gt;:&lt;port&gt;/tfs/&lt;collection&gt;/&lt;project&gt;</code> Visual Studio Team Services: <code>https://&lt;account&gt;.visualstudio.com/&lt;project&gt;</code>
Username	Specify a user to access Team Foundation Server. This can be a user name or DOMAIN\UserName string. Use blank to let TFS select a user account that is used to run the TeamCity Server. For VSTS use <a href="#">alternate credentials or tokens</a> .
Password	Enter the password of the user entered above
Pattern	Specify the work item id format in changeset comments in the form of regexp.

Learn more about authentication in [Visual Studio Team Services](#).

## Custom Resolved States

In addition, resolved states in TeamCity can be customized by using the `teamcity.tfs.workItems.resolvedStates` internal property set to "Closed?|Done|Fixed|Resolved?|Removed?" by default.

## Mapping External Links in Comments

TeamCity allows to map patterns in VCS change comments to arbitrary HTML pieces using regular expression search and replace patterns. One of the most common usages is to map an issue ID mentioning into a hyperlink to the issue page in the issue tracking system.

To configure mapping:

1. Navigate to the file [TeamCity data directory](#)/config/main-config.xml
2. Locate section `<comment-transformation>`, or create one under the `<server>` tag, if it doesn't exist (you may refer to the main-config.dtd file for the XML structure definition)
3. Specify the search and replace patterns. For example, you can use the following pattern for enabling JIRA integration:

```

<server>
...
<comment-transformation>
<transformation-pattern
    search="(>|(|\s|^)([A-Z]+-\d+)(\b|$)"
    replace="$1<a target="_blank" title="Click to open this issue a new
window" href="
        http://www.jetbrains.net/jira/browse/$2">$3</a>$3"
    description="JetBrains Jira issue link" />
</comment-transformation>
...
</server>

```

TeamCity can apply several patterns to a single piece of text, if they do not intersect (match different string segments).

 Search & replace patterns have `java.util.regex.Pattern` syntax.

## External Changes Viewer

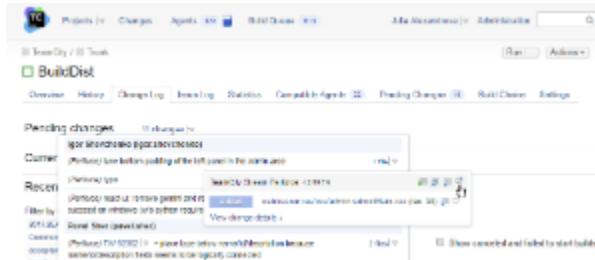
TeamCity supports integration with external changes viewers like JetBrains Upsource, Atlassian Fisheye, etc.

Since TeamCity 2017.2, some of the viewers are supported out of the box: commits to GitHub, VSTS or Bitbucket are automatically recognized and links are provided enabling you to view the changes externally.

To enable other external change viewers, create and configure the `<TeamCity Data Directory>/config/change-viewers.properties` file. These settings should be specified for each VCS root you want to use the external changes viewer for. A detailed example of the configuration file including the description of available formats, variables, and other parameters can be found in the `change-viewers.properties.dist` file in the `<TeamCity Data Directory>/config` directory.

When the configuration file is created, links to the external viewer () will appear on the following pages:

- Changes popups on the Projects and project home page, Overview tab and the Change Log tab of the build configuration home page)



- the Changes tab of the build results page
- the Change details page available by clicking the link when hovering over the changes on the Overview and Change Log tabs for a project and build configurations and on the Changes tab of the build results page
- the TeamCity file diff page.

## Integrating TeamCity with Docker

- Supported Environments
- Parameters Reported by Agent
- Features
  - Docker Support Build Feature
    - Clean-up of images
    - Automatic Login to/Logout of Docker Registry
  - Docker Connection for a Project
    - Registry Address Format
    - Connecting to Private Cloud Registry
    - Connecting to Insecure Registry

- Docker Runner
  - Docker Command
  - Docker Command Parameters
- Docker Compose Runner
- Docker Wrapper
  - Docker Settings
- Docker Disk Space Cleaner
- Service Message to Report Pushed Image

TeamCity comes with Docker Support, implemented as a bundled [plugin](#).



#### Requirements

The integration requires [Docker](#) installed on the build agents. [Docker Compose](#) also needs to be installed to use the [Docker Compose](#) build runner.

## Supported Environments

TeamCity-Docker support can run on Mac, Linux, and Windows build agents. It uses the '`docker`' executable on the build agent machine, so it should be runnable by the build agent user.



- On Linux, the integration will run if the installed Docker is detected.
- On Windows, the integration works in the Windows container mode only. Docker on Windows with the Linux container mode enabled is not supported, an [error](#) is reported in this case.
- On macOS, the official [Docker support for Mac](#) should be installed for the user running the build agent.

## Parameters Reported by Agent

During the build, the build agent reports the following parameters:

Parameter	Description
<code>docker.version</code>	The Docker Engine version
<code>dockerCompose.version</code>	The Docker Compose file version if the <a href="#">Docker Compose</a> build step is used
<code>docker.server.osType</code>	The Docker server OS type, can have the <code>linux</code> or <code>windows</code> value

If you are using the [Command Line Build step](#) (and not the TeamCity-provided docker steps), these parameters can be used as [agent requirements](#) to ensure your build is run only on the agents with Docker installed.

## Features

TeamCity-Docker integration provides the following features which facilitate working with Docker under TeamCity:

### Docker Support Build Feature

[Adding this build feature](#) will enable docker events monitoring: such operations as `docker pull`, `docker run` will be detected. The build feature adds the Docker Info tab to the [build results](#) page providing information on Docker-related operations. It also provides the following options:

- the ability to clean-up the images
- automatic login to an authenticated registry before the build and logout of it after the build

These options require a configured connection to a docker registry.

### Clean-up of images

If you have a build configuration which publishes images, you need to remove them at some point. You can select the corresponding option and instruct TeamCity to remove the images published by a certain build when the build itself is [cleaned up](#). It works as follows: when an image is published, TeamCity stores the information about the registry of the images published by the build. When the [server clean-up](#) is run and it deletes the build, all the configured connections are searched for the address of this registry and the images published by the build are cleaned up using the credentials specified in the connection found.

### Automatic Login to/Logout of Docker Registry

If you need to log in to a registry requiring authentication before a build, select the corresponding option and a connection to Docker configured in the project settings. Automatic logout will be performed after the build finishes.

## Docker Connection for a Project

The Project Settings | Connections page allows you to configure a connection to [docker.io](https://docker.io) (default) or a private Docker registry. More than one connection can be added to the project. The connection will be available in all the subprojects and build configurations of the current project.

### Registry Address Format

By default, <https://docker.io> is used.

To connect to a registry, use the following format: [http(s)://]hostname:port.

If the protocol is not specified, the connection over https is used by default.

### Connecting to Private Cloud Registry

- TeamCity supports the Azure container registry storing Docker images; you can authenticate using [Service principal](#) (the principal ID and password are used as connection credentials) or [Admin account](#).
- Since TeamCity 2018.1, Amazon Elastic Container Registry (AWS ECR) is supported: specify the AWS region and your AWS Security Credentials when configuring the connection.

### Connecting to Insecure Registry

To connect to an insecure registry:

- Configure all TeamCity agents where Docker is installed to work with insecure repositories as stated [Docker documentation](#). This is sufficient to allow the connection to the private registry over http.
- To connect to an insecure registry over https with a self-signed certificate, in addition to the step above, import the self-signed certificate to the JVM of the TeamCity server as described [here](#). You can consult the Docker documentation on [using self-signed certificates](#).

## Docker Runner

The Docker runner supports the `build`, `push`, `tag` Docker commands.

When creating TeamCity projects/ build configurations from a repository URL, the runner is offered as build step during auto-detection, provided a Dockerfile is present in the VCS repository.

Setting	Description	
Docker Command	Select the docker command. Depending on the selected command, the settings below will vary.	
Docker Command Parameters		
build	Dockerfile source	Depending on the selected source, the settings below will vary. The available options include File, a URL or File content.
	Path to file	Available if File is selected as the source. Specify the path to the Docker file. The path should be relative to the <a href="#">checkout directory</a> .
	Context folder	Available if File is selected as the source. Specify the context for the docker build. If blank, the enclosing folder for Dockerfile will be used.
	URL to file	Available if URL is selected as the source. The URL can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts, and plain text files. See <a href="#">Docker documentation</a> for details.
	File Content:	Available if the file content is selected as the source. You can enter the content of the Dockerfile into the field.
	Image name:tag	Provide a newline-separated list of image name:tag(s)

	Additional arguments for 'build' command	Supply additional arguments to the docker build command. See <a href="#">Docker documentation</a> for details.
push	Remove image from agent after push	If selected, TeamCity will remove the image with <code>docker rmi</code> at the end of the step
	Image name:tag	Provide a newline-separated list of image name:tag(s)
other	Command name	Docker sub-command, like <code>push</code> or <code>tag</code> . For run, use <a href="#">Docker Wrapper</a>
	Working directory	
	Additional arguments for the command	Additional arguments that will be passed to the docker command.

## Docker Compose Runner

The runner allows starting [Docker Compose](#) build services and shutting down those services at the end of the build.

The Docker Compose runner supports one or several [Docker Compose YAML file](#)(s) with a description of the services to be used during the build. The path to the docker-compose.yml file(s) should be relative to the [checkout directory](#). When specifying several files, separate them with a space.

The executed commands are

```
# The commands are executed with the current working directory, where the docker-compose file resides.
docker-compose -f <docker-compose.yml> [-f <docker-compose2.yml>] up -d
# At the end of the build, for each docker compose build step the build agent will run:
docker-compose -f <docker-compose.yml> [-f <docker-compose2.yml>] down -v
```

When using Docker Compose with images which support [HEALTHCHECK](#), TeamCity will wait for the `healthy` status of all containers, which support this parameter.

If the start of Docker Compose was successful, the TeamCity agent will register the `TEAMCITY_DOCKER_NETWORK` environment variable containing the name of the Docker Compose default network. This network will be passed transparently to the [Docker Wrapper](#) when used in some build runners.

## Docker Wrapper

TeamCity provides the Docker Wrapper extension for [Command Line](#), [Maven](#), [Ant](#), and [Gradle](#) runners. Each of the supported runners has the dedicated Docker settings section.

### Docker Settings

In this section, you can specify a Docker image which will be used to run the build step.

Setting	Description
Run step within Docker container	Specify a Docker image here. TeamCity will start a container from the specified image and will try to run this build step within this container.
Pull image explicitly (since TeamCity 2017.2)	If the checkbox is enabled, <code>docker pull &lt;imageName&gt;</code> will be run before the <code>docker run</code> command.
Additional docker run arguments	The Edit arguments field allows specifying additional options for <code>docker run</code> . The default argument is <code>--rm</code> .

Technically, the command of the build runner is wrapped in a shell script, and this script is executed inside a Docker container with the `docker run` command. All the details about the started process, text of the script etc. are written into the build log (the Verbose mode enables viewing them).

The [Checkout directory](#) and most build agent directories are mapped inside the Docker process, and TeamCity passes most environment variables from the build agent into the docker process.

After the build step with the Docker wrapper, a build agent will run the `chown` command to restore access of the `buildAgent` user to the checkout directory. This mitigates a possible problem when the files from a Docker container are created with the 'root' ownership and cannot be removed by the build agent later.

If the process environment contains the `TEAMCITY_DOCKER_NETWORK` variable, this network is passed to the started `docker run` command with `--network` switch.

It is possible to provide extra parameters for the `docker run` command, for instance, provide an additional volume mapping.

## Docker Disk Space Cleaner

When there is not enough disk space on the agent at the beginning of the build, the `docker system prune -a` command will be run cleaning the local Docker Caches. The command is run only as the last resort if all other cleaners did not manage to free enough space. Docker Disk Space Cleaner is an extension to the [Free Disk Space](#) build feature ensuring a certain amount of disk space for a build.

## Service Message to Report Pushed Image

If TeamCity (for some reason) cannot determine that an image was pushed, a user can send a special Service Message to report this information to the TeamCity server:

```
# #teamcity[dockerMessage type='dockerImage.push' value='<full_image_tag>,size:<size in bytes>,digest:<hash>']
```

For example:

```
# #teamcity[dockerMessage type='dockerImage.push' value='myRegistry/repo-test:17,size:2632,digest:sha256:8dc5a195c3cdcc7c288d16288ff3f9ab1d8a5a230e09afb9c8dc9215e861aa55']
```

## Configuring Connections to Docker

It is possible to configure a Docker connection in TeamCity if you need [Docker support](#) build feature enabling you to

- log in to an authenticated registry before running a build / log out after the build
- clean up the published images after the build.

The Project Settings | Connections page allows you to configure a connection to [docker.io](#) (default) or a private Docker registry. More than one connection can be added to the project. The connection will be available in all the subprojects and build configurations of the current project.

### Registry Address Format

By default, <https://docker.io> is used.

To connect to a registry, use the following format: [http(s)://]hostname:port.

If the protocol is not specified, the connection over https is used by default.

### Connecting to Private Cloud Registry

- TeamCity supports the Azure container registry storing Docker images; you can authenticate using [Service principal](#) (the principal ID and password are used as connection credentials) or [Admin account](#).

- Since TeamCity 2018.1, Amazon Elastic Container Registry (AWS ECR) is supported: specify the AWS region and your AWS Security Credentials when configuring the connection.

Connecting to Insecure Registry

To connect to an insecure registry:

1. Configure all TeamCity agents where Docker is installed to work with insecure repositories as stated [Docker documentation](#). This is sufficient to allow the connection to the private registry over http.
2. To connect to an insecure registry over https with a self-signed certificate, in addition to the step above, import the self-signed certificate to the JVM of the TeamCity server as described [here](#). You can consult the Docker documentation on [using self-signed certificates](#).

## Docker Wrapper

TeamCity provides the Docker Wrapper extension for [Command Line](#), [Maven](#), [Ant](#), [Gradle](#), and since TeamCity 2018.1, [.NET CLI \(dotnet\)](#) and [PowerShell](#) runners. This extension allows running a build step inside the specified docker image. Each of the supported runners has the dedicated Docker settings section.



### Requirements

The integration requires [Docker](#) installed on the build agents. [Docker Compose](#) also needs to be installed to use the [Docker Compose](#) build runner.

## Supported Environments

TeamCity-Docker support can run on Mac, Linux, and Windows build agents. It uses the '`docker`' executable on the build agent machine, so it should be runnable by the build agent user.



- On Linux, the integration will run if the installed Docker is detected.
- On Windows, the integration works in the Windows container mode only. Docker on Windows with the Linux container mode enabled is not supported, an [error](#) is reported in this case.
- On macOS, the official [Docker support for Mac](#) should be installed for the user running the build agent.

## Docker Settings

In this section, you can specify a Docker image which will be used to run the build step.

Setting	Description
Run step within Docker container	Specify a Docker image here. TeamCity will start a container from the specified image and will try to run this build step within this container.
Pull image explicitly (since TeamCity 2017.2)	If the checkbox is enabled, <code>docker pull &lt;imageName&gt;</code> will be run before the <code>docker run</code> command.
Additional docker run arguments	The Edit arguments field allows specifying additional options for <code>docker run</code> . The default argument is <code>--rm</code> .

Technically, the command of the build runner is wrapped in a shell script, and this script is executed inside a Docker container with the `docker run` command. All the details about the started process, text of the script etc. are written into the build log (the Verbose mode enables viewing them).

The [checkout directory](#) and most build agent directories are mapped inside the Docker process, and TeamCity passes most environment variables from the build agent into the docker process.

After the build step with the Docker wrapper, a build agent will run the `chown` command to restore access of the buildAgent user to the checkout directory. This mitigates a possible problem when the files from a Docker container are created with the 'root' ownership and cannot be removed by the build agent later.

If the process environment contains the `TEAMCITY_DOCKER_NETWORK` variable, this network is passed to the started `docker run` command with `--network` switch.

It is possible to provide extra parameters for the `docker run` command, for instance, provide an additional volume mapping.

## Managing User Accounts, Groups and Permissions

Before creating and managing user accounts, groups and changing users' permissions, we recommend you familiarize yourself with the following concepts:

- [User Account](#)
- [Guest User](#)
- [User Group](#)
- [Role and Permission](#)

These pages contain essential information about user accounts, their roles and permissions in TeamCity, and more.

In this section:

- [Managing Users and User Groups](#)
- [Viewing Users and User Groups](#)
- [Managing Roles](#)

## Managing Users and User Groups

On this page:

- [Creating New User](#)
- [Editing User Account](#)
  - General
    - [Authentication Settings](#)
  - [VCS Usernames](#)
  - [Groups](#)
  - Roles
    - [Assigning Roles to Users](#)
  - [Notification Rules](#)
    - Own rules
    - Inherited rules
- [Managing User Groups](#)
  - [Creating New Group](#)
  - [Editing Group Settings](#)
  - [Adding Multiple Users to Group](#)

### Creating New User

The Administration | Users page provides the Create user account option.

When creating a user account when [several authentication modes enabled](#) on the server, only a username is required.

If only the [default authentication](#) is used, the password is required as well. Any new user is automatically added to the [All Users group](#) and inherits the roles and permissions defined for this group.

If you do not use [per-project permissions](#), you can specify here whether a user should have administrative permissions or not. Otherwise, you can assign roles to this user [later](#).

### Editing User Account

To edit/delete a user account, click its name on the Users tab of the Administration | Users page and use the corresponding option. The page provides several tabs allowing you to modify various user account settings

## General

The General tabs allows modifying the user's name, email address and password if you have appropriate permissions. Users can change their own username only if free registration is allowed. The administrator can always change the username of any user.

### Authentication Settings

The Authentication Settings section appears if several authentication modules are enabled on the server. Here you can edit usernames for different authentication modules such as LDAP and Windows Domain.

### VCS Usernames

This tab allows viewing and editing default usernames for different VCSs used by the current user. Multiple usernames are supported for a VCS root type and for a separate VCS root: several newline-separated values can be used for each VCS username.

The names set here will be used to:

- show builds with changes committed by a user with such a VCS username on the [Changes page](#)
- highlight such builds on the [Projects page](#) if the appropriate [option is selected](#),
- notify the user on such builds when the [Builds affected by my changes](#) option is selected in [notifications settings](#).

### Groups

Use this tab to review the groups the user belongs to, and add/remove the user from groups.

### Roles

This tab is available only if per-project permissions are enabled on the server Administration / Authentication page. Use this tab to view the roles assigned to the user directly and those inherited from groups. The roles assigned directly can be modified/removed here. See also [Assigning Roles to Users](#) below.

### Assigning Roles to Users



To be able to grant roles to users on per-project basis, enable per-project permissions on the Administration | Authentication page.

The Administration | Roles page lists all existing roles detailing their permissions.

There are several ways to assign roles to one or several users:

- To assign a role to a specific user, on the Users tab for the user click View roles in the corresponding column. In the Roles tab, click Assign role.
- To assign a role to multiple users, on the Users tab, check the boxes next to the usernames and use the Assign roles button at the bottom of the page.
- To assign a role to all users in a group, on the Groups tab click View roles for the group in question, then assign a role on the group level.  
When assigning a role, you can:
  - Select whether a role should be granted globally, or in particular projects.
  - Replace existing roles with the newly selected. This will remove all roles assigned to user(s)/group and replace them with the selected one instead.

### Notification Rules

This tab displays [notification rules](#) for the user.

#### Own rules

The rules configured by the user are displayed here and can be modified.

#### Inherited rules

This section displays the rules inherited by the user from the groups they belong to.

## Managing User Groups

### Creating New Group

Open the Administration | Groups page and click Create new group. In the dialog, specify the group name. TeamCity will create an editable Group Key, which is a unique group identifier.

When creating a group, you can select the parent group(s) for it. All roles and notification rules configured for the parent group will be automatically assigned to the current group. To place the current group to the top level, deselect all groups in the list.

### Editing Group Settings

To edit a group, click its name on the Groups tab. You can modify the group name and description as well as parent group(s), and change the list of users, roles and permissions, and notification settings for the group.

 The All Users group includes all users and cannot be deleted. However, you can modify its roles and notification settings.

The Roles tab allows you to view and edit (assign/unassign) default roles for the current group. These roles will be automatically assigned to all users in the group.

Default roles for a user group are divided in two groups:

- roles inherited from a parent group. Inherited roles can not be unassigned from the group.
- roles assigned explicitly to the group

To assign a role for the current group explicitly, click the Assign role link.

To view permissions granted to a role, click the View roles permissions link.

You can also specify notification rules to be applied to all users in the current group. To learn more about notification rules, please refer to [Subscribing to Notifications](#).

### Adding Multiple Users to Group

On the Users page, select users, click the Add to groups button at the bottom, and specify the groups to add the users to. Note that all these users will inherit the roles defined for the group.

#### See also:

[Concepts: User Account | User Group | Role and Permission](#)

## Viewing Users and User Groups

You can view the list of users and user groups registered in the system on the Administration | Users and Groups pages. The content of this page depends on the [authentication settings](#) of server configuration and TeamCity edition you have. For example, user accounts search and assigning roles are not available in TeamCity Professional.

### Searching Users

On the top of the Users and Groups page there's search panel, which allows you to easily find users in question:

- In the Find field you can specify a search string, which can be a user visible name, full name, or email address, or a part of it.
- To narrow down the search you can also restrict it to particular user group, role, or role in specific project using corresponding drop-down lists. By selecting the Invert roles filter option, you can invert search results to show the list of users that do not have the specified role assigned.

#### See also:

[Concepts: User Account | User Group | Role and Permission](#)

## Managing Roles

If [per-project permissions](#) are enabled in your installation, you can view the existing roles, modify them and create new ones in the TeamCity Web UI using the Administration | Roles link (in the User Management section of Settings).

Using the Roles page you can:

- Create new roles.
- Delete existing roles.
- Add/delete permissions from existing roles.
- Include/exclude one role permissions.



The role settings are global.

You can also configure roles and permissions using the `roles-config.xml` file stored in `<TeamCity Data Directory>/config` directory.

See also:

[Concepts: Role and Permission](#)

## Customizing Notifications

TeamCity users can [select the events to be notified about](#). The default notification messages can be customized globally on a per-server basis. Customizing notifications received via Atom/RSS syndication feeds requires a special approach since the feeds use the "pull" model for receiving notifications instead of "push".

On this page:

- [Notifications Lifecycle](#)
- [Customizing Notifications Templates](#)
  - [Notification Templates Location](#)
  - [Supported Output Values](#)
  - [Customization Examples](#)
    - [Including ERRORS from the log](#)
    - [Listing build artifacts](#)
    - [Listing build parameters](#)
  - [Default Data Model](#)
  - [TeamCity Notification Properties](#)
  - [Syndication Feed Template](#)

### Notifications Lifecycle

TeamCity supports a set of events that can generate user notifications (such as build failures, investigation state changes, etc). On an event occurrence, for each notifier type, TeamCity processes notification settings for all the users to determine which users to notify.

When the set of users is determined, TeamCity fills the notification model (the objects relevant to the notification, such as as "build", investigation data, etc.) and evaluates a notification template that corresponds to the notification event. The template uses the data model objects to generate the output values (e.g. notification message text). The output values are then used by the notifier to send the message. Each notifier supports a specific set of the output values.

Please note that the template is evaluated once for an event which means that notification properties cannot be adjusted on a per-user basis.

The output values defined by the template are then used by the notifier to send alerts to the selected users.

### Customizing Notifications Templates

#### Notification Templates Location

Each of the bundled notifier has a directory under `<TeamCity data directory>/config/_notifications/` which stores [FreeMarker](#) (.ftl) templates. There are also .dist files that store the default templates. Each notification type evaluates a template file with a corresponding name. The template files can be modified while the server is running.

By default, the server checks for changes in the files each 60 seconds, but this can be changed by setting the `teamcity.notification.template.update.interval` internal property to the desired number of seconds.

If there occurs an error during the template evaluation, TeamCity logs the error details into `teamcity-notifications.log`. There can be non-critical errors that result in ignoring part of the template or critical errors that result in inability to send notification at all. Whenever you make changes to the notification templates please ensure the notification can still be sent.

This document doesn't describe the FreeMarker template language, so if you need a guidance on the FreeMarker syntax, please refer to the corresponding template manual at <http://freemarker.org/docs/dgui.html>.

## Supported Output Values

TeamCity notifiers use templates to evaluate output values (global template variables) which are then retrieved by name. The following output values are supported:

### Email Notifier

- subject - subject of the email message to send
- body - plain text of the email message to send
- bodyHtml - This is the optional HTML text of the email message to send. It will be included together with plain text part of the message. However, if it is present in the template, it should be customized as well.
- headers - (optional) Raw list of additional headers to include into an email. One header per line. For example:

```
<#global headers>
X-Priority: 1 (Highest)
Importance: High
</#global>
```

### Jabber

- message - plain text of the message to send

### IDE Notifications and Windows Tray Notifications

- message - plain text of the message to send
- link - URL of the TeamCity page that contains detailed information about the event

(i)The Atom/RSS feeds template differs from the others. For the details, please refer to the [dedicated section](#).

## Customization Examples

This section provides Freemarker code snippets that can be used for customization of notifications:

### Including ERRORS from the log

```
<#list build.buildLog.messages[1..] as message><!-- skipping the first message (it is a root node)-->
<#if message.status == "ERROR" || message.status == "FAILURE" >
    ${message.text}
</#if>
</#list>
```

The example below shows the snippet included into the `build_failed.ftl` template: the errors will be listed in both the plain text and the html part of the e-mail:

```

<!-- Uses FreeMarker template syntax, template guide can be found at
http://freemarker.org/docs/dgui.html -->

<#import "common.ftl" as common>

<#global subject>[<@common.subjMarker/> FAILED] ${project.name}:${buildType.name} - Build:
${build.buildNumber}</#global>

<#global body>TeamCity build: ${project.name}:${buildType.name} - Build: ${build.buildNumber}
failed ${var.buildShortStatusDescription}.
Agent: ${agentName}
Build results: ${link.buildResultsLink}

${var.buildCompilationErrors}${var.buildFailedTestsErrors}${var.buildChanges}

<#list build.buildLog.messages[1..] as message><!-- skipping the first message (it is a root
node)-->
<if message.status == "ERROR" || message.status == "FAILURE" >
    ${message.text}
</if>
</list>

<@common.footer/></#global>

<#global bodyHtml>
<div>
    <div>
        TeamCity build: <i>${project.name}:${buildType.name} - <a
href='${link.buildResultsLink}'>Build: ${build.buildNumber}</a></i> failed
        ${var.buildShortStatusDescription}
    </div>
    <@common.build_agent build/>
    <@common.build_comment build/>
    <br>
    <@common.build_changes var.changesBean/>
    <@common.compilation_errors var.compilationBean/>
    <@common.test_errors var.failedTestsBean/>

    <#list build.buildLog.messages[1..] as message><!-- skipping the first message (it is a root
node)-->
        <if message.status == "ERROR" || message.status == "FAILURE" >
            ${message.text}
        </if>
    </list>

    <@common.footerHtml/>
</div>
</#global>

```

## Listing build artifacts

There is no default way of listing build artifacts in an email template at the moment. See a [related issue](#) with a simple plugin allowing you to list artifacts via a relevant API.

### Current workaround

This assumes direct access to disk from the template and ignores external artifacts storage (like S3) as well as artifacts

browsing policies (not showing internal artifacts). Also, this way is likely to become deprecated in the future TeamCity versions.

```
<p>Build artifacts:</p>
<#list build.artifactsDirectory.listFiles() as file>
    <a href="${webLinks.getDownloadArtifactUrl(build.buildTypeExternalId, build.buildId,
file.name)}">${file.name}</a> (${file.length()}B)<br/>
</#list>
```

This will list only the root artifacts and include the `.teamcity` directory, which can be changed by modifications to the code.

## Listing build parameters

```
<#list build.parametersProvider.all?keys as param>
${param} - ${build.parametersProvider.all[param]}
<br>
</#list>
```

This will list the parameters that are passed to the build from the server.

## Default Data Model

For the template evaluation TeamCity provides the default data model that can be used inside the template. The objects exposed in the model are instances of the corresponding classes from [TeamCity server-side open API](#).  
The set of available objects model differs for different events.

You can also add your own objects into the model via plugin. See [Extending Notification Templates Model](#) for details.

Here is an example description of the model (the code can be used in IntelliJ IDEA to edit the template with completion):

```
<!-- @ftlvariable name="project" type="jetbrains.buildServer.serverSide.SProject" -->
<!-- @ftlvariable name="buildType" type="jetbrains.buildServer.serverSide.SBuildType" -->
<!-- @ftlvariable name="build" type="jetbrains.buildServer.serverSide.SBuild" -->
<!-- @ftlvariable name="agentName" type="java.lang.String" -->
<!-- @ftlvariable name="buildServer" type="jetbrains.buildServer.serverSide.SBuildServer" -->
<!-- @ftlvariable name="webLinks" type="jetbrains.buildServer.serverSide.WebLinks" -->

<!-- @ftlvariable name="var.buildFailedTestsErrors" type="java.lang.String" -->
<!-- @ftlvariable name="var.buildShortStatusDescription" type="java.lang.String" -->
<!-- @ftlvariable name="var.buildChanges" type="java.lang.String" -->
<!-- @ftlvariable name="var.buildCompilationErrors" type="java.lang.String" -->

<!-- @ftlvariable name="link.editNotificationsLink" type="java.lang.String" -->
<!-- @ftlvariable name="link.buildResultsLink" type="java.lang.String" -->
<!-- @ftlvariable name="link.buildChangesLink" type="java.lang.String" -->
<!-- @ftlvariable name="responsibility" type="jetbrains.buildServer.responsibility.ResponsibilityEntry" -->
<!-- @ftlvariable name="oldResponsibility" type="jetbrains.buildServer.responsibility.ResponsibilityEntry" -->
```

## TeamCity Notification Properties

The following [properties](#) can be useful to customize the notifications behaviour:

- `teamcity.notification.template.update.interval` - how often the templates are reread by system (integer, in seconds, default 60)
- `teamcity.notification.includeDebugInfo` - include debug information into the message in case of template processing errors (boolean, default false)

```
teamcity.notification.maxChangesNum - max number of changes to list in e-mail message (integer, default 10)
teamcity.notification.maxCompilationDataSize - max size (in bytes) of compilation error data to include in e-mail message (integer, default 20480)
teamcity.notification.maxFailedTestNum - max number of failed tests to list in e-mail message (integer, default 50)
teamcity.notification.maxFailedTestStacktraces - max number of test stacktraces in e-mail message (integer, default 5)
teamcity.notification.maxFailedTestDataSize - max size (in bytes) of failed test output data to include in a single e-mail message (integer, default 10240)
```

## Syndication Feed Template

The template uses different approach to configuration from other notification engines.

The default template is stored in the file: <TeamCity data directory>/config/default-feed-item-template.ftl. This file should never be edited: it is overwritten on every server startup with the default copy. To specify a new template to use, copy the file under the name feed-item-template.ftl into the same directory. This file can be edited and will not be overwritten. It will be used by the engine if present.

The template is a [FreeMarker](#) template and can be freely edited.

You can use several templates on the single sever. The template name can be passed as a [URL parameter](#) of the feed URL.

During feed rendering, the template is evaluated to get the feed content. The resultant content is defined by the global variables defined in the template.

See the default template for an example of available input variables and output variables.

See also:

[User's Guide: Subscribing to Notifications](#)

## Assigning Build Configurations to Specific Build Agents

It is sometimes necessary to manage the [Build Agents](#)' workload more effectively. For example, if the time-consuming performance tests are run, the Build Agents with low hardware resources may slow down. As a result, more builds will enter the [build queue](#), and the feedback loop can become longer than desired. To avoid such situation, you can:

1. Establish a [run configuration policy](#) for an agent, which defines the build configurations to run on this agent.
2. Define [special agent requirements](#), to restrict the pool of agents, on which a build configuration can run the builds.

These requirements are:

- [Build Agent name](#). If the name of a build agent is made a requirement, the build configuration will run builds on this agent only.
- [Build Agent property](#). If a certain property, for example, a capability to run builds of a certain configuration, an operating system etc., is made a requirement, the build configuration will run builds on the agents that meet this requirement.



- You can modify these parameters when setting up the project or build configuration, or at any moment you need. The changes you make to the build configurations are applied on the fly.
- You can specify a particular build agent to run a build on when [Triggering a Custom Build](#).

## Agent pools

You could split agents into pools. Each project could be associated to a number of pools. See [Agent Pools](#).

## Establishing a Run Configuration Policy

To establish a Build Agent's run configuration policy:

1. Click the Agents and select the desired build agent.
2. Click the Compatible Configurations tab.
3. Select Run selected configurations only and tick the desired build configurations names to run on the build agent.

## Making Build Agent Name and Property a Build Configuration Requirement

To make a build configuration run the builds on a build agent with the specified name and properties:

1. Click Administration and select the desired build configuration.
2. Click Agent Requirements (see [Configuring Agent Requirements](#)).
3. Click the Add requirement for a property link, type the agent.name property, set its condition to equals and specify the build agent's name.



#### Note

You can also use the condition contains, however, it may include more than one specific build agent (e.g. a build configuration with a requirement agent.name contains Agent10, will run on agents named Agent10, Agent10a, and Agent10b).

4. Click the Add requirement for a property link and add the required property, condition, and value. For example, if you have several Linux-only builds, you can add the teamcity.agent.jvm.os.name property and set the starts with condition and the linux value.

See also:

[Concepts: Build Agent | Agent Requirements | Run Configuration Policy](#)  
[Administrator's Guide: Triggering a Custom Build](#)

## Patterns For Accessing Build Artifacts



It is recommended to use the TeamCity [REST API](#) for accessing artifacts from scripts, as the REST API provides build selection facilities and allows listing artifacts.

This section is preserved for backward-compatibility with the previous TeamCity versions and for some specific functionality.

Check the following information as well:

- If you need to access the artifacts in your builds, consider using the TeamCity's built-in [Artifact Dependency](#) feature.
- You can also download artifacts from TeamCity using the [Ivy](#) dependency manager.
- For artifact downloads from outside TeamCity builds, consider using [REST API](#).
- See also [Accessing Server by HTTP](#) on basic rules covering HTTP access from scripts.

This page covers:

- [Obtaining Artifacts](#)
- [Obtaining Artifacts from an Archive](#)
- [Obtaining Artifacts from a Build Script](#)
- [Links to the Artifacts Containing the TeamCity Build Number](#)

### Obtaining Artifacts

To download artifacts of the latest builds (last finished, successful or pinned), use the following paths:

```
/repository/download/BUILD_TYPE_EXT_ID/.lastFinished/ARTIFACT_PATH  
/repository/download/BUILD_TYPE_EXT_ID/.lastSuccessful/ARTIFACT_PATH  
/repository/download/BUILD_TYPE_EXT_ID/.lastPinned/ARTIFACT_PATH
```

To download artifacts by the [build id](#), use:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_ID:id/ARTIFACT_PATH
```

To download artifacts by the build number, use:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_NUMBER/ARTIFACT_PATH
```

To download artifacts from the latest build with a specific tag, use:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_TAG.tcbuildtag/ARTIFACT_PATH
```

To download all artifacts in a .zip archive, use:

```
/repository/downloadAll/BUILD_TYPE_EXT_ID/BUILD_SPECIFICATION
```

where

- BUILD\_TYPE\_EXT\_ID is a [build configuration ID](#).
- BUILD\_SPECIFICATION can be .lastFinished, .lastSuccessful or .lastPinned, [specific buildNumber](#) or [build id](#) in format BUILD\_ID:id.
- ARTIFACT\_PATH is a path to the artifact on the TeamCity server. This path can contain a {build.number} pattern (%7B build.number%7D) which will be replaced by TeamCity with the build number of the build whose artifact is retrieved. By default, the archive with all artifacts does not include [hidden artifact](#). To include them, add "?showAll=true" at the end of the corresponding URL.

To download artifact from the last finished, last successful, last pinned or tagged build in a specific branch, add the "?branch=<branch\_name>" parameter at the end of the corresponding URL.

## Obtaining Artifacts from an Archive

TeamCity allows obtaining a file from an archive from the build artifacts directory by means of the following URL patterns:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_SPECIFICATION/<archive>!/PATH_WITHIN_ARCHIVE
```

- BUILD\_TYPE\_EXT\_ID is a [build configuration ID](#).
- BUILD\_SPECIFICATION can be .lastFinished, .lastPinned, .lastSuccessful, [specific buildNumber](#) or [build id](#) in format BUILD\_ID:id.
- PATH\_WITHIN\_ARCHIVE is a path to a file within a zip/7-zip/jar/tar.gz archive on TeamCity server.

Following archive types are supported (case-insensitive):

- .zip
- .7z
- .jar
- .war
- .ear
- .nupkg
- .sit
- .apk
- .tar.gz
- .tgz
- .tar.gzip
- .tar

## Obtaining Artifacts from a Build Script

It is often required to download artifacts of some build configuration by tools like wget or another downloader which does not support HTML login page. TeamCity asks for authentication if you accessing artifacts repository.

To authenticate correctly from a build script, you have to change URLs (add /httpAuth/ prefix to the URL):

```
/httpAuth/repository/download/BUILD_TYPE_EXT_ID/.lastFinished/ARTIFACT_PATH
```

Basic authentication is required for accessing artifacts by this URLs with `/httpAuth/` prefix.

You can use existing TeamCity username and password in basic authentication settings, but consider using `teamcity.auth.use  
rId/teamcity.auth.password` system properties as credentials for the download artifacts request: this way TeamCity will have a way to record that one build used artifacts of another and will display that on build's Dependencies tab.

To enable downloading an artifact with guest user login, you can use either of the following methods:

- Use old URLs without `/httpAuth/` prefix, but with added `guest=1` parameter. For example:

```
/repository/download/BUILD_TYPE_EXT_ID/.lastFinished/ARTIFACT_PATH?guest=1
```

- Add the `/guestAuth` prefix to the URLs, instead of using `guest=1` parameter. For example:

```
/guestAuth/repository/download/BUILD_TYPE_EXT_ID/.lastFinished/ARTIFACT_PATH
```

In this case you will not be asked for authentication.

The list of the artifacts can be found in `/repository/download/BUILD_TYPE_EXT_ID/.lastFinished/teamcity-ivy.xml`.

## Links to the Artifacts Containing the TeamCity Build Number

You can use `{build.number}` (%7Bbuild.number%7D in URL) as a shortcut to current build number in the artifact file name. For example:

```
http://teamcity.yourdomain.com/repository/download/MyConfExtId/.lastFinished/TeamCity-%7Bbuild.  
number%7D.exe
```

See also:

[Concepts: Build Artifact | Authentication Modules](#)  
[Administrator's Guide: Retrieving artifacts in builds](#)  
[Extending TeamCity: Accessing Server by HTTP](#)

## Mono Support

Mono framework is an alternative framework for running .NET applications on both Windows and Unix-based platforms. For more information please refer to the [Mono official site](#).

### Supported Build Runners

TeamCity supports running .NET builds using [NAnt](#) and [MSBuild](#) runners under Mono framework as well as under .NET Frameworks. (MSBuild as xbuild in Mono).

Since TeamCity 2017.1 [NuGet](#) runners support Linux and macOS when [Mono](#) is installed on the agent. Note that only NuGet CLI 3.2+ on Mono 4.4.2+ is supported.

[Tests reporting tasks](#) are also supported under Mono.

### Mono Platform Detection

When a build agent starts, it detects a Mono installation automatically.

On each platform, Mono detection is compatible with NAnt one. See [NAnt.exe.config](#) for frameworks detection on NAnt.

## Agent Properties

When Mono is detected automatically on the agent side, the following properties are set:

- Mono — path to the mono executable (Mono JIT)
- MonoVersion — Mono version
- MonoX.Z — set to `MONO_ROOT/lib/mono/X.Z` if exists
- MonoX.Z\_x64 — set to `MONO_ROOT/lib/mono/X.Z` if exists and Mono architecture is x64
- MonoX.Z\_x86 — set to `MONO_ROOT/lib/mono/X.Z` if exists and Mono architecture is x86

If the Mono installation cannot be detected automatically (for example, you have installed Mono framework into a custom directory), you can make these properties available to build runners by setting them manually in the [agent configuration file](#).

## Windows Specifics

Automatic detection of Mono framework under Windows has the following specifics:

1. The Mono version is read from `HKLM\SOFTWARE\Novell\Mono\DefaultCLR`
2. The Frameworks paths are extracted from `HKLM\SOFTWARE\Novell\Mono\ %MonoVersion%`
3. The platform architecture is detected by analyzing `mono.exe`

## Mac OS X Specifics

1. The framework is detected automatically from `/Library/Frameworks/Mono.framework/Versions`
2. The highest version is selected.
3. The frameworks path are extracted from `/Library/Frameworks/Mono.framework/Versions/%MonoVersion%/lib/mono`
4. The platform architecture is fixed to x86 as Mono official builds support only X86

## Custom Linux/Unix Specifics

Automatic detection of Mono framework under Unix has the following specifics:

1. Mono version is read from "`pkg-config --modversion mono`"
2. The frameworks paths are extracted from "`pkg-config --variable=prefix mono`" and "`pkg-config --variable=libdir mono`"
3. The platform architecture is detected by analyzing the `PREFIX/bin/mono` executable.

You can force Mono to be detected from a custom location by adding the `PREFIX/bin` directory to the beginning of the `PATH` and updating `PKG_CONFIG_PATH` (described in `pkg-config(1)`) with `PREFIX/lib/pkgconfig`.

See also:

[Administrator's Guide: NAnt | MSBuild | NuGet](#)

## Maven Server-Side Settings

### Maven Settings Resolution on the Server Side

The TeamCity server invokes Maven on the server side for functionality like Maven dependency triggers and the Maven model display on the "Maven" build configuration tab.

You can upload Maven settings using the Administration | Project Settings | Maven Settings tab and then select one of the uploaded settings on the [Maven step](#) settings.

During the process, TeamCity uses the usual Maven logic for finding the `settings.xml` files with several differences (see below).

- [Global Settings](#)
- [User-Level Settings](#)

### Global Settings

Maven global-level settings are used from the `.xml` file in the default Maven location for the TeamCity server process: `$(env.M2_HOME)/conf/settings.xml` (or `$(system.maven.home)/conf/settings.xml`) (global values of M2\_HOME environment

nt variable and maven.home JVM option are used - those set for the TeamCity server process),

## User-Level Settings

Maven user-level settings are defined in the User settings selection [section](#) of the Maven build step of the build configuration (if there are several Maven steps, settings from the first one are used).

The following options are available:

Value	Description
<Default>	TeamCity searches the following locations for the <code>settings.xml</code> file (listed in order of priority): <ol style="list-style-type: none"><li>1. <code>&lt;TeamCity Data Directory&gt;/system/pluginData/maven/settings.xml</code></li><li>2. <code>&lt;User Home&gt;/.m2/settings.xml</code> (The home directory of the user under whom the TeamCity server process runs is used)</li></ol>
<Custom>	The path to the file is provided by user. The file should be available both on the server and all the agents where the build will be run.
Uploaded settings name	TeamCity automatically uses the specified file content both on the server and agents. Maven settings are defined on the project level: the Project Settings   Maven Settings tab. The settings are stored in the <code>&lt;TeamCity Data Directory&gt;/config/projects/%projectID%/pluginData/mavenSettings</code> directory.   The settings are available in the current project and its subprojects. To override the inherited settings, in a subproject create a new settings file with the same name as the inherited one.

For the logic of Maven settings, please refer to the related Maven [documentation](#).

User-level settings can be configured in the [Maven Artifact Dependency Trigger](#).

See also:

[Administrator's Guide: Maven | Maven Artifact Dependency Trigger](#)

## Tracking User Actions

TeamCity logs user actions into the Audit log, which is available on the Administration | Audit page. Here you can find out who deleted a build configuration or project, assigned a role to a user, added a user to a group, modified a build configuration, and much more. To find the required information faster, filter the log by the activity type, projects/build configurations, and/or particular users.

If settings of a project or build configuration were modified, you can see the name of user who made the modification and view the change itself by clicking the corresponding link. Project and build configuration settings are stored on the disk in plain xml, so the link will open the usual TeamCity diff window showing changes in these xml files.

You can also view the latest modifications made to a project or build configuration on the project/build configuration settings page by clicking the view history link.

The audit log also can be retrieved in a text form, see the `logs\teamcity-activities.log` file.

### Audit storage period

By default, the audit log keeps records for one year (set to 365 by default) only if clean-up is enabled. Without clean-up, the records are intended to be kept forever.

To modify the audit storage period, specify a the number of days for the following internal property: `teamcity.audit.cleanupPeriod`.

## Jenkins to TeamCity Migration Guidelines

- [Introduction](#)
- [Concepts](#)
- [Migrating Freestyle project](#)
  - [Project and Build Configuration](#)

- VCS Roots
- Build Environment
- Triggers
- Build
- Building Maven project
- Post-build Actions
- Working with branches
- Plugins
- Build pipelines
- Distributed builds

## Introduction

This document provides the basics you need to know when migrating from Jenkins to a TeamCity CI server. TeamCity is quite different from Jenkins in terms of concepts related to managing CI and CD. If you just want to jump in and get started, try to [this guide](#).

## Concepts

Jenkins and TeamCity mostly feature the same set of concepts, however, with slightly different naming. The following table provides a mapping for some of the Jenkins concepts to the TeamCity counterparts.

Jenkins	TeamCity
Jenkins Master/Node	TeamCity server
Dumb Slave / Permanent Agent	Agent Pool
Executor	TeamCity Agent
View or Folder	Project
Job/Item/Project	Build configuration
Build	Build Steps
Pre Step	Build Steps (partially)
Post-build Action	Build Steps (partially)
Build Triggers	Build Triggers
Source Control Management (SCM)	Version Control System (VCS)
Workspace	Build Checkout Directory
Pipeline	Build Chain (via snapshot dependencies)
Label	Agent Requirements

## Migrating Freestyle project

A Freestyle project is the most common project type in Jenkins so we describe the TeamCity counterparts for a Freestyle project to guide the migration.

### Project and Build Configuration

There are some conceptual differences in how the build jobs are configured in Jenkins and TeamCity.

**Build Configuration** is the TeamCity's counterpart of Jenkins' Job/Item/Project. However, a Build Configuration requires a **Project** instance to be created first. In fact, the notion of Project in TeamCity is the first big difference that a user encounters when migrating from Jenkins. The Project contains a good portion of settings required for the build configurations. All settings that are assigned to a Project in TeamCity are listed [here](#).

### VCS Roots

Source Code Management (SCM) of a Jenkins' build job corresponds to a [VCS root](#) in TeamCity's Project settings. A Project may include an arbitrary number of VCS Roots. Any Build Configuration may use any number of VCS Roots of the Project.

When only a part of the VCS Root is required, it is possible to use [VCS Checkout Rules](#). This allows mapping the directories from the configured VCS Root to subdirectories in the [build checkout directory](#) on a Build Agent.

## Build Environment

The Build Environment section of the Freestyle project in Jenkins specifies additional features for the project. In TeamCity, the corresponding features may be configured in various sections, depending on the goal: general project settings, build configuration settings, VCS root settings, etc.

## Examples

- To clean the workspace before the build, enable the "Clean build" checkbox in Version Control Settings of a Build Configuration.
- To cancel the build if the process is stuck, configure [failure conditions](#) of a Build Configuration. What is more, TeamCity provides [hanging build detection](#) out of the box.
- Environment variables can be accessed by specifying a special `%env.<environment variable name>%` pattern (example: `%env.JAVA_HOME%`) in various places of the configuration.

## Triggers

A Freestyle project allows configuring optional triggers to control when Jenkins will start the builds. In most cases, a trigger in a Jenkins project will be a counterpart in TeamCity.

A [number of options](#) for triggering builds in TeamCity are available. It is possible to watch for changes in a source control repository, monitor an external resource, or even a Maven dependency. It is possible to schedule builds periodically, besides, the [REST API](#) is available to trigger builds externally.

## Build

This is where the real work happens. This part is mostly identical in Jenkins and TeamCity. TeamCity provides a large number of [build runners](#) out of the box. You can [configure several build steps](#) to run the required tasks for the given Build Configuration.

## Building Maven project

A Jenkins Maven project doesn't have a direct counterpart in TeamCity. Instead, an appropriate build step is selected to perform the work. In fact, [TeamCity automates](#) a lot of the setup for Maven projects and provides an additional build triggering option [based on dependencies](#).

## Post-build Actions

Post-build Actions of the Freestyle project can be mapped either to the specific attributes of a Project/Build Configuration in TeamCity or by using an additional build step. You may also find a relevant solution to the task by adding [Build features](#) to the Build Configuration in TeamCity.

An individual build step may be [configured to execute](#) depending on the success or failure of the previous build step(s) of the same Build Configuration.

## Working with branches

TeamCity provides support for the [Feature Branches](#) in distributed version control systems (DVCS).

The support for Feature Branches integrates with the various TeamCity functions. For instance, the [Branch Remote Run Trigger](#) automatically starts a new personal build each time TeamCity detects changes in particular branches of the VCS roots of the build configuration.

Here's a list of some of the highlights related to Feature Branches in TeamCity:

- The build branch name can be specified in an artifact dependency
- The VCS, Schedule, and Finish build triggers support the branch filter setting
- VCS labeling also supports the branch filter
- Mercurial bookmarks can be used in the branch specification, so, if you prefer to use bookmarks instead of standard Mercurial branches, you can fully use the TeamCity feature branches with them
- Git tags can be used in the branch specification. Some teams use Git tags the same way as branches, i.e., once a new tag is set, a build should be started in TeamCity in a branch designated by the tag name.
- [Automatic Merge](#) functionality to merge a branch into another after a successful build. The functionality is available as a

Build feature option in Build Configuration settings.

## Plugins

TeamCity comes with a lot of features built-in by default, and can be further extended by installing [additional plugins](#).

## Build pipelines

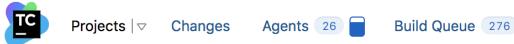
A build pipeline in TeamCity is called a **Build Chain**. It is a set of Build Configurations connected via [snapshot dependencies](#). The “[TeamCity Take on Build Pipelines](#)” article describes in detail how TeamCity handles build chains and what the implications are.

## Distributed builds

In Jenkins, to offload the master node, a permanent agent is configured to run builds.

TeamCity server doesn't run any builds itself. Instead, it always delegates the job to a [build agent](#), which means in TeamCity builds are distributed by design.

The active agents count is visible at the top of the TeamCity server UI:



You can break the agents into separate groups called [agent pools](#) and assign those to the specific projects. In Build Configuration settings, it is possible to specify a number of [Agent Requirements](#) needed for the build.

## Installing Tools

TeamCity has a number of add-ons that provide seamless integration with various IDEs and greatly extend their capabilities with features like [Personal build](#) and [Pre-tested \(delayed\) commit](#).

- [IntelliJ Platform Plugin](#)
- [Eclipse Plugin](#)
- [Visual Studio Addin](#)
- [Windows Tray Notifier](#)
- [Syndication Feed](#)

## IntelliJ Platform Plugin

TeamCity plugin provides TeamCity integration for IntelliJ Platform-based IDEs, including JetBrains IntelliJ IDEA, RubyMine, PyCharm, PhpStorm/WebStorm, AppCode, and Rider. Remote run/pre-tested commit functionality is only supported with the VCS integrations bundled with the IDEs by JetBrains.

See a [separate page](#) for the list of supported versions.



A usage example is provided in [the related blog post](#).

This section covers:

- [Features](#)
- [Installing TeamCity plugin](#)
  - [Installing the Plugin from the Plugin Repository](#)
  - [Installing the Plugin Manually](#)
- [Configuring IntelliJ IDEA-platform based IDE to Check for Plugin Updates](#)

## Features

TeamCity integration provides the following features:

- [Remote Run and Pre-Tested \(Delayed\) Commit](#),
- customizing parameters for personal builds,
- [Remote Debug](#)
- possibility to review the code duplicates,
- analyzing the results of remote code inspections,
- monitoring the status of particular projects and build configurations and the status of changes committed to the project code base,

- viewing failed tests and build logs with highlighted stacktraces and current project file names,
- start investigation of a failed build,
- assign investigation of a build configuration problem or failed test from the plugin to another team member,
- viewing build failures, which you are supposed to investigate, and giving up investigation when the problem is fixed,
- applying quick-fixes to the results of remote code analysis: the problematic code can be highlighted in the editor and you can work with a complete report of the project inspection results in a toolwindow,
- downloading and viewing only the new inspection results that appeared since the last build was created
- work with the results of server-side code duplicates search in the dedicated toolwindow,
- accessing the server-side code coverage information and visualizing the portions of code covered by unit tests,
- viewing build compilation errors in a separate tab of the build results pane with navigation to source code,
- re-running failed tests from IntelliJ IDEA plugin using JUnit or TestNG,
- opening the patch from the change details web page (for this feature to work you need to have IDEA X installed).

## Installing TeamCity plugin

TeamCity IDE plugin version should correspond to the version of the TeamCity server it connects to. Connections to TeamCity servers with different versions are generally not supported.

### Installing the Plugin from the Plugin Repository

The [plugin repository](#) has a [TeamCity plugin](#) from one of the recently released versions. You can install the plugin from repository (e.g. from IntelliJ IDEA Settings > Plugins), then enter the address of your local TeamCity server and let the plugin update itself to the version corresponding to the server.

To install the TeamCity plugin for IntelliJ platform IDE:

1. In IDE, open the Settings dialog. To do so either press Ctrl+Alt+S or choose File > Settings... (Apple > Settings... on macOS) from the main menu.
2. Open Plugins section.
3. In the Plugins section, search for 'TeamCity' or click Install JetBrains plugin... to view the list of available plugins.
4. Select the TeamCity Integration, click the Install button.
5. Restart the IDE.
6. Use the TeamCity menu to log in to your TeamCity server from the plugin.
7. Invoke the Update command in the TeamCity menu to install the plugin version matching the server version. and restart the IDE.

### Installing the Plugin Manually

The plugin for IntelliJ platform can be downloaded from the TeamCity Tools area on the My Settings & Tools page of TeamCity web UI.

To install the TeamCity plugin:

1. In the top right corner of the TeamCity web UI, click the arrow next to your username, and select My Settings & Tools.
2. On the right, locate the IntelliJ Platform Plugin section in the TeamCity Tools area, click the download link, and save the archive.
3. In IDE, open the Settings dialog. To do so either press Ctrl+Alt+S or choose File > Settings... (Apple > Settings... on macOS) from the main menu.
4. Open Plugins section.
5. In the Plugins section click Install plugin from disk....
6. In Choose Plugin File dialog select downloaded archive, click the Ok button.
7. Ensure there's enabled checkbox next to TeamCity Integration in plugins list.
8. Restart the IDE.

All additional information on how to work with the TeamCity plugin is available in IDE Help System.

## Configuring IntelliJ IDEA-platform based IDE to Check for Plugin Updates

1. In IntelliJ IDEA, open Settings/ Updates
2. Add "http://<your\_teamcity\_server\_URL>/update/idea-plugins.xml" to the list
3. Set "Check for updates" to "Daily"
4. Press "Apply", then "Check Now"

See also:

[Troubleshooting: Logging in IntelliJ IDEA/Platform-based IDEs](#)

## IntelliJ Platform Plugin Compatibility

IntelliJ IDEA version	IntelliJ Platform version	TeamCity versions								
		2017.2	2017.1*	10.0 *	9.1 *	9.0 *	8.1		8.0	7.1
2017.3	173									
2017.2	172									
2017.1	171									
2016.3	163									
2016.2.3	162					TW-46864				
2016.2	162						No Info			
2016.1	145						No Info			
15.0	143						9.0.5+	TW-41314		
14.1	141							8.1.5+		
14.0	139							8.1.5+		
13.1	135									
13.0	133									
12.1	129									
12.0	123									
11.1	117									
11.0	111									

#### Notes:

\* - Plugin from version 2017.2 could be used

[More information](#) about the IntelliJ Platform versions and IDE's (PhpStorm, RubyMine, etc) versions.

## Eclipse Plugin

### Plugin Features

TeamCity integration with Eclipse provides the following features:

- [Remote Run](#) and [Pre-Tested \(Delayed\) Commit](#) for Subversion, Perforce, CVS and Git.
- customizing parameters for personal builds,
- monitoring the projects status in the IDE,
- exploring changes introduced in the source code and comparing the local version with the latest version in the project repository,
- navigating from build logs opened in Eclipse to files referenced in the build log,
- viewing failed tests of a particular build,
- navigating to the TeamCity web interface,
- starting investigation of a build failure,
- viewing server-provided code coverage results run on TeamCity using the IDEA or EMMA code coverage engine: "<Main Menu>/TeamCity/Code Coverage Data...>,"
- comparing personal patch content with workspace resources,
- viewing compilation errors,
- downloading a patch to IDE from the TeamCity server,
- shelving changes,
- re-running tests failed on the TeamCity agent locally,
- support for P4Eclipse up to 2015.1 and Eclipse EGIT 2.0+

### Installing the Plugin

The TeamCity Eclipse plugin version must correspond to the version of the TeamCity server it connects to. Connections to

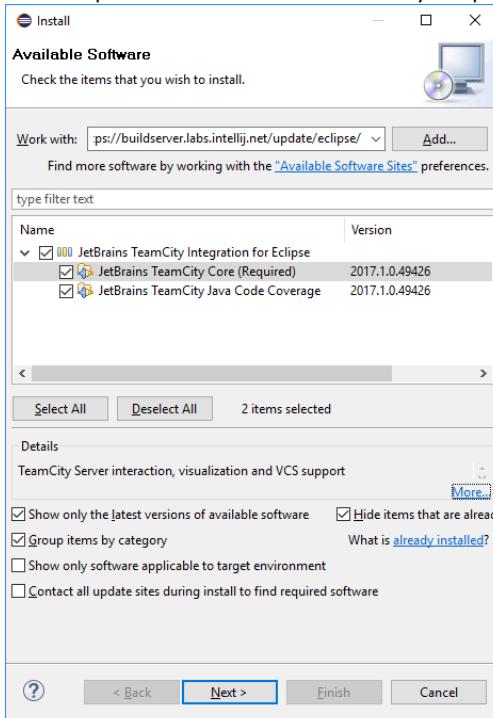
TeamCity servers with different versions are generally not supported.

#### Prerequisites

- **Subversive or Subclipse** plugins: to enable **Remote Run** and **Pre-tested Commit** for the Subversion Version Control System.  
Quick links: Subversive [download page](#). Subclipse [installation instructions](#).
- **P4Eclipse** plugin: to enable **Remote Run** and **Pre-tested Commit** for the Perforce Version Control System. Please make sure you initialize Perforce support (for example, perform project update) after opening the project before using TeamCity Remote Run.
- **CVS plugin for Eclipse** to enable **Remote Run** and **Pre-tested Commit** for CVS
- **Egit plugin for Eclipse** to support **Remote Run** and **Pre-tested Commit** for Git version control.
- **JDK 1.6-1.8** (JDK 1.8 is recommended): Eclipse must be run under JDK 1.6-1.8 for the TeamCity plugin to work.

To install the TeamCity plugin for Eclipse:

1. In the top right corner of the TeamCity web UI, click the arrow next to your username, and select **My Settings & Tools**.
2. Locate the **TeamCity Tools** section on the right.
3. Under the Eclipse plugin header, copy the update site link URL. For example, in Internet Explorer you can right-click the link and choose **Copy shortcut** from the context menu.
4. In Eclipse, click **Help | Install New Software...** on the main menu. The **Install** dialog appears.
5. Enter the URL copied above (`http://<your TeamCity Server address>/update/eclipse/`) into the URL field of the new update site in Eclipse, and click **Enter**.
6. Select the required features of the TeamCity Eclipse Plugin.



7. Click the **Next** button and follow the installation instructions.

For detailed instructions on how to work with the plugin, refer to the TeamCity section of the Eclipse help system after the plugin installation.

See also:

[Troubleshooting: Logging in TeamCity Eclipse plugin](#)

## Visual Studio Addin

The TeamCity add-in for Microsoft Visual Studio provides the following features:

- **Remote Run** for TFS, Subversion and Perforce (for remote run for Mercurial and Git see [Branch Remote Run Trigger](#)),

- Pre-Tested (Delayed) Commit for TFS, Subversion and Perforce,
- fetching JetBrains dotCover coverage analysis data from the TeamCity server (see [more](#)) to MS Visual Studio (requires dotCover of the [supported version](#) installed in Visual Studio),
- viewing recently committed changes and personal builds with their build status in the My Changes tool window,
- opening build failure details in MS Visual Studio from the TeamCity web UI,
- viewing failed tests' details for a build,
- rerunning tests failed in the TeamCity build locally via the [ReSharper](#) test runner,
- navigation from the IDE to the build results web page,
- reapplying changes sent in Remote Run or Pre-tested commit to the working directory.

For detailed instructions, refer to the [TeamCity Add-in Online Help](#).

 To enable navigation to the failed tests in MS Visual Studio by using "open in IDE" actions in the web UI, make sure that .pdb file generation for the assemblies involved in NUnit/MSTest unit tests is switched on in the current Visual Studio project.

On this page:

- [Installing Add-in](#)
- [Requirements](#)

## Installing Add-in

The TeamCity VS Add-in version must correspond to the version of the TeamCity server it connects to. Connections to TeamCity servers with different versions are generally not supported.

1. Close all running instances of Visual Studio before starting the Add-in installation (initial or upgrade).
2. Navigate to the download page of the Visual Studio Add-in:
  - a) Click the arrow next to your username in the top right corner of the TeamCity web UI and select My Settings & Tools
  - b) In the TeamCity Tools section on the right, click the Visual Studio Add-in download link.

The TeamCity Visual Studio Add-in is shipped as a part of [ReSharper Ultimate](#) products bundle. After installation, the TeamCity Add-in will be available under the RESHARPER menu in Visual Studio.

 Note that the installer will remove the pre-bundle products versions: TeamCity and ReSharper versions prior to 9.0, dotCover prior to 3.0, dotTrace prior to 6.0. ReSharper Ultimate does not support the Visual Studio versions 2005 and 2008.

The Legacy version of the TeamCity VS Add-in for Visual Studio versions from 2005 to 2013 compatible with JetBrains .NET tools prior to ReSharper 9.0, dotCover 3.0 and dotTrace 6.0 is not available since TeamCity 10.0.

## Requirements

See the [Supported Platforms and Environment](#) page for the system requirements to configure integration with different version control systems or coverage tools.

See also:

Related blog posts: [TeamCity plugin for Visual Studio@TeamCity blog](#)  
[Troubleshooting TeamCity Visual Studio Add-in issues](#)

## Windows Tray Notifier

The Windows Tray Notifier is a utility which allows monitoring the status of specific build configurations in the system tray via popup alerts and status icons.

On this page:

- [Installing Windows Tray Notifier](#)
- [Configuring Windows Tray Notifier](#)
- [Windows Tray Notifier UI](#)
  - Status Icons
  - Quick View Window
  - Pop-up Notification
- [Windows Tray Notifier Logs](#)

## Installing Windows Tray Notifier

To install the Windows Tray Notifier:

1. In the top right corner of the TeamCity web UI, click the arrow next to your username, and select My Settings & Tools.
2. In the TeamCity Tools area, click the download link under Windows tray notifier.
3. Run the TrayNotifierInstaller.msi file and follow the instructions.

## Configuring Windows Tray Notifier

To launch Windows Tray Notifier, run the Start > Programs > JetBrains TeamCity Tray Notifier menu.

When the application started, you need to connect and log in to your server:

1. Specify your TeamCity server URL and credentials to log in to it.
2. Wait for the notifier to connect to your TeamCity server.

Once the connection is established, you'll be prompted to configure notification rules in the My Settings & Tools | Notification rules | Windows Tray Notifier section:

The screenshot shows the 'My Settings & Tools' page in the TeamCity web interface. The 'Notification Rules' section is selected. A yellow banner at the top right of the table says 'Notification rule added.' The table has two columns: 'Watching' and 'Send notification when'. Under 'Watching', it lists 'Events in' and 'and related to the following projects and build configurations'. The configuration details include '<Root project>', 'Documentation', 'IntelliJ Platform Products' (with 'TeamCity Plugin for IntelliJ Platform' checked), 'dotNet Products', and 'TeamCity Add-In'. Under 'Send notification when', it lists '- Build fails', '- Build is successful', and '- Investigation is updated'. There are 'Edit' and 'Delete' links for each row. At the bottom of the table, there's a note: 'To unsubscribe from group notifications, add your own rule with the same watched builds and different notification events. To unsubscribe from all events, add a rule with the corresponding watched builds and no events selected.' A green 'Add new rule' button is at the bottom left.

3. When Windows Tray Notifier is launched, the status icon in the Windows System Tray appears.

## Windows Tray Notifier UI

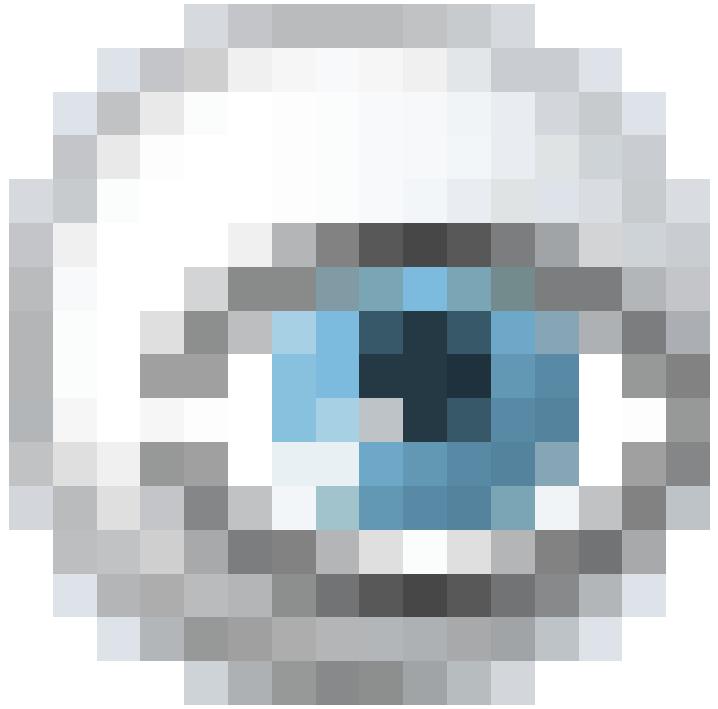
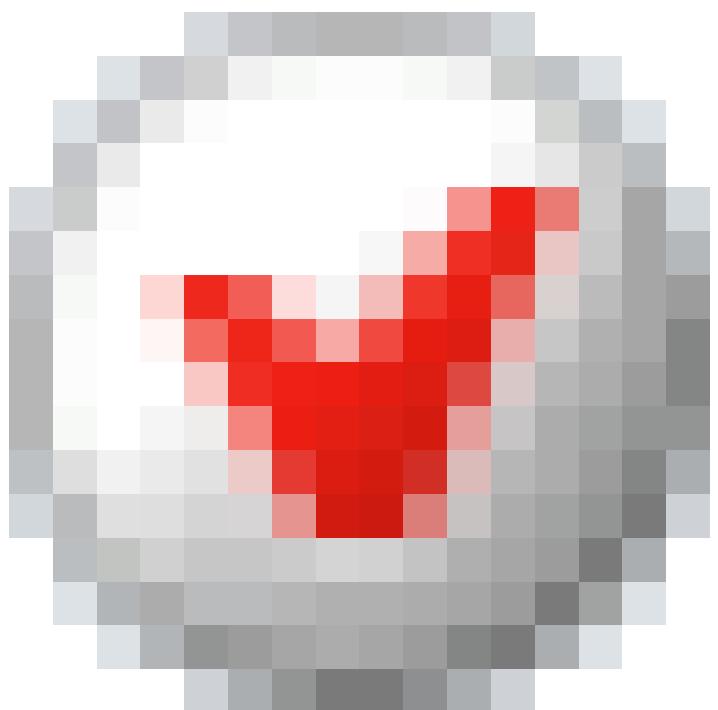
- **Tray Status Icons**
- **Quick View Window**
- **Pop-up Notification**

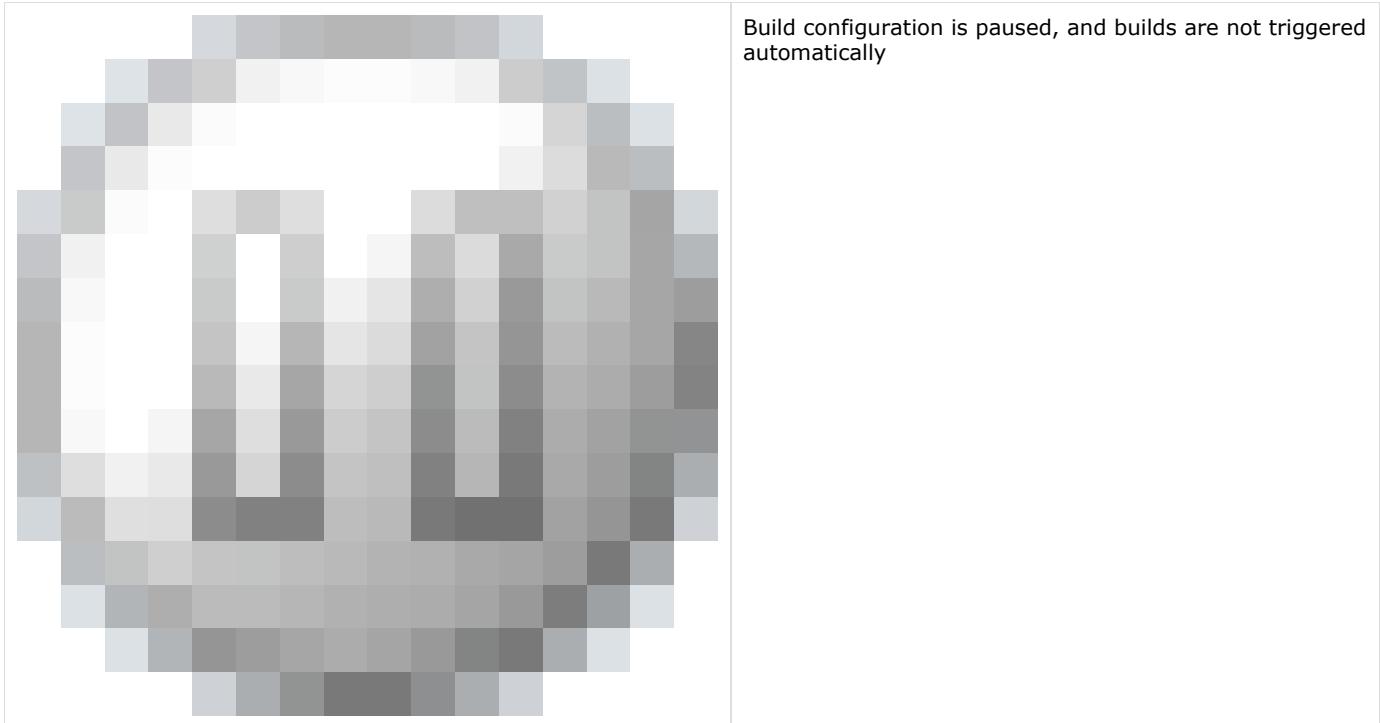
### Status Icons

After you have launched Windows Tray Notifier and specified your TeamCity username and password, the Notifier icon showing the state of your projects and build configurations appears in Windows System Tray.

If you have no projects and build configurations to monitor, the icon represents a question mark. After you have configured a list of build configurations and projects and their state changes, the status icon changes to reflect the change as well. The table below represents these possible states.

Icon	Meaning
	Build is successful
	Build failed and nobody is investigating the failure

	A team member has started investigating the build failure
	The person who investigated the build failure has submitted a fix, but the build has not executed successfully yet



Build configuration is paused, and builds are not triggered automatically



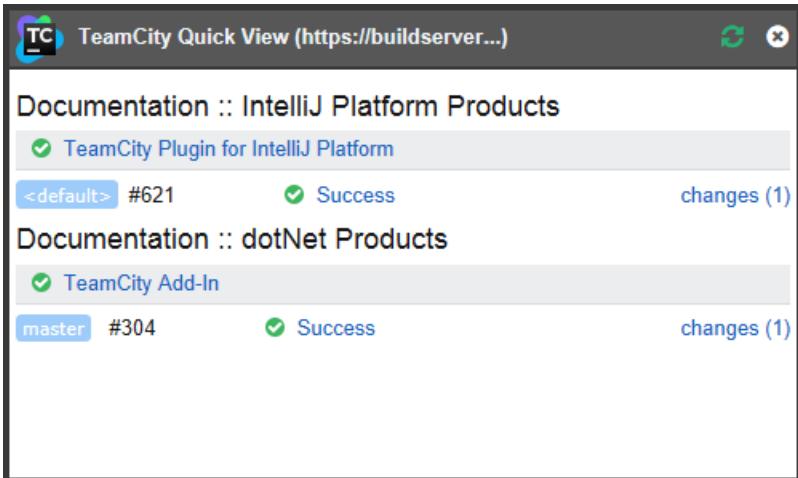
The Notifier icon always shows the status of the last completed build of your watched project or build configurations, unless you select to be notified when the first build error occurs option on the Windows Tray Notifier settings page. In this case, the notifier does not wait for the failing build to finish, and it displays a failed build icon as soon as the running build fails a test.

If you right-click the status icon, you can access all Windows Tray Notifier features described in table below.

Option	Description
Open Quick View Window	Displays the Quick View window.
Go to "Projects" Page...	Opens the Projects tab.
Go to "My Changes" Page...	Opens the My Changes tab.
Configure Watched Builds...	Opens the Windows Tray Notifier settings page where you can select the build configurations to monitor and notification events.
Auto Upgrade	Select this option to allow the program to automatically upgrade.
Run on Startup	Select this option to automatically launch the program when windows boots.
About	Displays the information on the program's splash screen.
Logout	Use this function to log out of the TeamCity server. This will allow you to a different one.
Exit	Quits the program.

## Quick View Window

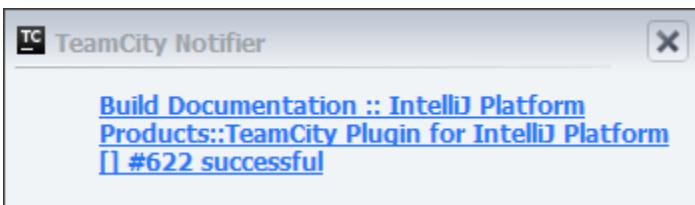
On left-clicking the Notifier icon, the TeamCity Quick View window appears displaying the status of the watched projects / build configurations:



Click the build results (Success in the image above) or changes links to navigate to the Build Results and Changes pages respectively in the TeamCity web interface for details.

#### Pop-up Notification

Besides the state icons, Windows tray notifier displays pop-up alerts with a brief build results information on the particular build configurations and notification events.



When a pop-up notification appears, you can click the link in it to go the Build results page for details.

#### Windows Tray Notifier Logs

Since TeamCity 2017.2 Windows Tray Notifier logs events and warnings to the `%ProgramData%` or `%AppData%` (since 2017.2.1) `JetBrains\TeamCity\TrayNotifier\logs` directory containing the following files:

- `teamcity-tray.log` with common details
- `teamcity-update.log` with update details.

You can tune the logger verbosity via the `/verbosity` command line switch: debug logs can be enabled using the following command:

```
C:\Program Files (x86)\JetBrains\TeamCity\TrayNotifier\JetBrains.TrayNotifier.exe /verbosity:debug
```

See also:

[User's Guide: Subscribing to Notifications](#)  
[Administrator's Guide: Customizing Notifications](#)  
[Installing Tools: Working with Windows Tray Notifier](#)

# Syndication Feed

To configure a syndication feed for obtaining information about the builds of certain build configurations:

1. In the top right corner of the TeamCity web UI, click the arrow next to your username, and select My Settings & Tools.
2. In the TeamCity Tools section, in the Syndication Feed section click the Customize link.
3. In the [Feed URL Generator page](#) specify the build configurations and events (builds and/or changes) you want to be notified about, and define authentication settings.
4. Copy the Feed URL, generated by TeamCity, to your feed reader, or click Subscribe.
5. Preview summary of the subscription and click Subscribe now.

See also:

[User's Guide: Feed URL Generator](#)

# Extending TeamCity

TeamCity behavior can be extended in several ways. You can communicate with TeamCity from the build script and report tests, change build number or provide statistics data. Or you can write full-fledged plugin which will provide custom UI, customized notifications and much more.

If you cannot find relevant information here, have questions or want to share your TeamCity plugins experience with other users, welcome to [TeamCity Plugins Forum](#).

## Customizing TeamCity without Plugins

- [Build Script Interaction with TeamCity](#)
- [REST API and Accessing Server by HTTP](#)
- [Including Third-Party Reports in the Build Results](#)

## Plugin Development

- [Getting Started with Plugin Development](#)
- [Typical Plugins](#)
  - [Build Runner Plugin](#)
  - [Risk Tests Reordering in Custom Test Runner](#)
  - [Custom Build Trigger](#)
  - [Extending Notification Templates Model](#)
  - [Issue Tracker Integration Plugin](#)
  - [Version Control System Plugin](#)
  - [Version Control System Plugin \(old style - prior to 4.5\)](#)
  - [Custom Authentication Module](#)
  - [Custom Notifier](#)
  - [Custom Statistics](#)
  - [Custom Server Health Report](#)
  - [Extending Highlighting for Web diff view](#)
  - [External Storage Implementation Guide](#)
- [Bundled Development Package](#)
- [Open API Changes](#)
- [Plugin Types in TeamCity](#)
- [Plugins Packaging](#)
- [Server-side Object Model](#)
- [Agent-side Object Model](#)
- [Extensions](#)
- [Web UI Extensions](#)
- [Plugin Settings](#)
- [Development Environment](#)
- [Developing Plugins Using Maven](#)
- [Plugin Development FAQ](#)

[Open API Javadoc](#)

[Open API Javadoc \(ver. 10.0.x\)](#)

## Publicly Available Plugins

- TeamCity Plugins
- Open-source Bundled Plugins

## Build Script Interaction with TeamCity

If TeamCity doesn't support your testing framework or build runner out of the box, you can still avail yourself of many TeamCity benefits by customizing your build scripts to interact with the TeamCity server. This makes a wide range of features available to any team regardless of their testing frameworks and runners. Some of these features include displaying real-time test results and customized statistics, changing the build status, and publishing artifacts before the build is finished.

The build script interaction can be implemented by means of:

- service messages in the build script
- `teamcity-info.xml` file (obsolete approach, consider using service messages instead)



If you use MSBuild build runner, you can use [MSBuild Service Tasks](#).

In this section:

- Service Messages
  - Service messages formats
  - Escaped values
  - Common Properties
    - Message Creation Timestamp
    - Message FlowId
  - Reporting Messages For Build Log
    - Blocks of Service Messages
    - Reporting Compilation Messages
    - Reporting Tests
      - Supported test service messages
      - Nested test reporting
      - Interpreting test names
  - Reporting .NET Code Coverage Results
  - Reporting Inspections
    - Inspection type
    - Inspection instance
    - Example
  - Publishing Artifacts while the Build is Still in Progress
  - Reporting Build Progress
  - Reporting Build Problems
  - Reporting Build Status
  - Reporting Build Number
  - Adding or Changing a Build Parameter
  - Reporting Build Statistics
  - Disabling Service Messages Processing
  - Importing XML Reports
- Libraries reporting results via TeamCity Service Messages
- `teamcity-info.xml`
  - Modifying the Build Status
  - Reporting Custom Statistics
    - Providing data using the `teamcity-info.xml` file
    - Describing custom charts

## Service Messages

Service messages are specially constructed pieces of text that are used to pass commands/information about the build from the build script to the TeamCity server.

To be processed by TeamCity, they need to be written to the standard output stream of the build, i.e. printed or echoed from a build step. It is recommended to output a single service message per line (i.e. divide messages with newline symbol(s))

Examples:

Windows

```
echo ##teamcity[<messageName> 'value']
```

Linux

```
echo "##teamcity[<messageName> 'value']"
```

PowerShell script

```
Write-Host "##teamcity[<messageName> 'value']"
```

A single service message cannot contain a newline character inside it, it cannot span across multiple lines.

## Service messages formats

Service messages support two formats:

- Single-attribute message:

```
##teamcity[<messageName> 'value']
```

- Multiple-attribute message:

```
##teamcity[<messageName> name1='value1' name2='value2']
```

Multiple attributes message can more formally be described as:

```
##teamcity[messageNameWSPpropertyNameOWSP=OWSP'value'WSPpropertyName_IDOWSP=OWSP'value'...OWSP]
```

where:

- **messageName** is a name of the message. See below for supported messages. The message name should be a valid Java id (only alpha-numeric characters and "-", starting with an alpha character)
- **propertyName** is a name of the message attribute. Should be a valid Java id.
- **value** is a value of the attribute. Should be an escaped value (see below).
- WSP is a required whitespace(s): space or tab character (t)
- OWSP is an optional whitespace(s)
- ... is any number of WSP**propertyNameOWSP=OWSP'\_value'\_** blocks

## Escaped values

For escaped values, TeamCity uses a vertical bar "|" as an escape character. In order to have certain characters properly interpreted by the TeamCity server, they must be preceded by a vertical bar. For example, the following message:

```
##teamcity[testStarted name='foo|'s test']
```

will be displayed in TeamCity as 'foo's test'. Please, refer to the table of the escaped values below.

Character	Escape as
' (apostrophe)	'
\n (line feed)	n
\r (carriage return)	r

\uNNNN (unicode symbol with code 0xNNNN)	0xNNNN
(vertical bar)	
[ (opening bracket)	[
] (closing bracket)	]

## Common Properties

Any "message and multiple attribute" message supports the following list of optional attributes: `timestamp`, `flowId`. In the following examples `<messageName>` is the name of the specific service message.

### Message Creation Timestamp

```
# #teamcity[<messageName> timestamp='timestamp' ...]
```

The timestamp format is `yyyy-MM-dd'T'HH:mm:ss.SSSZ` or `yyyy-MM-dd'T'HH:mm:ss.SSS` according to Java [SimpleDateFormat syntax](#), e.g.

```
# #teamcity[<messageName> timestamp='2008-09-03T14:02:34.487+0400' ...]
# #teamcity[<messageName> timestamp='2008-09-03T14:02:34.487' ...]
```

For .NET [DateTimeOffset](#), a code like this

```
var date = DateTimeOffset.Now;
var timestamp = $"{date:yyyy-MM-dd'T'HH:mm:ss.fff}{date.Offset.Ticks:+;-}{date.Offset:hhmm}";
```

will result in

```
# #teamcity[<messageName> timestamp='2008-09-03T14:02:34.487+0400' ...]
```

## Message FlowId

The `flowId` is a unique identifier of the messages flow in a build. Flow tracking is necessary, for example, to distinguish separate processes running in parallel. The identifier is a string that should be unique in the scope of individual build.

```
# #teamcity[<messageName> flowId='flowId' ...]
```

## Reporting Messages For Build Log

You can report messages for a build log in the following way:

```
# #teamcity[message text='<message text>' errorDetails='<error details>' status='<status value>']
```

where:

- The `status` attribute may take following values: `NORMAL`, `WARNING`, `FAILURE`, `ERROR`. The default value is `NORMAL`.
- The `errorDetails` attribute is used only if `status` is `ERROR`, in other cases it is ignored.

This message fails the build in case its status is `ERROR` and "Fail build if an error message is logged by build runner" box is checked on the [Build Failure Conditions](#) page of a build configuration. For example:

```
##teamcity[message text='Exception text' errorDetails='stack trace' status='ERROR']
```

## Blocks of Service Messages

Blocks are used to group several messages in the build log.

Block opening:

The `blockOpened` system message allows the `name` attribute, you can also add a description to the to the `blockOpened` message:

```
##teamcity[blockOpened name='<blockName>' description='<this is the description of blockName>']
```

Block closing:

```
##teamcity[blockClosed name='<blockName>']
```



Please note that when you close the block, all inner blocks are closed automatically.

## Reporting Compilation Messages

```
##teamcity[compilationStarted compiler='<compiler name>']
...
##teamcity[message text='compiler output']
##teamcity[message text='compiler output']
##teamcity[message text='compiler error' status='ERROR']
...
##teamcity[compilationFinished compiler='<compiler name>']
```

where:

- `compiler name` is an arbitrary name of the compiler performing compilation, eg. `javac`, `groovyc` and so on. Currently it is used as a block name in the build log.
- any message with status `ERROR` reported between `compilationStarted` and `compilationFinished` will be treated as a compilation error.

## Reporting Tests

To use the TeamCity on-the-fly test reporting, a testing framework needs dedicated support for this feature to work (alternatively, [XML Report Processing](#) can be used).

If TeamCity doesn't support your testing framework natively, it is possible to modify your build script to report test runs to the TeamCity server using service messages. This makes it possible to display test results in real-time, make test information available on the [Tests tab of the Build Results page](#).

## Supported test service messages

Test suite messages:

Test suites are used to group tests. TeamCity displays tests grouped by suites on [Tests tab of the Build Results page](#) and in other places.

```
##teamcity[testSuiteStarted name='suiteName']
<individual test messages go here>
##teamcity[testSuiteFinished name='suiteName']
```

All the individual test messages are to appear between `testSuiteStarted` and `testSuiteFinished` (in that order) with the same `name` attributes.

#### Nested test reporting

Prior to TeamCity 9.1, one test could have been reported [from within another test](#).

In the later versions, starting another test finishes the currently started test in the same "flow". To still report tests from within other tests, you will need to specify another `flowId` in the nested test service messages.

#### Test start/stop messages:

```
##teamcity[testStarted name='testName' captureStandardOutput='<true/false>']
<here go all the test service messages with the same name>
##teamcity[testFinished name='testName' duration='<test_duration_in_milliseconds>']
```

Indicates that the test "testName" was run. If the `testFailed` message is not present, the test is regarded successful, where

- duration (optional numeric attribute) - sets the test duration in milliseconds (should be an integer) to be reported in TeamCity UI. If omitted, the test duration will be calculated from the messages timestamps. If the timestamps are missing, from the actual time the messages were received on the server.
- captureStandardOutput (optional boolean attribute) - if `true`, all the standard output (and standard error) messages received between `testStarted` and `testFinished` messages will be considered test output. The default value is `false` and assumes usage of `testStdOut` and `testStdErr` service messages to report the test output.



All the other test messages (except for `testIgnored`) with the same `name` attribute should appear between the `testStarted` and `testFinished` messages (in that order).

If using Ant's echo task to output the messages, make sure to include `flowId` attribute with the same value in all the messages related to the same test/test suite as otherwise they [will not be processed correctly](#).

It is highly recommended to ensure that the pair of `test suite + test name` is unique within the build.

For advanced TeamCity test-related features to work, test names should not deviate from one build to another (a single test must be reported under the same name in every build). Include absolute paths in the reported test names is strongly discouraged.

#### Ignored tests:

```
##teamcity[testIgnored name='testName' message='ignore comment']
```

Indicates that the test "testName" is present but was not run (was ignored) by the testing framework.

As an exception, the `testIgnored` message can be reported without the matching `testStarted` and `testFinished` messages.

#### Test output:

```
##teamcity[testStarted name='className.testName']
##teamcity[testStdOut name='className.testName' out='text']
##teamcity[testStdErr name='className.testName' out='error text']
##teamcity[testFinished name='className.testName' duration='50']
```

The `testStdOut` and `testStdErr` service messages report the test's standard and error output to be displayed in the TeamCity UI. There must be only one `testStdOut` and one `testStdErr` message per test.

An alternative but a less reliable approach is to use the `captureStandardOutput` attribute of the `testStarted` message.

Test result:

```
##teamcity[testStarted name='MyTest.test1']
##teamcity[testFailed name='MyTest.test1' message='failure message' details='message and stack trace']
##teamcity[testFinished name='MyTest.test1']

##teamcity[testStarted name='MyTest.test2']
##teamcity[testFailed type='comparisonFailure' name='MyTest.test2' message='failure message' details='message and stack trace' expected='expected value' actual='actual value']
##teamcity[testFinished name='MyTest.test2']
```

Indicates that the "testname" test failed. Only one testFailed message can appear for a given test name.

- message contains the textual representation of the error
- details contains detailed information on the test failure, typically a message and an exception stacktrace.
- actual and expected attributes can only be used together with type='comparisonFailure' to report comparison failure. The values will be used when opening the test in the IDE.

Here is a longer example of test reporting with service messages:

```
##teamcity[testSuiteStarted name='suiteName']
##teamcity[testSuiteStarted name='nestedSuiteName']
##teamcity[testStarted name='package_or_namespace.ClassName.TestName']
##teamcity[testFailed name='package_or_namespace.ClassName.TestName' message='The number should be 20000' details='junit.framework.AssertionFailedError: expected:<20000> but was:<10000>|n|r at junit.framework.Assert.fail(Assert.java:47)|n|r at junit.framework.Assert.failNotEquals(Assert.java:280)|n|r...']
##teamcity[testFinished name='package_or_namespace.ClassName.TestName']
##teamcity[testSuiteFinished name='nestedSuiteName']
##teamcity[testSuiteFinished name='suiteName']
```

## Interpreting test names

A full test name can have a form of: <suite name>:<package/namespace name>.<class name>.<test name>,

where <class name> and <test name> can have no dots in the names.

The [Tests tab of the Build Results page](#) allows grouping by suites, packages/namespaces, classes, and tests. Usually the attribute values are provided as they are reported by your test framework and TeamCity is able to interpret which part of the reported names is the test name, class, package as follows:

TeamCity takes the suite name from the corresponding suite message and

- if the reported test name starts with the suite name, the suite name is truncated from the test name before further processing
- the part of the test name after the last dot is treated as a test name
- the part of the test name before the last dot is treated as a class name
- the rest of the test name is treated as a package/namespace name

## Reporting .NET Code Coverage Results

You can configure .NET coverage processing by means of service messages. To learn more, refer to [Manually Configuring Reporting Coverage page](#).

## Reporting Inspections

You can report inspections from a custom tool to TeamCity using the service messages described below.

Among other uses, the number of inspections can be used as a build metric to [fail a build on](#).

## Inspection type

Each specific warning or an error in code (inspection instance) has an inspection type - the unique description of the conducted inspection, which can be reported via

```
##teamcity[inspectionType id='<id>' name='<name>' description='<description>'  
category='<category>']
```

where all the attributes are required and can have either numeric or textual values

`id` - (mandatory) limited by 255 characters

`name` - (mandatory) limited by 255 characters

`category` - (mandatory) limited by 255 characters. The `category` attribute examples are "Style violations", "Calling contracts" etc.

`description` (mandatory) limited by 4000 characters. The description can also be in html, e.g.

```
<html>  
<body>  
Reports unnecessary local variables, which add nothing to the comprehensibility of a method. Variables  
caught include local variables which are immediately returned, local variables that are immediately  
assigned to another variable and then not used, and local variables which always have the same value  
as another  
local variable or parameter.  
<!-- tooltip end -->  
<p>  
Use the first checkbox below to have this inspection ignore variables which are immediately returned  
or thrown. Some coding styles suggest using such variables for clarity and ease of debugging.  
<p>  
Use the second checkbox below to have this inspection ignore variables which are annotated.  
<p>  
</body>  
</html>
```

## Inspection instance

Reports a specific defect, warning, error message. Includes location, description, and various optional and custom attributes.

```
##teamcity[inspection typeId='<inspection type identity>' message='<instance description>'  
file='<file path>' line='<line>' additional attribute='<additional attribute>']
```

where all the attributes can have either numeric or textual values:

`typeId` - (mandatory), reference to the `inspectionType.id` described [above](#) limited by 255 characters

`message` - (optional) current instance description limited by 4000 characters

`file` - (mandatory) file path limited by 4000 characters. The path can be absolute or relative to the `checkout directory`

`line` - (optional) line of the file, integer

`additional attribute` - can be any attribute, `SEVERITY` is often used here, with one of the following values (⚠ mind the upper case): `INFO`, `ERROR`, `WARNING`, `WEAK WARNING`

## Example

```
##teamcity[inspectionType id='UnnecessaryLocalVariable' name='Redundant local variable'  
description='<html><body>Reports unnecessary local variables...</body>  
</html>' category='Data flow issues']  
##teamcity[inspection typeId='UnnecessaryLocalVariable' message='Local variable <code>i</code>  
is redundant' file='src/Test.java' line='19' SEVERITY='WARNING']
```

## Publishing Artifacts while the Build is Still in Progress

You can publish the build artifacts while the build is still running, immediately after the artifacts are built.

To do this, you need to output the following line:

```
# #teamcity[publishArtifacts '<path>']
```

The `<path>` has to adhere to the same rules as the [Build Artifact specification](#) of the Build Configuration settings. The files matching the `<path>` will be uploaded and visible as the artifacts of the running build.

The message should be printed after all the files are ready and no file is locked for reading.

Artifacts are uploaded in the background, which can take time. Please make sure the matching files are not deleted till the end of the build (e.g. you can put them in a directory that is cleaned on the next build start, in a [temp directory](#) or use [Swabra](#) to clean them after the build.)

 The process of publishing artifacts process can affect the build because it consumes network traffic and some disk/CPU resources (should be pretty negligible for not large files/directories).

Artifacts that are specified in the build configuration setting will be published as usual.

## Reporting Build Progress

You can use special progress messages to mark long-running parts in a build script. These messages will be shown on the projects dashboard for the corresponding build and on the [Build Results page](#).

To log a single progress message, use:

```
# #teamcity[progressMessage '<message>']
```

This progress message will be shown until another progress message occurs or until the next target starts (in case of Ant builds).

If you wish to show a progress message for a part of a build only, use:

```
# #teamcity[progressStart '<message>']
...some build activity...
# #teamcity[progressFinish '<message>']
```

 The same message should be used for both `progressStart` and `progressFinish`. This allows nesting progress blocks. Also note that in case of Ant builds, progress messages will be replaced if an Ant target starts.

## Reporting Build Problems

To fail a build directly from the build script, a build problem has been reported. Build problems appear on the [Build Results page](#) and also affect the build status text.

To add a build problem to a build, use:

```
# #teamcity[buildProblem description='<description>' identity='<identity>']
```

where:

- `description` - (mandatory) a human-readable plain text describing the build problem. By default, the `description` appears in the build status text and in the list of build's problems. The text is limited to 4000 symbols, and

- will be truncated if the limit is exceeded.
- identity - (optional) a unique problem id. Different problems must have different identity, same problems - same identity, which should not change throughout builds if the same problem occurs, e.g. the same compilation error. It should be a valid Java id up to 60 characters. If omitted, the identity is calculated based on the description text.

## Reporting Build Status

TeamCity allows changing the build status text from the build script. Unlike [progress messages](#), this change persists even after a build has finished.

You can also change the build status of a failing build to success.

Prior to TeamCity 7.1, this service message could be used for changing the build status to failed. In the later TeamCity versions, the [buildProblem](#) service message is to be used for that.

To set the status and/or change the text of the build status (for example, note the number of failed tests if the test framework is not supported by TeamCity), use the [buildStatus](#) message with the following format:

```
# #teamcity[buildStatus status='<status value>' text='{build.status.text} and some aftertext']
```

where:

- status attribute is optional and may take the value `SUCCESS`.
- text attribute sets the new build status text. Optionally, the text can use `{build.status.text}` substitution pattern which represents the status, calculated by TeamCity automatically using passed test count, compilation messages and so on.

The status set will be presented while the build is running and will also affect the final build results.

## Reporting Build Number

To set a custom build number directly, specify a [buildNumber](#) message using the following format:

```
# #teamcity[buildNumber '<new build number>']
```

In the `<new build number>` value, you can use the `{build.number}` substitution to use the current build number automatically generated by TeamCity. For example:

```
# #teamcity[buildNumber '1.2.3_{build.number}-ent']
```

## Adding or Changing a Build Parameter

By using a dedicated service message in your build script, you can dynamically update build parameters of the build right from a build step (the parameters need to be defined in the [Parameters section](#) of the build configuration).

The changed build parameters will be available in the build steps following the modifying one. They will also be available as build parameters and can be used in the dependent builds via `%dep.*%` parameter references, e.g.

```
# #teamcity[setParameter name='ddd' value='fff']
```

When specifying a build parameter's name, mind the prefix:

- system for system properties.
- env for environment variables.
- no prefix for configuration parameter.

Read more about build parameters and their prefixes.

## Reporting Build Statistics

In TeamCity, it is possible to configure a build script to report statistical data and then display the charts based on the data. Please refer to the [Customizing Statistics Charts#customCharts](#) page for a guide to displaying the charts on the web UI. This section describes how to report the statistical data from the build script via service messages. You can publish the build statics values in two ways:

- Using a service message in a build script directly
- Providing data using the [teamcity-info.xml](#) file

To report build statistics using service messages: Specify a 'buildStatisticValue' service message with the following format for each statistics value you want to report:

```
# #teamcity[buildStatisticValue key='<valueTypeKey>' value='<value>']
```

where

- The key should not be equal to any of [predefined keys](#).
- The value should be a positive/negative integer of up to 13 digits; float values with up to 6 decimal places are also supported.

## Disabling Service Messages Processing

If you need for some reason to disable searching for service messages in the output, you can disable the service messages search with the messages:

```
# #teamcity[enableServiceMessages]  
# #teamcity[disableServiceMessages]
```

Any messages that appear between these two are not parsed as service messages and are effectively ignored. For server-side processing of service messages, enable/disable service messages also supports the flowId attribute and will ignore only the messages with the same flowId.

## Importing XML Reports

In addition to the [UI Build Feature](#), XML reporting can be configured from within the build script with the help of the `importData` a service message.

Also, the message supports importing of the previously collected code coverage and code inspection/duplicates reports.

The service message format is:

```
# #teamcity[importData type='typeID' path='<path to the xml file>']
```



To be processed, report XML files (or a directory) must be located in the checkout directory, and the path must be relative to [this directory](#).

where `typeID` can be one of the following (see also [XML Report Processing](#)):

<code>typeID</code>	Description
Testing frameworks	
junit	JUnit Ant task XML reports
surefire	Maven Surefire XML reports

nunit	NUnit-Console XML reports
mstest	MSTest XML reports
vstest	VSTest XML reports
gtest	Google Test XML reports
<b>Code inspection</b>	
intellij-inspections	Since TeamCity 2017.1 IntelliJ IDEA inspection results
checkstyle	Checkstyle inspections XML reports
findBugs <sup>2)</sup>	FindBugs inspections XML reports
jslint	JSLint XML reports
ReSharperInspectCode <sup>1)</sup>	ReSharper inspectCode.exe XML reports
FxCop <sup>1)</sup>	FxCop inspection XML reports
pmd	PMD inspections XML reports
<b>Code duplication</b>	
pmdCpd	PMD Copy/Paste Detector (CPD) XML reports
DotNetDupFinder <sup>1)</sup>	ReSharper dupfinder.exe XML reports
<b>Code coverage</b>	
dotNetCoverage <sup>1) 3)</sup>	XML reports generated by dotcover, partcover, ncover or ncover3

Notes:

<sup>1)</sup> only supports specific file in the path attribute

<sup>2)</sup> also requires the findBugsHome attribute specified pointing to the home directory of the installed FindBugs tool.

<sup>3)</sup> also requires the tool='<tool name>' service message attribute, where the <tool name> is either dotcover, partcover, nc over or ncover3.

If not specially noted, the report types support Ant-like wildcards in the path attribute.

the verbose='true' attribute will enable detailed logging into the build log.

the parseOutOfDate='true' attribute will process all the files matching the path. By default, only those updated during the build (determined by last modification timestamp) are processed. If any not matching reports are found, the "report skipped as out-of-date" message appears in the build log.

the whenNoDataPublished=<action> (where <action> is one of the following: info (default), nothing, warning, error) will change output level if no reports matching the path specified were found.

(deprecated, use [Build Failure Conditions](#) instead)

findBugs, pmd or checkstyle importData messages also take optional errorLimit and warningLimit attributes specifying errors and warnings limits, exceeding which will cause the build failure.

 After the importData message is received, TeamCity agent starts to monitor the specified paths on the disk and imports matching report files in the background as soon as the files appear on disk.

The parsing only occurs within the build step in which the messages were received. On the step finish, the agent ensures all the present reports are processed before beginning the next step. This behavior is different from that of [ML Report Processing](#) build feature, which completes files parsing only at the end of the build.

Please ensure the report files are available after the generation process ends (the files are not deleted, nor overwritten by the build script)

To initiate monitoring of several directories or parse several types of the report, send the corresponding service messages one after another.

 Only several reports of different types can be included in a build. Processing reports of several inspections or

duplicates tools in a single build is not supported. See the [related feature request](#).

## Libraries reporting results via TeamCity Service Messages

Several platform-specific libraries from JetBrains and external sources are able to report the results via TeamCity Service messages.

- **Service messages .NET library** - .NET library for generating (and parsing) TeamCity service messages from .NET applications. See a [related blog post](#).
- **Jasmine 2.0 TeamCity reporter** - support for emitting TeamCity service messages from Jasmine 2.0 reporter
- **Perl TAP Formatter** - formatter for Perl to transform TAP messages to TeamCity service messages
- **PHPUnit 5.0** - supports TeamCity service messages for tests. For earlier PHPUnit versions, the following external libraries can be used: **PHPUnit Listener 1**, **PHPUnit Listener 2** - listeners which can be plugged via PHPUnit's `suite.xml` to produce TeamCity service messages for tests.
- **Python Unit Test Reporting to TeamCity** - the package that automatically reports unit tests to the TeamCity server via service messages (when run under TeamCity and provided the testing code is adapted to use it).
- **Mocha** - on-the-fly reporting via service messages for Mocha JavaScript testing framework. See the related [post](#) with instructions.
- **Karma** - support in the JavaScript testing tool to report tests progress into TeamCity using TeamCity service messages

## teamcity-info.xml

As an obsolete approach, it is also possible to have the build script collect information, generate an XML file called `teamcity-info.xml` in the root build directory. When the build finishes, this file will automatically be uploaded as a build artifact and processed by the TeamCity server.

Please note that this approach can be discontinued in the future TeamCity versions, so service messages approach is recommended instead. In case service messages does not work for you, please let us know the details and describe the case via [email](#).

## Modifying the Build Status

TeamCity has the ability to change the build status directly from the build script. You can set the status (build failure or success) and change the text of the build status (for example, note the number of failed tests if the test framework is not supported by TeamCity).

### XML schema for teamcity-info.xml

It is possible to set the following information for the build:

Build number — Sets the new number for the finished build. You can reference the TeamCity-provided build number using `{build.number}`.

Build status — Change the build status. Supported values are "FAILURE" and "SUCCESS".

Status text — Modify the text of build status. You can replace the TeamCity-provided status text or add a custom part before or after the standard text. Supported `action` values are "append", "prepend" and "replace".

Example `teamcity-info.xml` file:

```
<build number="1.0.{build.number}">
  <statusInfo status="FAILURE"> <!-- or SUCCESS -->
    <text action="append"> fitness: 45</text>
    <text action="append"> coverage: 54%</text>
  </statusInfo>
</build>
```



It is up to you to figure out how to retrieve test results that are not supported by TeamCity and accurately add them to the `teamcity-info.xml` file.

## Reporting Custom Statistics

It is possible to provide [custom charts](#) in TeamCity. Your build can provide data for such graphs using `teamcity-info.xml` file.

### Providing data using the `teamcity-info.xml` file

This file should be created by the build in the root directory of the build. You can publish multiple statistics (see the details on the data format below) and create separate charts for each set of values.

The `teamcity-info.xml` file is to contain the code in the following format (you can combine various data in the `teamcity-info.xml` file):

```
<build>
  <statisticValue key="chart1Key" value="342"/>
  <statisticValue key="chart2Key" value="53"/>
</build>
```

The key should not be equal to any of [predefined keys](#).

The value should be a positive/negative integer of up to 13 digits. Float values with up to 6 decimal places are also supported.

The key here relates to the key of the `valueType` tag used when describing the chart.

### Describing custom charts

See [Customizing Statistics Charts](#) page for detailed description.

## Accessing Server by HTTP

In addition to the commands described here, there is a [REST API](#) that you can use for certain operations. When available, using REST API is a preferred way over one described here.

The examples below assume that your server web UI is accessible via <http://teamcity.jetbrains.com:8111/> URL.

The TeamCity server supports basic HTTP authentication allowing users to access certain web server pages and perform actions from various scripts. Please consult the manual for the client tool/library on how to supply basic HTTP credentials when issuing a request.

Use the valid TeamCity server username and password to use basic HTTP authentication. Appropriate user permissions are required to perform the actions.

 You may want to [configure](#) the server to use HTTPS as username and password are passed in insecure form during basic HTTP authentication.

To force using a basic HTTP authentication instead of redirecting to the login page if no credentials are supplied, prepend a path in the usual TeamCity URL with `"/httpAuth"`. For example:

```
http://teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf
```

The HTTP authentication can be useful when [downloading build artifacts](#) and triggering a build.

If you have Guest user enabled, it can be used to perform the action too. Use `"/guestAuth"` before the URL path to perform the action on Guest user behalf. For example:

```
http://teamcity.jetbrains.com:8111/guestAuth/action.html?add2Queue=MyBuildConf
```

 Please make sure the user used to perform the authentication (or Guest user) has appropriate role to perform the necessary operation.

### Triggering a Build From Script

Since TeamCity 8.1 the recommended and more feature-rich way to trigger a build is via [REST API](#). The approach below will be removed in the future TeamCity versions.

To trigger a build, send the HTTP POST request for the URL: `http://<server address>/httpAuth/action.html?add2Queue=<build configuration ID>` performing basic HTTP authentication.

Some tools (for example, [Wget](#)) support the following syntax for the basic HTTP authentication:

```
http://<user name>:<user password>@<server address>/httpAuth/action.html?add2Queue=<build configuration Id>
```

Example:

```
http://testuser:testpassword@teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf
```

You can trigger a build on a specific agent passing additional `agentId` parameter with the agent's Id. You can get the agent Id from the URL of the Agent's details page (Agents page > <agent name>). For example, you can infer that agent's Id equals "2", if its details page has the following URL:

```
http://teamcity.jetbrains.com:8111/agentDetails.html?id=2
```

To trigger a build on two agents at the same time, use the following URL:

```
http://testuser:testpassword@teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf&agentId=1&agentId=2
```

To trigger a build on all enabled and compatible agents, use "allEnabledCompatible" as agent ID:

```
http://testuser:testpassword@teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf&agentId=allEnabledCompatible
```

## Triggering a Custom Build

TeamCity allows you to trigger a build with customized parameters. You can select particular build agent to run the build, define additional properties and environment variables, and select the particular sources revision (by specifying the last change to include in the build) to run the build with. These customizations will affect only the single triggered build and will not affect other builds of the build configuration.

To trigger a build on a specific change inclusively, use the following URL:

```
http://testuser:testpassword@teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf&modificationId=11112
```

`modificationId` — modification/change internal id which can be obtained from the web diff url.

To trigger a build with custom parameters (system properties and environment variables), use:

```
http://testuser:testpassword@teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf&name=<full property name1>&value=<value1>&name=<full property name2>&value=<value2>
```

Where <full property name> is a full property name with system./env. prefix or no prefix to define configuration parameter. Please note that previous TeamCity versions used different syntax for this action. That syntax is still supported for compatibility reason, though.

To move build to the top of the queue, add the following to the query string

- `&moveToTop=true`

To run a personal build, add `&personal=true` to the query string.

To run a build on a feature branch:

```
http://testuser:testpassword@teamcity.jetbrains.com/httpAuth/action.html?add2Queue=bt10&branch  
Name=master
```

## REST API

On this page:

- General information
    - General Usage Principles
    - REST Authentication
      - Superuser access
    - REST API Versions
    - URL Structure
      - Locator
      - Supported HTTP Methods
    - Response Formats
    - Full and Partial Responses
    - Logging
    - CORS Support
    - API Client Recommendations
    - TeamCity Data Entities Requests
    - Projects and Build Configuration/Templates Lists
    - Project Settings
    - Project Features
    - VCS Roots
      - VCS root instance locator
    - Build Configuration And Template Settings
      - Build Configuration Locator
  - Build Requests
    - Build Locator
    - Queued Builds
    - Triggering a Build
      - Build node example
    - Build Tags
    - Build Pinning
    - Build Canceling/Stopping
    - Build Artifacts
      - Authentication
    - Other Build Requests
      - Changes
      - Revisions
      - Snapshot dependencies
      - Artifact dependencies
      - Build Parameters
      - Build fields
      - Statistics
      - Build log
  - Tests and Build problems
    - Muted tests and build problems
  - Investigations
  - Agents
    - Agent Pools
    - Assigning Projects to Agent Pools
  - Users
  - User Groups
- Other
  - Data Backup

- Typed Parameters Specification
- Build Status Icon
- TeamCity Licensing Information Requests
- CCTray
- Request Examples
  - Request Sending Tool
    - Creating a new project
    - Making user a system administrator

## General information

REST API is an open-source [plugin](#) bundled since TeamCity 5.0.

To use the REST API, an application makes an HTTP request to the TeamCity server and parses the response.

The TeamCity REST API can be used for integrating applications with TeamCity and for those who want to script interactions with the TeamCity server. TeamCity's REST API allows accessing resources (entities) via URL paths.

 The URL examples on this page assume that your TeamCity server web UI is accessible via the <http://teamcity:8111> URL.

## General Usage Principles

This documentation is not meant to be comprehensive, but just provide some initial knowledge useful for using the API.

You can start by opening <http://teamcity:8111/app/rest> URL in your browser: this page will give you several pointers to explore the API.

Use <http://teamcity:8111/app/rest/application.wadl> to get the full list of supported requests and names of parameters. This is the primary source of discovery for the supported requests and their parameters. The same data is also exposed in Swagger format via .../app/rest/swagger.json endpoint You can start with <http://teamcity:8111/app/rest/server> request and then drill down following "href" attributes of the entities listed.

Please make sure you read through this "General information" section before using the API.

For the list of supported [locator dimensions](#), use "\$help" locator.

Experiment and read the error messages returned: for the most part they should guide you to the right requests.

### Example on how to explore the API

Suppose you want to know more on the agents and see (in "/app/rest/server" response) that there is a "/app/rest/agents" URL.

- try the "/app/rest/agents/" request - see the authorized agent list, get the "default" way of linking to an agent from the agent's element href attribute.
- get individual agent details via /app/rest/agents/id:10 URL (obtained from "href" for one of the elements of the previous request).
- if you send a request to "/app/rest/agents/\$help", or "/app/rest/agents/aaa:bbb" (supplying unsupported locator dimension), you will get the list of the supported dimensions to find an agent via the agent's locator
- most of the attributes of the returned agent data (name, connected, authorized) can be used as "<field name>" in the "app/rest/agents/<agentLocator>/<field name>" request. Moreover, if you issue a request to the "app/rest/agents/id:10/test" URL, you will get a list of the supported fields in the error message

## REST Authentication

You can authenticate yourself for the REST API in the following ways:

- Using basic HTTP authentication (it can be slow with certain authentications, see below). Provide a valid TeamCity username and password with the request. You can force basic auth by including "httpAuth" before the "/app/rest" part: e.g. <http://teamcity:8111/httpAuth/app/rest/builds>
- Using access to the server as a [guest user](#) (if enabled) include "guestAuth" before the "/app/rest" part: e.g.: <http://teamcity:8111/guestAuth/app/rest/builds>
- if you are checking REST GET requests from within a browser and you are logged in to TeamCity in the browser, you can just use "/app/rest" URL: e.g. <http://teamcity:8111/app/rest/builds>

Authentication can be slow when not built-in authentication module is used, consider applying the [session reuse approach](#) for reusing authentication between sequential requests.

If you perform a request from within a TeamCity build, for a limited set of build-related operations (like downloading artifacts) you can use values of [teamcity.auth.userId/teamcity.auth.password](#) system properties as credentials (within TeamCity

settings you can reference them as `%system.teamcity.auth.userId%` and `%system.teamcity.auth.password%`).

Within a build, a request for current build details can look like:

```
curl -u "%system.teamcity.auth.userId%:%system.teamcity.auth.password%"  
"%teamcity.serverUrl%/httpAuth/app/rest/builds/id:%teamcity.build.id%"
```

## Superuser access

You can use the [super user account](#) with REST API: just provide no user name and the generated password logged into the server log.

## REST API Versions

As REST API evolves from one TeamCity version to another, there can be incompatible changes in the protocol.

Under the `http://teamcity:8111/app/rest/` or `http://teamcity:8111/app/rest/latest` URL the latest version is available.

Under the `http://teamcity:8111/app/rest/<version>` URL, the current version is available and earlier versions CAN be available. Our general policy is to supply TeamCity with at least one previous version.

e.g. in TeamCity 2018.1 for `<version>` you can use "2018.1" for the current and "2017.2", "2017.1", "10.0", "9.1", "9.0", "8.1", "8.0" to get earlier versions of the protocol. The protocol version corresponds to the TeamCity version where it was first introduced.

Breaking changes in the API are described in the related [Upgrade Notes](#) section.

Please note that additions to the objects returned (such as new XML attributes or elements) are not considered major changes and do not cause the protocol version to increment.

Also, the endpoints marked with "Experimental" comment in `application.wadl` may change without a special notice in future versions.

Note: The examples on this page use the `"/app/rest"` relative URL, replace it with the one containing the version if necessary.

## URL Structure

The general structure of the URL in the TeamCity API is `teamcityserver:port/<authType>/app/rest/<apiVersion>/<restApiPath>?<parameters>`, where

- `teamcityserver` and `port` define the server name and the port used by TeamCity. This page uses "`http://teamcity:8111/`" as example URL
- `<authType>` (optional) is the [authentication type](#) to be used, this is generic TeamCity functionality
- `app/rest` is the root path of TeamCity REST API
- `<apiVersion>` (optional) is a reference to specific version of REST API
- `<restApiPath>?<parameters>` is the REST API part of the URL

When `<restApiPath>` represents a collection of items `.../app/rest/<items>` (e.g. `.../app/rest/builds`), then the URL regularly accepts the "[locator](#)" parameter which can filter the items returned. Individual items can regularly be addressed by a URL in the form of `.../app/rest/<item>/<item_locators>`. Both multiple and single items requests regularly support the [fields](#) parameter.

## Locator

In a number of places, you can specify a filter string which defines what entities to filter/affect in the request. This string representation is referred to as "locator" in the scope of REST API.

The locators formats can be:

- single value: a string without the following symbols: `, :-( )`
- dimension, allowing to filter entities using multiple criteria: `<dimension1>:<value1>, <dimension2>:<value2>, <dimension3>:(<dimension3.1>:<value3.1>, <dimension3.2>:<value3.2>)`

Refer to each entity description below for the most popular locator descriptions.

If in doubt what a specific locator supports, send a request with "\$help" as the locator value. In response you will get a textual description of what the locator supports. If a request with invalid locators is sent, the error messages often hint at the error and list the supported locator dimensions as well.

Note: If the value contains the `,` symbol, it should be enclosed into parentheses: `"(<value>)"`. The value of a dimension can also be encoded as Base64url ("URL and Filename safe type base64" from RFC4648) and sent as `"<dimension>:($base64:<base64-encoded-value>)"` instead of `"<dimension>: <value>"`.

Examples:

```

http://teamcity:8111/app/rest/projects gets you the list of projects
http://teamcity:8111/app/rest/projects/<projectsLocator>- http://teamcity:8111/app/rest/projects/id:RESTAPI
Plugin (the example id is used) gets you the full data for the REST API Plugin project.
http://teamcity:8111/app/rest/buildTypes/id:bt284/builds?locator=<buildLocator> - http://teamcity:8111/app/
rest/buildTypes/id:bt284/builds?locator=status:SUCCESS,tag:EAP - (example ids are used) to get builds
http://teamcity:8111/app/rest/builds/?locator=<buildLocator> - to get builds by build locator.

```

## Supported HTTP Methods

- GET: retrieves the requested data. e.g. usually .../app/rest/entities retrieves a list of entities, .../app/rest/entities/<entity locator> retrieves a single entity
- POST: creates the entity in the request adding it to the existing collection. When posting XML, be sure to specify the "Content-Type: application/xml" HTTP header. e.g. to create a new entity, one regularly needs to post a single entity data to the .../app/rest/entities URL
- PUT: based on the existence of the entity, creates or updates the entity in the request. e.g. supported for some entities, for URLs like .../app/rest/entities/<entity locator>
- DELETE: removes the requested data e.g. for the .../app/rest/entities/<entity locator> URL

## Response Formats

The TeamCity REST APIs returns HTTP responses in the following formats according to the HTTP "Accept" header:

Format	Response Type	HTTP "Accept" header value
plain text	single-value responses	text/plain
XML	complex value responses	application/xml
JSON	complex value responses	application/json

## Full and Partial Responses

By default, when a list of entities is requested, only basic fields are included into the response. When a single entry is requested, all the fields are returned. The complex field values can be returned in full or basic form, depending on a specific entity.

It is possible to change the set of fields returned for XML and JSON responses for the majority of requests.

This is done by supplying the fields request parameter describing the fields of the top-level entity and sub-entities to return in the response. An example syntax of the parameter is: field,field2(field2\_subfield1,field2\_subfield1). This basically means "include field and field2 of the top-level entity and for field2 include field2\_subfield1 and field2\_subfield1 fields". The order of the fields specification plays no role.Examples:

```

http://teamcity.jetbrains.com/app/rest/buildTypes?locator=affectedProject:(id:TeamCityPluginsByJetBrains)&f
ields=buildType(id,name,project)
http://teamcity.jetbrains.com/app/rest/builds?locator=buildType:(id:bt345),count:10&fields=count,build(numb
er,status,statusText,agent,lastChange,pinned)

```

At this time, the response can sometimes include the fields/elements not specifically requested. This can change in the future versions, so it is recommended to specify all the fields/elements used by the client.

## Logging

You can get details on errors and REST request processing in logs\teamcity-rest.log **server log**.

If you get an error in response to your request and want to investigate the reason, look into **rest-related server logs**.

To get details about each processed request, turn on debug logging (e.g. set Logging Preset to "debug-rest" on the **Administration/Diagnostics** page or modify the Log4J "jetbrains.buildServer.server.rest" category).

## CORS Support

TeamCity REST can be configured to allow **cross-origin requests** using the `rest.cors.origins` **internal property**.

To allow requests from a page loaded from a specific domain:

- Add the page address (including the protocol and port , do not use wildcards) to the comma-separated **internal property** `rest.cors.origins`, e.g.

```
rest.cors.origins=http://myinternalwebpage.org.com:8080,https://myinternalwebpage.org.com
```

To enable support for a **preflight OPTIONS** request:

1. Add the `rest.cors.optionsRequest.allowUnauthorized=true` internal property.
2. Restart the TeamCity server.
3. Use the '`/app/rest/latest`' URL for the requests Do not use '`/app/rest`', do not use the '`httpAuth`' prefix.

If that does not help, enable debug **logging** and look for related messages. If there are none, capture the browser traffic and messages to investigate the case.

## API Client Recommendations

When developing a client using REST API, consider the following recommendations:

- Make root REST API URL configurable (e.g. allow to specify an alternative for "`app/rest/<version>`" part of the URL). This will allow to direct the client to another version of the API if necessary.
- Ignore (do not error out) item's attributes and sub-items which are unknown to the client. New sub-items are sometimes added to the API without version change and this will ensure the client is not affected by the change.
- Set large (and make them configurable) request timeouts. Some API calls can take minutes, especially on a large server.
- Use HTTP sessions to make consecutive requests (use `TCSESSIONID` cookie returned from the first authenticated response instead of supplying raw credentials all the time). This saves time on authentication which can be significant for external authentication providers.
- Beware of partial answers when requesting list of items: some requests are paged by default. Value of the "count" attribute in the response indicate the number of the items on the current page and there can be more pages available. If you need to process more (e.g. all) items, read and process "`nextHref`" attribute of the response entity for items collections. If the attribute is present it means there might be more items when queried by the URL provided. Related locator dimensions are "count" (page limit) and "lookupLimit" (depth of search). Even when the returned "count" is 0, it does not mean there are no more items if there is "`nextHref`" attribute present.
- Do not increase the "`lookupLimit`" value in the locators without a second thought. Doing so has the direct effect of loading the server more and may require increased amounts of CPU and memory. It is assumed that those increasing the default limit understand the negative consequences for the server performance.
- Do not abuse the ability to execute automated requests for TeamCity API: do not query the API too frequently and restrict the data requested to only that necessary (using due **locators** and specifying necessary **fields**). Check the server behavior under load from your requests. Make sure not to repeat the request frequently if it takes time to process the request.

## TeamCity Data Entities Requests

### Projects and Build Configuration/Templates Lists

List of projects: `GET http://teamcity:8111/app/rest/projects`

Project details: `GET http://teamcity:8111/app/rest/projects/<projectLocator>` where `<projectLocator>` can be `id:<internal_project_id>` or `name:<project%20name>`

List of Build Configurations: `GET http://teamcity:8111/app/rest/buildTypes`

List of Build Configurations of a project: `GET http://teamcity:8111/app/rest/projects/<projectLocator>buildTypes`

Get projects with sub-projects/ Build Configurations data and their order as configured by the specified user on the Overview page: `GET http://teamcity:8111/app/rest/projects?locator=selectedByUser:current&fields=count,project(id,parentProjectId,projects(count,project(id,$locator(selectedByUser:current))),buildTypes(count,buildType(id),$locator(selectedByUser:current)))`

List of templates for a particular project: `GET http://teamcity:8111/app/rest/projects/<projectLocator>/templates`  
List of all the templates on the server: `GET http://teamcity:8111/app/rest/buildTypes?locator=templateFlag:true`

### Project Settings

Get project details: `GET http://teamcity:8111/app/rest/projects/<projectLocator>/`

Delete a project: `DELETE http://teamcity:8111/app/rest/projects/<projectLocator>/`

Create a new empty project: `POST plain text (name) to http://teamcity:8111/app/rest/projects/`

Create (or copy) a project: `POST XML <newProjectDescription name='New Project Name' id='newProjectId' copyAllAssociatedSettings='true'><parentProject locator='id:project1' /><sourceProject locator='id:project2' /></newProjectDescription> to http://teamcity:8111/app/rest/projects. Also see an example.`

Edit project parameters: `GET/DELETE/PUT http://teamcity:8111/app/rest/projects/<projectLocator>/parameters/<parameter_name>` (produces XML, JSON and plain text depending on the "Accept" header, accepts plain text and XML and JSON)

Also supported are requests `.../parameters/<parameter_name>/name` and `.../parameters/<parameter_name>/value`.

Project name/description/archived status: GET/PUT [http://teamcity:8111/app/rest/projects/<projectLocator>/<field\\_name>](http://teamcity:8111/app/rest/projects/<projectLocator>/<field_name>) (accepts/produces text/plain) where <field\_name> is one of "name", "description", "archived".

Project's parent project: GET/PUT XML <http://teamcity:8111/app/rest/projects/<projectLocator>/parentProject>

## Project Features

Project features (e.g. issue trackers, versioned settings, custom charts, shared resources and third-party report tabs) are exposed as entries under the "project" node and via dedicated requests.

List of project features: <http://teamcity:8111/app/rest/projects/<projectLocator>/projectFeatures> To filter features, add "?locator=<projectFeaturesLocator>" to the URL e.g. to find all issue tracker features of GitHub type, use the locator "type:IssueTracker,property(name:type,value:GithubIssues)"

Create feature: POST to <http://teamcity:8111/app/rest/projects/<projectLocator>/projectFeatures>

Edit features: GET/DELETE/PUT <http://teamcity:8111/app/rest/projects/<projectLocator>/projectFeatures/<featureId>>

## VCS Roots

List all VCS roots: GET <http://teamcity:8111/app/rest/vcs-roots>, add locator=<vcsRootLocator> parameter to list only the VCS roots matched

Get details of a VCS root/delete a VCS root: GET/DELETE <http://teamcity:8111/app/rest/vcs-roots/<vcsRootLocator>>, where "<vcsRootLocator>" can be "id:<internal VCS root id>" or other VCS root locator

Create a new VCS root: POST VCS root XML (similar to the one retrieved by a GET request for VCS root details) to <http://teamcity:8111/app/rest/vcs-roots>

Also supported:

GET/PUT [http://teamcity:8111/app/rest/vcs-roots/<vcsRootLocator>/properties/<property\\_name>](http://teamcity:8111/app/rest/vcs-roots/<vcsRootLocator>/properties/<property_name>)  
GET/PUT [http://teamcity:8111/app/rest/vcs-roots/<vcsRootLocator>/<field\\_name>](http://teamcity:8111/app/rest/vcs-roots/<vcsRootLocator>/<field_name>), where <field\_name> is "id", "name", "project" (post project locator to "project" to associate a VCS root with a specific project).

List VCS root instances: GET <http://teamcity:8111/app/rest/vcs-root-instances?locator=<vcsRootInstancesLocator>>

A 'VCS root' is the setting configured in the TeamCity UI, a "VCS root instance" is an internal TeamCity entity which is derived from the "VCS root" to perform the actual VCS operation.

If a VCS root has no %-references to parameters, a single VCS root corresponds to a single "VCS root instance".

If a VCS root has %-reference to a parameter and the reference resolves to a different value when the VCS root is attached to different configurations or when custom builds are run, a single "VCS root" can generate several "VCS root instances".

Since TeamCity 10.0:

There are two endpoints dedicated to being used in commit hooks from the version control repositories:

POST <http://teamcity:8111/app/rest/vcs-root-instances/checkingForChangesQueue?locator=<vcsRootInstancesLocator>> - schedules checking for changes for the matched VCS root instances and returns the list of VCS root instances matched (just like GET <http://teamcity:8111/app/rest/vcs-root-instances?locator=<vcsRootInstancesLocator>>)

POST <http://teamcity:8111/app/rest/vcs-root-instances/commitHookNotification?locator=<vcsRootInstancesLocator>> - schedules checking for changes for the matched VCS root instances and returns plain-text human-readable message on the action performed, HTTP response 202 in case of successful operation

Both perform the same action (put the VCS root instances matched by the <locator>) to the queue for "checking for changes" process and differ only in responses they produce.

Note that since the matched VCS root instances are the same as for <http://teamcity:8111/app/rest/vcs-root-instances?locator=<locator>> request and that means that by default only the first 100 are matched and the rest are ignored. If this limit is reached, consider tweaking the <locator> to match fewer instances (recommended) or increase the limit, e.g. by adding ".count:1000" to the locator.

## VCS root instance locator

Some of the supported "<vcsRootInstancesLocator>" from above:

type:<VCS root type> - VCS root instances of the specified version control (e.g. "jetbrains.git", "mercurial", "svn")

vcsRoot:( <vcsRootLocator>) - VCS root instances corresponding to the VCS root matched by "<vcsRootLocator>"

buildType:(<buildTypeLocator>) - VCS root instances attached to the matching build configuration

property:(name:<name>,value:<value>,matchType:<matching>) - VCS root instances with the property of name "<name>" and value matching condition "<matchType>" (e.g. equals, contains) by the value "<value>".

## Build Configuration And Template Settings

Build Configuration/Template details: GET <http://teamcity:8111/app/rest/buildTypes/<buildConfigurationLocator>> (details on the [Build Configuration locator](#)).

Please note that there is no transaction, etc. support for settings editing in TeamCity, so all the settings modified via REST API are taken into account at once. This can result in half-configured builds triggered, etc. Please make sure you pause a build configuration before changing its settings if this aspect is important for your case.

To get aggregated status for several build configurations, see [Build Status Icon](#) section.

Get/set paused build configuration state: GET/PUT <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/paused> (put "true" or "false" text as text/plain)

Build configuration settings: GET/DELETE/PUT [http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/settings/<setting\\_name>](http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/settings/<setting_name>)

Build configuration parameters: GET/DELETE/PUT [http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/parameters/<parameter\\_name>](http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/parameters/<parameter_name>)

(produces XML, JSON and plain text depending on the "Accept" header, accepts plain text and XML and JSON). The requests [.../parameters/<parameter\\_name>/name](http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/parameters/<parameter_name>/name) and [.../parameters/<parameter\\_name>/value](http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/parameters/<parameter_name>/value) are also supported.

Build configuration steps: GET/DELETE [http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/steps/<step\\_id>](http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/steps/<step_id>)

Create build configuration step: POST <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/steps>. The XML/JSON posted is the same as retrieved by GET request to [.../steps/<step\\_id>](http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/steps/<step_id>) except for the secure settings like password: these are not included into responses and should be supplied before POSTing back

Features, triggers, agent requirements, artifact and snapshot dependencies follow the same pattern as steps with URLs like:

<http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/features/<id>>

<http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/triggers/<id>>

<http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/agent-requirements/<id>>

<http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/artifact-dependencies/<id>>

<http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/snapshot-dependencies/<id>>

Since TeamCity 10, it is possible to disable/enable artifact dependencies and agent requirements:

Disable/enable an artifact dependency PUT <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/artifact-dependencies/<id>/disabled> (put "true" or "false" text as text/plain)

Disable/enable an agent requirement PUT <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/agent-requirements/<id>/disabled> (put "true" or "false" text as text/plain)

Build configuration VCS roots: GET/DELETE <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/vcs-root-entries/<id>>

Attach VCS root to a build configuration: POST <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/vcs-root-entries>. The XML/JSON posted is the same as retrieved by GET request to <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/vcs-root-entries/<id>> except for the secure settings like password: these are not included into responses and should be supplied before POSTing back.

Create a new build configuration with all settings: POST <http://teamcity:8111/app/rest/buildTypes>. The XML/JSON posted is the same as retrieved by GET request. (Note that [/app/rest/project/XXX/buildTypes](http://teamcity:8111/app/rest/project/XXX/buildTypes) still uses the previous version notation and accepts another entity.)

Create a new empty build configuration: POST plain text (name) to <http://teamcity:8111/app/rest/projects/<projectLocator>/buildTypes>

Copy a build configuration: POST XML <newBuildTypeDescription name='Conf Name' sourceBuildTypeLocator='id:XXX' copyAllAssociatedSettings='true' shareVCSRoots='false' /> to <http://teamcity:8111/app/rest/projects/<projectLocator>/buildTypes>

Since TeamCity 2017.2: Read, detach and attach a build configuration from/to a template: GET/DELETE/POST/PUT <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/templates>

Before 2017.2: Read, detach and attach a build configuration from/to a template: GET/DELETE/PUT <http://teamcity:8111/app/rest/buildTypes/<buildTypeLocator>/template> (PUT accepts template locator with the "text/plain" Content-Type)

▼ Some examples: [click to expand](#)

Set build number counter:

```
curl -v --basic --user <username>:<password> --request PUT  
http://<teamcity.url>/app/rest/buildTypes/<buildTypeLocator>/settings/buildNumbe  
rCounter --data <new number> --header "Content-Type: text/plain"
```

Set build number format:

```
curl -v --basic --user <username>:<password> --request PUT  
http://<teamcity.url>/app/rest/buildTypes/<buildTypeLocator>/settings/buildNumbe  
rPattern --data <new format> --header "Content-Type: text/plain"
```

## Build Configuration Locator

The most frequently used values for "<buildTypeLocator>" are `id:<buildConfigurationOrTemplate_id>` and `name:<Build%20Configuration%20name>`.

Since TeamCity 2017.2, the experimental `type` locator is supported with one of the values: `regular`, `composite` or `deployment`

Other supported `dimensions` are (these are in experimental state):

`internalId` - internal id of the build configuration

`project` - `<projectLocator>` to limit the build configurations to those belonging to a single project

`affectedProject` - `<projectLocator>` to limit the build configurations under a single project (recursively)

`template` - `<buildTypeLocator>` of a template to list only build configurations using the template

`templateFlag` - boolean value to get only templates or only non-templates

`paused` - boolean value to filter paused/not paused build configurations

## Build Requests

List builds: `GET http://teamcity:8111/app/rest/builds/?locator=<buildLocator>`

Get details of a specific build: `GET http://teamcity:8111/app/rest/builds/<buildLocator>` (also supports `DELETE` to delete a build)

Get the list of build configurations in a project with the status of the last finished build in each build configuration:

```
GET http://teamcity:8111/app/rest/buildTypes?locator=affectedProject:(id:ProjectId)&fields=buildType(id,nam  
e,builds($locator(running:false,canceled:false,count:1),build(number,status,statusText)))
```

## Build Locator

Using a `locator` in build-related requests, you can filter the builds to be returned in the build-related requests. It is referred to as "build locator" in the scope of REST API.

For some requests, a default filtering is applied which returns only "normal" builds (finished builds which are not canceled, not failed-to-start, not personal, and on default branch (in branched build configurations)), unless those types of builds are specifically requested via the locator. To turn off this default filter and process all builds, add "`defaultFilter:false`" dimension to the build locator. Default filtering varies depending on the specified locator dimensions. e.g. when "agent" or "user" dimensions are present, personal, canceled and failed to start builds are included into the results.

Examples of supported build locators:

- `id:<internal build id>` - use `internal build id` when you need to refer to a specific build
- `number:<build number>` - to find build by build number, provided build configuration is already specified
- `<dimension1>:<value1>,<dimension2>:<value2>` - to find builds by multiple criteria

The list of supported build locator dimensions:

`project:<project locator>` - limit the list to the builds of the specified project (belonging to any build type directly under the project).

`affectedProject:<project locator>` - limit the list to the builds of the specified project (belonging to any build type directly or indirectly under the project)

`buildType:(<buildTypeLocator>),defaultFilter:false` - all the builds of the specified build configuration

`tag:<tag>` - since TeamCity 10 get tagged builds. If a list of tags is specified, e.g. `tag:<tag1>, tag:<tag2>`, only the builds containing all the specified tags are returned. The legacy `tags:<tags>` locator is supported for compatibility

status:<SUCCESS/FAILURE/ERROR> - list builds with the specified status only  
user:<userLocator> - limit builds to only those triggered by the user specified  
personal:<true/false/any> - limit builds by the personal flag. By default, personal builds are not included.  
canceled:<true/false/any> - limit builds by the canceled flag. By default, canceled builds are not included.  
failedToStart:<true/false/any> - limit builds by the failed to start flag. By default, canceled builds are not included.  
state: <queued/running/finished> - limit builds by the specified state.  
running:<true/false/any> - limit builds by the running flag. By default, running builds are not included.  
state:running,hanging:true - fetch hanging builds (since TeamCity 10.0)  
pinned:<true/false/any> - limit builds by the pinned flag.  
branch:<branch locator> - limit the builds by branch. <branch locator> can be the branch name displayed in the UI, or "(name:<name>,default:<true/false/any>,unspecified:<true/false/any>,branched:<true/false/any>)". By default only builds from the default branch are returned. To retrieve all builds, add the following locator: branch:default:any. The whole path will look like this: /app/rest/builds/?locator=buildType:One\_Git,branch:default:any  
revision:<REVISION> - find builds by revision, e.g. all builds of the given build configuration with the revision: /app/rest/builds?locator=revision:(REVISION),buildType:(id:BUIDL\_TYPE\_ID). See more information [below](#).  
agentName:<name> - agent name to return only builds ran on the agent with name specified  
sinceBuild:<buildLocator> - limit the list of builds only to those after the one specified  
sinceDate:<date> - limit the list of builds only to those started after the date specified. The date should be in the same format as dates returned by REST API (e.g. "20130305T170030+0400").  
queuedDate/startDate/finishDate:(date:<time-date>,build:<build locator>,condition:<before/after>) - filter builds based on the time specified by the build locator, e.g. for the builds finished after November 23, 2017, 20:34:46, GMT+1 timezone use: finishDate:(date:20171123T203446%2B0100,condition:after)  
count:<number> - serve only the specified number of builds  
start:<number> - list the builds from the list starting from the position specified (zero-based)  
lookupLimit:<number> - limit processing to the latest N builds only (the default is 5000). If none of the latest N builds match the other specified criteria of the build locator, 404 response is returned for single build request and empty collection for multiple builds request. See related note in the [section above](#)

## Queued Builds

GET <http://teamcity:8111/app/rest/buildQueue>

Supported locators:

- project:<locator>
- buildType:<locator>

Get details of a queued build:

GET <http://teamcity:8111/app/rest/buildQueue/id:XXX>

For queued builds with snapshot dependencies, the revisions are available in the `revisions` element of the queued build node if a revision is fixed (for regular builds without snapshot dependencies it is not).

Get compatible agents for queued builds (useful for builds having "No agents" to run on)

GET <http://teamcity:8111/app/rest/buildQueue/id:XXX/compatibleAgents>

Examples:

List queued builds per project:

GET <http://teamcity:8111/app/rest/buildQueue?locator=project:<locator>>

List queued builds per build configuration:

GET <http://teamcity:8111/app/rest/buildQueue?locator=buildType:<locator>>

## Triggering a Build

To start a build, send a POST request to <http://teamcity:8111/app/rest/buildQueue> with the "build" node (see below) in content - the same node as details of a queued build or finished build. The queued build details will be returned.

When the build is started, the request to the queued build (/app/rest/buildQueue/XXX) will return running/finished build data. This way, you can monitor the build completeness by querying build details using the "href" attribute of the build details returned on build triggering, until the build has the `state="finished"` attribute.

## Build node example

Basic build for a build configuration:

```
<build>
  <buildType id="buildConfID"/>
</build>
```

Build for a branch marked as personal with a fixed agent, comment and a custom parameter:

```
<build personal="true" branchName="logicBuildBranch">
  <buildType id="buildConfID"/>
  <agent id="3"/>
  <comment><text>build triggering comment</text></comment>
  <properties>
    <property name="env.myEnv" value="bbb"/>
  </properties>
</build>
```

Queued build assignment to an agent pool:

```
<build>...
  <agent>
    <pool id="N"/>
  </agent>
...
</build>
```

Build on a change of given revision, forced rebuild of all dependencies and clean sources before the build, moved to the build queue top on triggering. (Note that the change should be already known to TeamCity (displayed in UI for the build configuration, [more on "lastChanges" element](#)):

```
<build>
  <triggeringOptions cleanSources="true" rebuildAllDependencies="true" queueAtTop="true"/>
  <buildType id="buildConfID"/>
  <lastChanges>
    <change
locator="version:a286767fc1154b0c2b93d5728dd5bbcdefdfaca,buildType:(id:buildConfID)"/>
  </lastChanges>
</build>
```

▼ Example command line for the build triggering: click to expand

```
curl -v -u user:password http://teamcity.server.url:8111/app/rest/buildQueue
--request POST --header "Content-Type:application/xml" --data-binary @build.xml
```

## Build Tags

Get tags: GET <http://teamcity:8111/app/rest/builds/<buildLocator>/tags/>

Replace tags: PUT <http://teamcity:8111/app/rest/builds/<buildLocator>/tags/> (put the same XML or JSON as returned by GET)

Add tags: POST `http://teamcity:8111/app/rest/builds/<buildLocator>/tags/` (post the same XML or JSON as returned by GET or just a plain-text tag name)  
(`<buildLocator>` here should match a single build only)

## Build Pinning

Get current pin status: GET `http://teamcity:8111/app/rest/builds/<buildLocator>/pin/` (returns "true" or "false" text)  
Pin: PUT `http://teamcity:8111/app/rest/builds/<buildLocator>/pin/` (the text in the request data is added as a comment for the action)  
Unpin: DELETE `http://teamcity:8111/app/rest/builds/<buildLocator>/pin/` (the text in the request data is added as a comment for the action)  
(`<buildLocator>` here should match a single build only)

## Build Canceling/Stopping

Cancel a running or a queued build: POST the `<buildCancelRequest comment='CommentText' readdIntoQueue='false' />` item to the URL of a running or a queued build:

### Example of cancelling a queued build: click to expand

```
curl -v -u user:password --request POST "http://teamcity:8111/app/rest/buildQueue/<buildLocator >"  
--data "<buildCancelRequest comment='' readdIntoQueue='false' />" --header "Content-Type:  
application/xml"
```

Stop a running build and readd it to the queue: POST the `<buildCancelRequest comment='CommentText' readdIntoQueue='true' />` item to the URL of a running build:

### Example of cancelling a running build: click to expand

```
curl -v -u user:password --request POST "http://teamcity:8111/app/rest/builds/<buildLocator >" --data  
"<buildCancelRequest comment='' readdIntoQueue='true' />" --header "Content-Type: application/xml"
```

Expose cancelled build details:

See the `canceledInfo` element of the build item (available via GET `http://teamcity:8111/app/rest/builds/<buildLocator >`)

## Build Artifacts

GET `http://teamcity:8111/app/rest/builds/<build_locator>/artifacts/content/<path>` (returns the content of a build artifact file for a build determined by `<build_locator>`)

**i** `<path>` above can be empty for the root of build's artifacts or be a path within the build's artifacts. The path can span into the archive content, e.g. `dir/path/archive.zip!/path_within_archive`

Media-Type: application/octet-stream or a more specific media type (determined from artifact file extension)  
Possible error: 400 if the specified path references a directory

GET `http://teamcity:8111/app/rest/builds/<build_locator>/artifacts/metadata/<path>` (returns information about a build artifact)

Media-Type: application/xml or application/json

GET `http://teamcity:8111/app/rest/builds/<build_locator>/artifacts/children/<path>` (returns the list of artifact children for directories and archives)

Media-Type: application/xml or application/json

Possible error: 400 if the artifact is neither a directory nor an archive

GET `http://teamcity:8111/app/rest/builds/<build_locator>/artifacts/archived/<path>?locator=pattern:<wildcard>` (returns the archive containing the list of artifacts under the path specified. The optional locator parameter can have file `<wildcard>` to limit the files only to those matching the `wildcard`)

Media-Type: application/zip

Possible error: 400 if the artifact is neither a directory nor an archive `<artifact relative name>` supports referencing files under archives using `!/` delimiter after the archive name.

Examples:

```
GET http://teamcity:8111/app/rest/builds/id:100/artifacts/children/my-great-tool-0.1.jar\!/META-INF  
GET http://teamcity:8111/app/rest/builds/buildType:(id:Build_Installers),status:SUCCESS/artifacts/metadata/m  
y-great-tool-0.1.jar\!/META-INF/MANIFEST.MF  
GET http://teamcity:8111/app/rest/builds/buildType:(id:Build_Installers),number:16.7.0.2/artifacts/metadata/  
my-great-tool-0.1.jar!/lib/commons-logging-1.1.1.jar!/META-INF/MANIFEST.MF
```

```
GET http://teamcity:8111/app/rest/builds/buildType:(id:Build_Installers).tag:release/artifacts/content/my-gr  
eat-tool-0.1.jar!/lib/commons-logging-1.1.1.jar!/META-INF/MANIFEST.MF
```

## Authentication

If you download artifacts from within a TeamCity build, consider using `teamcity.auth.userId`/`teamcity.auth.password` system properties as credentials for the download artifacts request: this way TeamCity will have a way to record that one build used artifacts of another and will display it on the build's Dependencies tab.

## Other Build Requests

### Changes

#### <changes>

<changes> is meant to represent changes the same way as displayed in the build's Changes in TeamCity UI. In the most cases these are the commits between the current and previous build. The <changes> tag is not included into the build by default, it has the href attribute only. If you execute the request specified in the href, you'll get the required changes.

Get the list of all of the changes included into the build: GET [http://teamcity:8111/app/rest/changes?locator=build:\(id:<buildId>\)](http://teamcity:8111/app/rest/changes?locator=build:(id:<buildId>))

Get details of an individual change: GET <http://teamcity:8111/app/rest/changes/id:<changeId>>

Get information about a changed file action: the files node lists changed files. The information about the changed file action is reported via the `changeType` attribute for the files listed as one of the following: added, edited, removed, copied or unchanged.

Filter all changes by a locator: GET <http://teamcity:8111/app/rest/changes?locator=<changeLocator>>

Note that the change id is the change's internal id, not the revision. The id can be seen in the change node listed by the REST API or in the URL of the change details (as modId).

Get all changes for a project: GET <http://teamcity:8111/app/rest/changes?locator=project:<projectId>>

Get all the changes in a build configuration since a particular change identified by its id: [http://teamcity:8111/app/rest/changes?locator=buildType:\(id:<buildConfigurationId>\),sinceChange:\(id:<changeId>\)](http://teamcity:8111/app/rest/changes?locator=buildType:(id:<buildConfigurationId>),sinceChange:(id:<changeId>))

Get pending changes for a build configuration [http://teamcity:8111/app/rest/changes?locator=buildType:\(id:<BUILD\\_CONFIG\\_ID>\),pending:true](http://teamcity:8111/app/rest/changes?locator=buildType:(id:<BUILD_CONFIG_ID>),pending:true)

#### <lastChanges>

The <lastChanges> tag contains information about the last commit included into the build. When triggering a build, it's nested <change> element can contain "locator" field which specified which change to use for the build triggering.

### Revisions

#### <revisions>

The <revisions> tag the same as revisions table on the build's Changes tab in TeamCity UI: it lists the revisions of all of the VCS repositories associated with this build that will be checked out by the build on the agent.

A revision might or might not correspond to a change known to TeamCity. e.g. for a newly created build configuration and a VCS root, a revision will have no corresponding change.

Get all builds with the specified revision: [http://teamcity:8111/app/rest/builds?locator=revision\(version:<XXXX>\)](http://teamcity:8111/app/rest/builds?locator=revision(version:<XXXX>))

#### <versionedSettingsRevision>

Since TeamCity 10, <versionedSettingsRevision> is added to represent revision of the versioned settings of the build.

### Snapshot dependencies

It is possible to retrieve the entire build chain (all snapshot-dependency-linked builds) for a particular build:

```
http://teamcity:8111/app/rest/builds?locator=snapshotDependency:(to:(id:XXXX),includeInitial:true),defaultFilter:false
```

This gets all the snapshot dependency builds recursively for the build with id XXXX

It possible to find all the snapshot-dependent builds for a particular build:

```
http://teamcity:8111/app/rest/builds?locator=snapshotDependency:(from:(id:XXXX),includeInitial:true),defaultFilter:false
```

### Artifact dependencies

Since TeamCity 10.0.3, there is an experimental ability to:

- get all the builds which downloaded artifacts from the build with the given ID (Delivered artifacts in the TeamCity Web UI):  
GET `http://teamcity:8111/app/rest/builds?locator=artifactDependency:(from:(id:<build ID>),recursive:false)`
- get all the builds whose artifacts were downloaded by the build with the given ID (Downloaded artifacts in the TeamCity Web UI):  
GET `http://teamcity:8111/app/rest/builds?locator=artifactDependency:(to:(id:<build ID>),recursive:false)`

## Build Parameters

Get the parameters of a build: `http://teamcity:8111/app/rest/builds/id:<build id>/resulting-properties`

## Build fields

Get single build's field: GET `http://teamcity:8111/app/rest/builds/<buildLocator>/<field_name>` (accepts/produces text/plain) where `<field_name>` is one of "number", "status", "id", "branchName" and other build's bean attributes

## Statistics

Get statistics for a single build: GET `http://teamcity:8111/app/rest/builds/<buildLocator>/statistics/` only standard/bundled statistic values are listed. See also [Custom Charts](#)

Get single build statistics value: GET `http://teamcity:8111/app/rest/builds/<buildLocator>/statistics/<value_name>`

Get statistics for a list of builds: GET `http://teamcity:8111/app/rest/builds?locator=BUILDS_LOCATOR&fields=build(id,number,status,buildType(id,name,projectName),statistics(property(name,value)))`

## Build log

Downloading build logs via a REST request is not supported, but there is a way to download the log files described [here](#).

## Tests and Build problems

List build problems: GET `http://teamcity:8111/app/rest/problemOccurrences?locator=build:(BUILD_LOCATOR)`

List tests: GET `http://teamcity:8111/app/rest/testOccurrences?locator=<locator dimension>:<value>`

Supported locators:

- `build:(<build locator>)` - test run in the build
- `build:(<build locator>),muted:true` - failed tests which were muted in the build
- `currentlyFailing:true,affectedProject:<project locator>` - tests currently failing under the project specified (recursively)
- `currentlyMuted:true,affectedProject:<project locator>` - tests currently muted under the project specified (recursively) - See also project's Muted Problems tab

Examples:

List all build's tests: GET `http://teamcity:8111/app/rest/testOccurrences?locator=build:<buildLocator>`

Get individual test history:

GET `http://teamcity:8111/app/rest/testOccurrences?locator=test:<testLocator>`

List build's tests which were muted when the build ran:

GET `http://teamcity:8111/app/rest/testOccurrences?locator=build:(id:XXX),muted:true`

List currently muted tests (muted since the failure):

GET `http://teamcity:8111/app/rest/testOccurrences?locator=build:(id:XXX),currentlyMuted:true`

Supported test locators:

- "id:<internal test id>" available as a part of the URL on the test history page
- "name:<full test name>"

Since TeamCity 10 there is experimental support for exposing single test invocations / runs:

Get invocations of a test:

```
GET http://teamcity:8111/app/rest/testOccurrences?locator=build:(id:XXX),test:(id:XXX)&fields=$long,testOccurrences($short,invocations($long))
```

List all test runs with all the invocations flattened:

```
GET http://teamcity:8111/app/rest/testOccurrences?locator=build:(id:XXX),test:(id:XXX),expandInvocations:true
```

## Muted tests and build problems

(only since TeamCity 2017.2)

List all muted tests and build problems GET <http://teamcity:8111/app/rest/mutes>

Unmute a test or build problems DELETE <http://teamcity:8111/app/rest/mutes/id:XXXX>

Mute a test or build problems POST to <http://teamcity:8111/app/rest/mutes>. Use the same XML or JSON as returned by GET

## Investigations

List investigations in the Root project and its subprojects: <http://teamcity:8111/app/rest/investigations>

Supported locators:

- test: (id:TEST\_NAME\_ID)
- test: (name:FULL\_TEST\_NAME)
- assignee: (<user locator>)
- buildType:(id:XXXX)

Get investigations for a specific test:

```
http://teamcity:8111/app/rest/investigations?locator=test:(id:TEST_NAME_ID)
```

```
http://teamcity:8111/app/rest/investigations?locator=test:(name:FULL_TEST_NAME)
```

Get investigations assigned to a user: [http://teamcity:8111/app/rest/investigations?locator=assignee:\(<user locator>\)](http://teamcity:8111/app/rest/investigations?locator=assignee:(<user locator>))

Get investigations for a build configuration: [http://teamcity:8111/app/rest/investigations?locator=buildType:\(id:XXX\)](http://teamcity:8111/app/rest/investigations?locator=buildType:(id:XXX))

To assign/replace investigations:

POST/PUT to <http://teamcity:8111/app/rest/investigations> (accepts a single investigation) and experimental support for multiple investigations: POST/PUT to <http://teamcity:8111/app/rest/investigations/multiple> (accepts a list of investigations). Use the same XML or JSON as returned by GET.

## Agents

List agents (only authorized agents are included by default): GET <http://teamcity:8111/app/rest/agents>

List all connected authorized agents: GET <http://teamcity:8111/app/rest/agents?locator=connected:true,authorized:true>

List all authorized agents: GET <http://teamcity:8111/app/rest/agents?locator=authorized:true>

List all enabled authorized agents: GET <http://teamcity:8111/app/rest/agents?locator=enabled:true,authorized:true>

List all agents (including unauthorized): GET <http://teamcity:8111/app/rest/agents?locator=authorized:any>

The request uses default filtering (depending on the specified locator dimensions, others can have default implied value). To disable this filtering, add ",defaultFilter:false" to the locator.

Enable/disable an agent: PUT <http://teamcity:8111/app/rest/agents/<agentLocator>/enabled> (put "true" or "false" text as text/plain). See an [example](#).

Authorize/unauthorize an agent: PUT <http://teamcity:8111/app/rest/agents/<agentLocator>/authorized> (put "true" or "false" text as text/plain)

Add a comment when enabling/disabling and authorizing/unauthorizing an agent:

Agent enabled/authorized data is exposed in the `enabledInfo` and `authorizedInfo` nodes:

```

<agent id="1" name="agentName" typeId="1" connected="true" enabled="true" authorized="true"
uptodate="true" ip="....." href="/app/rest/agents/id:1">
<enabledInfo status="true">
<comment>
<user username="userName" id="1" href="/app/rest/users/id:1"/>
<timestamp>20160406T175040+0300</timestamp>
<text>newcomment</text>
</comment>
</enabledInfo>
<authorizedInfo status="true">
<comment>
<user username="userName" id="1" href="/app/rest/users/id:1"/>
<timestamp>20160406T183033+0300</timestamp>
</comment>
</authorizedInfo>
...
</agent>
```

GET and PUT requests are supported to the following URLs:

`http://teamcity:8111/app/rest/agents/<agentLocator>/enabledInfo`  
`http://teamcity:8111/app/rest/agents/<agentLocator>/authorized`

On PUT only status and comment/text sub-items are taken into account:

▼ Example of disabling an agent with a comment: click to expand

```
curl -v -u user:password --request PUT "http://teamcity:8111/app/rest/agents/id:1/enabledInfo" --data
"<enabledInfo status='false'><comment><text>commentText</text></comment></enabledInfo>" --header
"Content-Type:application/xml"
```

Get/PUT agent's single field: GET/PUT `http://teamcity:8111/app/rest/agents/<agentLocator>/<field name>`  
Delete a build agent: DELETE `http://teamcity:8111/app/rest/agents/<agentLocator>`

## Agent Pools

Get/modify/remove agent pools:

GET/PUT/DELETE `http://teamcity:8111/app/rest/projects/XXX/agentPools/ID`

Add an agent pool:

POST the agentPool name='PoolName' element to `http://teamcity:8111/app/rest/projects/XXX/agentPools`

Move an agent to the pool from the previous pool:

POST <agent id='YYY' /> to the pool's agents `http://teamcity.url/app/rest/agentPools/id:XXX/agents`

Example:

```
curl -v -u user:password http://teamcity.url/app/rest/agentPools/id:XXX/agents --request POST --header
"Content-Type:application/xml" --data "<agent id='1' />"
```

## Assigning Projects to Agent Pools

Add a project to a pool:

POST the <project> node to `http://teamcity.url/app/rest/agentPools/id:XXX/projects`

Delete a project from a pool:

DELETE `http://teamcity.url/app/rest/agentPools/id:XXX/projects/id:YYY`

## Users

List of users: GET `http://teamcity:8111/app/rest/users`

Get specific user details: GET `http://teamcity:8111/app/rest/users/<userLocator>`

Create a user: POST `http://teamcity:8111/app/rest/users`

Update/remove specific user: PUT/DELETE `http://teamcity:8111/app/rest/users/`

For the `POST` and `PUT` requests for a user, post data in the form retrieved by the corresponding GET request. Only the following attributes/elements are supported: name, username, email, password, roles, groups, properties.

Work with user roles: <http://teamcity:8111/app/rest/users/<userLocator>/roles>

`<userLocator>` can be of a form:

- `id:<internal user id>` - to reference the user by internal ID
- `username:<user's username>` - to reference the user by username/login name

User's single field: GET/PUT <http://teamcity:8111/app/rest/users/<userLocator>/<field name>>

User's single property: GET/DELETE/PUT <http://teamcity:8111/app/rest/users/<userLocator>/properties/<property name>>

## User Groups

List of groups: GET <http://teamcity:8111/app/rest/userGroups>

List of users within a group: GET [http://teamcity:8111/app/rest/userGroups/key:<Group\\_Key>](http://teamcity:8111/app/rest/userGroups/key:<Group_Key>)

Create a group: POST <http://teamcity:8111/app/rest/userGroups>

Delete a group: DELETE [http://teamcity:8111/app/rest/userGroups/key:<Group\\_Key>](http://teamcity:8111/app/rest/userGroups/key:<Group_Key>)

## Other

### Data Backup

Start backup: POST <http://teamcity:8111/app/rest/server/backup?includeConfigs=true&includeDatabase=true&includeBuildLogs=true&fileName=<fileName>>

where `<fileName>` is the prefix of the file to save backup to. The file will be created in the default backup directory (see [more](#)).

Get current backup status (idle/running): GET <http://teamcity:8111/app/rest/server/backup>

## Typed Parameters Specification

List [typed parameters](#):

- for a project: <http://teamcity:8111/app/rest/projects/<locator>/parameters>
- for a build configuration: <http://teamcity:8111/app/rest/buildTypes/<locator>/parameters>  
The information returned is: parameters count, property name, value, and type. The `rawValue` of the type element is the [parameter specification](#) as defined in the UI.

Get details of a specific parameter:

GET to <http://teamcity:8111/app/rest/buildTypes/<locator>/parameters/<name>>. Accepts/returns plain-text, XML, JSON. Supply the relevant Content-Type header to the request.

Create a new parameter:

POST the same XML or JSON or just plain-text as returned by GET to <http://teamcity:8111/app/rest/buildTypes/<locator>/parameters/>. Note that [secure parameters](#), i.e. `type=password`, are listed, but the values not included into response, so the result should be amended before POSTing back.

Since TeamCity 9.1, partial updates of a parameter are possible (currently in an experimental state):

- `name`: PUT the same XML or JSON as returned by GET to <http://teamcity:8111/app/rest/buildTypes/<locator>/parameters/<NAME>>
- `type`: GET/PUT accepting XML and JSON as returned by GET to the URL <http://teamcity:8111/app/rest/buildTypes/<locator>/parameters/<NAME>/type>
- `type's rawValue`: GET/PUT accepting plain text <http://teamcity:8111/app/rest/buildTypes/<locator>/parameters/<NAME>/type/rawValue>

## Build Status Icon

Icon that represents a build status:

An .svg icon (recommended): GET <http://teamcity:8111/app/rest/builds/<buildLocator>/statusIcon.svg>

A .png icon: GET <http://teamcity:8111/app/rest/builds/<buildLocator>/statusIcon>

Icon that represents build status for several builds (since TeamCity 10.0):

GET request and "strob" build locator dimension:

### Example requests:

For project with id "PROJECT\_ID":

GET

`http://teamcity:8111/app/rest/builds/aggregated/strob:(buildType:(project:(id:PROJECT_ID)))/statusIcon.svg`

For all active branches in a build configuration with id "BUILD\_CONF\_ID":

GET

`http://teamcity:8111/app/rest/builds/aggregated/strob:(branch:(buildType:(id:BUILD_CONF_ID),policy:active_history_and_active_vcs_branches),locator:(buildType:(id:BUILD_CONF_ID)))/statusIcon.svg`

For request /app/rest/builds/aggregated/<build locator> the status is calculated by list of the builds:  
`app/rest/builds?locator=<build locator>`

This allows embedding a build status icon into any HTML page with a simple `img` tag:

For build configuration with internal id "btXXX":

Status of the last build: `

Status of the last build tagged with tag "myTag": `



All other `<buildLocator>` options are supported.

e.g. you can use the following markdown markup to add the build status for GitHub repository for the build configuration with id "TeamCityPluginsByJetBrains\_TeamcityGoogleTagManagerPlugin\_Build" and server <https://teamcity.jetbrains.com> with guest authentication enabled:

```
[![Build  
status](https://teamcity.jetbrains.com/guestAuth/app/rest/builds/buildType:(id:TeamCityPluginsByJet  
Brains_TeamcityGoogleTagManagerPlugin_Build)/statusIcon.svg)](https://teamcity.jetbrains.com/view  
Type.html?buildTypeId=TeamCityPluginsByJetBrains_TeamcityGoogleTagManagerPlugin_Build)
```

If the returned image contains "no permission to get data" text (), ensure that one of the following is true:

- the server has the `guest` user access enabled and the guest user has permissions to access the build configuration referenced, or
- the build configuration referenced has the "enable status widget" `option` ON
- you are logged in to the TeamCity server in the same browser and you have permissions to view the build configuration referenced. Note that this will not help for embedded images in GitHub pages as GitHub retrieves the images from the server-side.

## TeamCity Licensing Information Requests

Since TeamCity 10:

Licensing information: GET <http://teamcity:8111/app/rest/server/licensingData>

List of license keys: GET <http://teamcity:8111/app/rest/server/licensingData/licenseKeys>

License key details: GET [http://teamcity:8111/app/rest/server/licensingData/licenseKeys/<license\\_key>](http://teamcity:8111/app/rest/server/licensingData/licenseKeys/<license_key>)

```
Add license key(s): POST text/plain newline-delimited keys to http://teamcity:8111/app/rest/server/licensingData/licenseKeys  
Delete a license key: DELETE http://teamcity:8111/app/rest/server/licensingData/licenseKeys/<license_key>
```

## CCTray

CCTray-compatible XML is available via <http://teamcity:8111/app/rest/cctray/projects.xml>.

Without authentication (only build configurations available for guest user): <http://teamcity:8111/guestAuth/app/rest/cctray/projects.xml> .

The CCTray-format XML does not include paused build configurations by default. The URL accepts "locator" parameter instead with standard [build configuration locator](#).

## Request Examples

### Request Sending Tool

You can use [curl](#) command line tool to interact with the TeamCity REST API.

Example command:

```
curl -v --basic --user USERNAME:PASSWORD --request POST "http://teamcity:8111/app/rest/users/" --data  
@data.xml --header "Content-Type: application/xml"
```

Where USERNAME, PASSWORD, "teamcity:8111" are to be substituted with real values and data.xml file contains the data to send to the server.

### Creating a new project

Using curl tool

```
curl -v -u USER:PASSWORD http://teamcity:8111/app/rest/projects --header "Content-Type: application/xml"  
--data-binary  
"<newProjectDescription name='New Project Name' id='newProjectId'><parentProject  
locator='id:project1'></newProjectDescription>"
```

### Making user a system administrator

1. Get [super user token](#)

3. Issue the request

Get [curl](#) command line tool and use a command line:

```
curl -v -u :SUPERUSER_TOKEN --request PUT http://teamcity:8111/app/rest/users/username:USERNAM  
E/roles/SYSTEM_ADMIN/g/
```

where

"SUPERUSER\_TOKEN" - the super user token unique for each server start

"teamcity:8111" - the TeamCity server URL

"USERNAME" - the username of the user to be made the system administrator



More examples (for TeamCity 8.0) are available in [this external posting](#).

## Including Third-Party Reports in the Build Results

If your reporting tool produces reports in HTML format, you can extend TeamCity with a custom tab to show the information provided by the third-party reporting tool.

The report provided by your tool can be then displayed either on the build results page, or on the project home page.

The general flow is as follows:

- configure the build script to produce the HTML report (preferably in a zip archive);
- configure publishing the report as the [build artifact](#) to the server: at this point you can check that the archive is

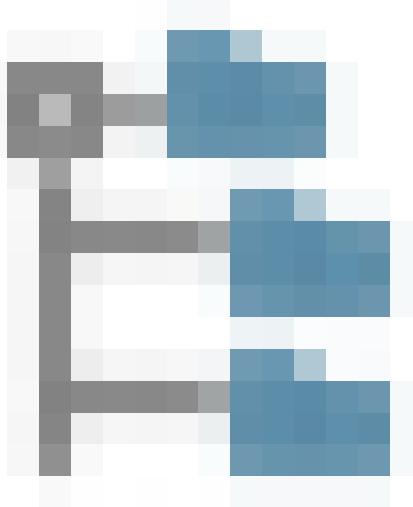
- available in the build artifacts;
- configure the Report Tab to make the report available as an extra tab on the build or project level (see below)

Report tabs support project hierarchy. There are two types of tabs available:

- Build-level: appears on the [build results](#) page for each build that produced an artifact with the specified name. These report tabs are defined in a project and are inherited in its subprojects.  
To override an inherited Report tab in a subproject, create a new report tab with the same name as the inherited one in the subproject.
- Project-level: appears on the Project home page for a particular project only if a build within the project produces the specified reports artifact.

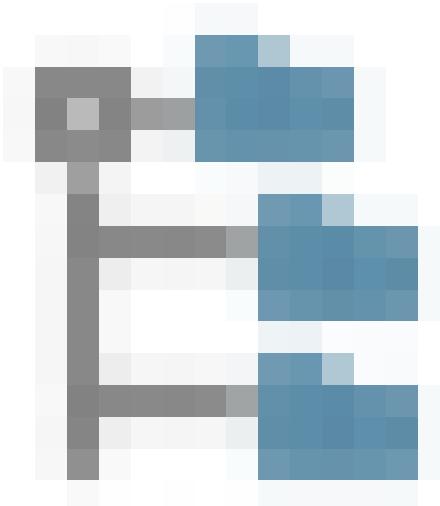
To configure a report tab, go to the Project Settings | Report Tabs and select the type of report tab you want to add.

For a project report tab, specify the following:

Option	Description
Tab Title	Specify a unique title of the report tab that will be displayed in the web UI.
Get artifacts from	Select the build configuration and specify the build whose artifacts will be shown on the tab. Select whether the report should be taken from the last successful, pinned, finished build or the build with the specified build number or the last build with the specified tag.
Start page	<p>Specify the path to the artifacts to be displayed as the contents of the report page. The path must be relative to the root of the build artifact directory.</p> <p>To use a file from an archive, use the <a href="#">path-to-archive!relative-path syntax</a>, e.g. <code>javadoc.zip!/index.html</code>. See the list of <a href="#">supported archives</a>.</p>  <p>You can use the file browser next to the field to select artifacts. <a href="#">Parameter references</a> are supported here, e.g. <code>%parameter%.zip!/index.htm</code></p>

For a build report tab, specify the following:

Option	Description
Tab Title	Specify a unique title of the report tab that will be displayed in the web UI.

Start page	<p>Specify the path to the artifacts to be displayed as the contents of the report page. The path must be relative to the root of the build artifact directory.</p> <p>To use a file from an archive, use the <a href="#">path-to-archive!relative-path syntax</a>, e.g. <code>javadoc.zip!/index.html</code>. See the list of <a href="#">supported archives</a>.</p>  <p>You can use the file browser next to the field to select artifacts. <a href="#">Parameter references</a> are supported here, e.g. <code>%parameter%.zip!/index.htm</code></p>
------------	--

## Custom Chart

In addition to statistic charts generated automatically by TeamCity on the Statistics tab, it is possible to configure your own statistical charts based on the set of [statistic values provided by TeamCity](#) or values reported from a build script. In the latter case you will need to configure your build script to report custom statistical data to TeamCity.

You can view statistic values reported by the build on the [Build parameters](#) page.

 The information in this section refers to TeamCity 10+. For other versions, refer to the [listing](#) to choose the corresponding documentation.

On this page:

- [Managing Custom Charts via the TeamCity Web UI](#)
  - [Adding Custom Charts](#)
  - [Modifying Custom Charts](#)
  - [Reordering Custom Charts](#)
- [Managing Custom Charts Manually](#)
  - [Displaying Custom Chart in TeamCity Web UI](#)
    - [Parameters Reference](#)
    - [Chart Dimensions](#)
    - [Chart Axis Settings](#)
    - [Default Statistics Values Provided by TeamCity](#)
    - [Custom Build Metrics](#)

## Managing Custom Charts via the TeamCity Web UI

It is possible to manage custom charts using the TeamCity Web UI.

### Adding Custom Charts

- The Statistics tab for a project or build configuration provides an option to create a new chart. Note that only one build configuration can be currently added as the data source. More configurations can be added manually.
- On the Parameters tab of the [build results](#) page, the list of Reported statistic values provides checkboxes to select the statistics type for a new [project- or build-configuration-level](#) chart.
  - A project-level chart will be added to the selected target project. The [root project](#) cannot be selected as the target.
  - A build-configuration-level chart will be added to all build configurations of the selected target project and its subprojects. Specifying the [root project](#) as the target will add the chart to all build configurations available on the server.

### Modifying Custom Charts

Use the pencil  icon to edit or delete a custom chart. Note that the Add Statistic Values drop-down displays all statistic values registered on the server. If you select a value non-existent in the current build configuration or project when editing a chart, the chart will not be saved.

Using the cog  icon, you can also configure the Y-axis settings and save them as defaults for all users.

 There is a number of [limitations](#) to editing charts from the TeamCity UI.

### Reordering Custom Charts

To reorder custom charts for a project/build configuration, click the Reorder button and drag-and-drop the charts to arrange them as required and apply your changes.

## Managing Custom Charts Manually

 Since TeamCity 10, manual editing of custom charts has changed. For earlier versions, see this page in the corresponding [documentation](#).

To manually create custom charts to be displayed in the TeamCity web UI, configure the `<TeamCity Data Directory>/config/projects/<ProjectID>/project-config.xml` file. The file has the `<project-extensions>` element which contains all project features, including custom charts. For each chart an `<extention>` element is added.

### Displaying Custom Chart in TeamCity Web UI

To make TeamCity display a custom chart in the web UI, update the `<TeamCity Data Directory>/config/projects/<ProjectID>/project-config.xml` configuration file adding a new `<extention>` sub-element to the `<project-extensions>` element.

Each extension must have a unique `id` in the project.

The `type` attribute is set to

- `<project-graphs>` for Project-level chart
- `<buildType-graphs>` for Build Configuration-level chart.

Each chart is described by the `<parameters>` element. It must contain the `<param>` sub-elements with data shown in the chart in name/value pairs; the "series" parameter uses the JSON format to list series of data shown on the chart.

See the example below:

Custom build configuration-level chart in `project-config.xml`

```

<project-extensions>
  <extension id="customChart1" type="buildtype-graphs">
    <parameters>
      <param name="title" value="Custom chart"/>
      <param name="hideFilters" value="showFailed"/>
      <param name="seriesTitle" value="Some key"/>
      <param name="format" value="duration"/>
      <param name="series"><![CDATA[[
      {
        "type": "valueType",
        "key": "BuildDuration",
        "title": "duration1",
        "sourceBuildTypeId": "my_first_configuration_id"
      }, {
        "type": "valueType",
        "key": "customKey",
        "title": "Custom data",
        "color": "#ee0055 "
      }, {
        "type": "valueTypes",
        "pattern": "buildStageDuration:*",
        "title": "Stage: {1}"
      }
    ]]>
      </param>
      <param name="properties.width" value="300"/>
      <param name="properties.height" value="300"/>
      <param name="properties.axis.y.type" value="logarithmic"/>
      <param name="properties.axis.y.includeZero" value="false"/>
      <param name="properties.axis.y.max" value="10000"/>
    </parameters>
  </extension>
  <extension id="secondChart" type="buildtype-graphs">
    <parameters>
      <param name="title" value="empty"/>
    </parameters>
  </extension>
</project-extensions>

```

This chart will be shown on Statistics tabs of the Build Configurations of the project where the `project-config.xml` file is located and all its subprojects. To display a chart for all Build Configurations, add it to the `project-config.xml` of the [Root Project](#).

## Parameters Reference

Name	Description
<code>title</code>	The title above the chart.
<code>seriesTitle</code>	The title above the list of series used on the chart (in the singular form). The default is "Serie".
<code>defaultFilters</code>	The list of comma-separated options to be checked by default. Can include the following: <ul style="list-style-type: none"> <li>• <code>showFailed</code> — include results from failed builds by default.</li> <li>• <code>averaged</code> — by default, show averaged values on the chart.</li> </ul>

hideFilters	The list of comma-separated filter names that will not be shown next to the chart: <ul style="list-style-type: none"> <li>• all — hide all filters.</li> <li>• series — hide series filter (you won't be able to show only data from specific valueType specified for the chart.)</li> <li>• range — hide the date range filter.</li> <li>• showFailed — hide the checkbox which allows including data for failed builds.</li> <li>• averaged — hide the checkbox which allows viewing averaged values.</li> <li>• Defaults — empty (all filters are shown).</li> </ul>
format	The format of the y-axis values. Supported formats are: <ul style="list-style-type: none"> <li>• duration, data should be in milliseconds;</li> <li>• percent, data should be in percents (from 0 to 100);</li> <li>• percentby1, the format will show data between 0 and 1 as percents (from 0 to 100);</li> <li>• size, data should be in bytes.</li> </ul> If no format is specified, the numeric format is used.

The `<series>` parameter uses JSON format to list series of data shown on the chart. Each series is drawn in a separate color and you can choose one or another series using a filter.

Name	Description
type	<ul style="list-style-type: none"> <li>• valueType describes a series of data shown on the chart. Each series is drawn with a separate color and you may choose one or another series using a filter.</li> <li>• valueTypes allows displaying several series on the chart by a pattern (described <a href="#">below</a>)</li> </ul>
key	The name of the valueType (series). It can be predefined by TeamCity, like <code>BuildDuration</code> or <code>ArtifactsSize</code> (see below <a href="#">Default Statistics Values Provided by TeamCity</a> for the complete list of predefined statistic values), or you can provide your own data by reporting it from the build script.
title	The series name shown in the series selector. Defaults to <code>&lt;key&gt;</code> . For several series, pattern group markers can be used: <code>{1}</code> stands for the first captured group in the pattern, <code>{0}</code> stands for the whole pattern.
sourceBuildTypeId	This field allows you to explicitly specify a build configuration to use the data from for the given series. This field is mandatory for the first valueType used in a chart if the chart is added at the project level. In other cases it is optional. However, note that TeamCity chooses the build configuration to take the data from according to the following rules: <ol style="list-style-type: none"> <li>1. if the <code>sourceBuildTypeId</code> is set within the <code>valueType</code>, the data is taken from this build configuration even if it belongs to a different project.</li> <li>2. if the <code>sourceBuildTypeId</code> is not set within current the <code>valueType</code>, but it is set in the <code>valueType</code> above the current one within the chart, the data from the build configuration referenced above will be taken. See example for the <code>plugin-settings.xml</code> file above.</li> <li>3. if the <code>sourceBuildTypeId</code> is not set within current the <code>valueType</code> and is not set above, the chart will show data for the current build configuration, i.e. this chart will work only for build configurations.</li> </ol>
color	The color of a series to be used in the chart. Standard web color formats can be used - "#RRGGBB", color names, etc. For more information see <a href="#">HTML Colors reference</a> and <a href="#">HTML Color Names reference</a> . If not specified, an automatic color will be assigned based on the series title.
pattern	Pattern for names of the Value Types (or series) to be shown on the chart. The asterisk (*) sign is allowed to filter Value Types (or series) either predefined by TeamCity, like <code>BuildDuration</code> or <code>ArtifactsSize</code> (see below <a href="#">Default Statistics Values Provided by TeamCity</a> for the complete list of predefined statistic values), or your own data can be provided by reporting it from the build script.

## Chart Dimensions

You can set the custom chart width/height in pixels using the `properties.width` and `properties.height` attributes for the `param` elements, e.g. `<param name="properties.width" value="300"/>`.

## Chart Axis Settings

You can also customize the default axis settings for a chart via parameter names starting with `properties`, e.g. `properties.axis.y.type`

Supported properties:

Name	Description
<code>properties.axis.y.type</code>	<ul style="list-style-type: none"> <li>• <code>linear</code> (default) for the standard scale.</li> <li>• <code>logarithmic</code> for the logarithmic Y axis scale</li> </ul>
<code>properties.axis.y.includeZero</code>	Whether the zero value is included on the Y axis: <ul style="list-style-type: none"> <li>• <code>true</code> (default)</li> <li>• <code>false</code> (zero is not included)</li> </ul>
<code>properties.axis.y.min</code>	An integer value to start the Y axis from.
<code>properties.axis.y.max</code>	An integer value to use as the maximum for the Y axis value .

## Default Statistics Values Provided by TeamCity

The table below lists the predefined value providers that can be used to configure a custom chart. The values reported for each build differ depending on your build configuration settings.

You can view the all statistic values reported by the build on the Build Results | Parameters | Reported statistic values tab. For each of the values, a statistics chart is available on clicking the View Trend icon .

Key	Description	Unit
<code>ArtifactsSize</code>	The sum of all <code>artifact</code> file sizes in the artifact directory	Bytes
<code>VisibleArtifactsSize</code>	The sum of all <code>artifact</code> file sizes excluding hidden artifacts (those placed under <code>.teamcity</code> directory)	Bytes
<code>buildStageDuration:artifactsPublishing</code>	The duration of the artifact publishing step in the build	Milliseconds
<code>buildStageDuration:sourcesUpdate</code>	The duration of the source checkout step	Milliseconds
<code>buildStageDuration:dependenciesResolving</code>	The duration of resolving dependencies of the build	Milliseconds
<code>BuildDuration</code>	The build duration (all build stages)	Milliseconds
<code>BuildDurationNetTime</code>	The build steps duration (excluding the checkout and artifact publishing time, etc.)	Milliseconds
<code>CodeCoverageB</code>	Block-level code coverage	%
<code>CodeCoverageC</code>	Class-level code coverage	%
<code>CodeCoverageL</code>	Line-level code coverage	%
<code>CodeCoverageM</code>	Method-level code coverage	%
<code>CodeCoverageR</code>	Branch Coverage	%
<code>CodeCoverageS</code>	Statement Coverage	%
<code>CodeCoverageAbsBCovered</code>	The number of covered blocks	int
<code>CodeCoverageAbsBTotal</code>	The total number of blocks	int
<code>CodeCoverageAbsCCovered</code>	The number of covered classes	int
<code>CodeCoverageAbsCTotal</code>	The total number of classes	int
<code>CodeCoverageAbsLCovered</code>	The number of covered lines	int
<code>CodeCoverageAbsLTotal</code>	The total number of lines	int

CodeCoverageAbsMCovered	The number of covered methods	int
CodeCoverageAbsMTotal	The total number of methods	int
CodeCoverageAbsRCovered	The number of covered branches	int
CodeCoverageAbsRTotal	The total number of branches	int
CodeCoverageAbsSCovered	The number of covered statements	int
CodeCoverageAbsSTotal	The total number of statements	int
DuplicatorStats	The number of code duplicates found	int
TotalTestCount	The total number of tests in the build	int
PassedTestCount	The number of successfully passed tests in the build	int
FailedTestCount	The number of failed tests in the build	int
IgnoredTestCount	The number of ignored tests in the build	int
InspectionStatsE	The number of inspection errors in the build	int
InspectionStatsW	The number of inspection warnings in the build	int
SuccessRate	An indicator whether the build was successful	0 - failed, 1 - successful
TimeSpentInQueue	How long the build was queued	Milliseconds

## Custom Build Metrics

If the predefined build metrics do not cover your needs, you can report custom metrics to TeamCity from your build script and use them to create a custom chart. There are two ways to report custom metrics to TeamCity:

- using [service messages](#) from your build,
- or (obsolete approach) using the [teamcity-info.xml](#) file.

Note that custom value keys should be unique and should not interfere with value keys predefined by TeamCity.

See also:

[Concepts: Code Coverage | Code Inspection | Code Duplicates](#)  
[User's Guide: Statistic Charts](#)  
[Extending TeamCity: Build Script Interaction with TeamCity | Custom Statistics](#)

## Edit Custom Chart Limitations

Currently editing a custom chart from the TeamCity UI is limited:

- Only one source Build Configuration could be assigned to a chart
- It is impossible to move a chart to another project
- A chart cannot be created if it contains no data
- A statistic value can be added only if it contains some data
- The width and height of a chart are not configurable
- A patterned statistic value cannot be added or removed.

## Developing TeamCity Plugins



TeamCity is hiring! Learn about [the available vacancies](#) on the JetBrains site. [Read](#) about working in the TeamCity team.

TeamCity functionality can be significantly extended by custom plugins. TeamCity plugins are written in Java (any JVM language with Java invulnerability like Kotlin or Groovy can be used), run within the TeamCity application and have access to internal entities of the TeamCity server or agent.

Aside from this documentation, please refer to the following sources:

- [Open API Javadoc](#)
- bundled [sample plugin](#)
- [list of existing plugins](#) and [bundled open-source plugins](#)

If you need more information or have a question regarding the API, please do not hesitate to post your question into [TeamCity Plugins forum](#). Please use the search before posting to avoid possible duplication of discussions.

Consider making your plugin public and submit it to be listed on the [plugins page](#).

Please refer to corresponding section for further details.

- [Getting Started with Plugin Development](#)
- [Typical Plugins](#)
  - [Build Runner Plugin](#)
  - [Risk Tests Reordering in Custom Test Runner](#)
  - [Custom Build Trigger](#)
  - [Extending Notification Templates Model](#)
  - [Issue Tracker Integration Plugin](#)
  - [Version Control System Plugin](#)
  - [Version Control System Plugin \(old style - prior to 4.5\)](#)
  - [Custom Authentication Module](#)
  - [Custom Notifier](#)
  - [Custom Statistics](#)
  - [Custom Server Health Report](#)
  - [Extending Highlighting for Web diff view](#)
  - [External Storage Implementation Guide](#)
- [Bundled Development Package](#)
- [Open API Changes](#)
- [Plugin Types in TeamCity](#)
- [Plugins Packaging](#)
- [Server-side Object Model](#)
- [Agent-side Object Model](#)
- [Extensions](#)
- [Web UI Extensions](#)
- [Plugin Settings](#)
- [Development Environment](#)
- [Developing Plugins Using Maven](#)
- [Plugin Development FAQ](#)

## Plugin Quick Start

See [Getting Started with Plugin Development](#) to create your first plugin with Maven. [Developing Plugins Using Maven](#) provides more details.

There are also several approaches to create plugins provided by third parties or existing out of the main TeamCity development line.

- [Developing Plugins Using Maven](#) - Maven archetype supported by JetBrains
- [Gradle TeamCity plugin](#) - Git, Gradle build. Supports agent and server-side plugins, and helpers to download, install a TeamCity server, tasks to deploy, start and stop the server and agent.
- [template plugin 1](#), see also a [blog post](#) - Git, IDEA project
- [template plugin 2](#) - Subversion, IDEA project and Ant build, generates a plugin with custom name, see details in the `readme.txt` of the checkout
- [Gradle TeamCity plugin](#) - earlier, but a bit outdated version of Gradle build for TeamCity plugins
- (obsolete) [Maven Archetype for TeamCity server plugin](#)

See also a [post](#) on the very first steps for setting up the plugin development environment.

## Getting Started with Plugin Development

 TeamCity is hiring! Learn about [the available vacancies](#) on the JetBrains site. Read about working in the TeamCity team.

The use of plugins allows you to extend the TeamCity functionality. See the [list of existing TeamCity plugins](#) created by JetBrains developers and community.

This document provides information on how to develop and publish a server-side plugin for TeamCity [using Maven](#). The plugin will return the "Hello World" jsp page when using a specific URL to the TeamCity Web UI.

On this page:

- [Introduction](#)
- [Step 1. Set up the environment](#)
- [Step 2. Generate a Maven project](#)
  - [View the project structure](#)
- [Step 3. Edit the plugin descriptor](#)
- [Step 4. Create the plugin sources](#)
  - [A. Create the plugin web-resources](#)
  - [B. Create the controller and obtain the path to the JSP](#)
  - [C. Update the Spring bean definition](#)
- [Step 5. Build your project with Maven](#)
- [Step 6. Install the plugin to TeamCity](#)
- [Next Steps](#)

### Introduction

A plugin in TeamCity is a `zip` archive containing a number of classes packed into a JAR file and [plugin descriptor](#) file. The TeamCity Open API can be found in the JetBrains [Maven repository](#). The Javadoc reference for the API is available [online](#) and locally in `<TeamCity Home Directory>/devPackage/javadoc/openApi-help.jar`, after you install TeamCity.

### Step 1. Set up the environment

To get started writing a plugin for TeamCity, set up the plugin development environment.

1. Download and install [Oracle Java](#). Set the `Java_Home` environment variable on your system. Java 1.8 is required since TeamCity 10, the 32-bit version is recommended, the 64-bit version [can be used](#).
2. Download and install [TeamCity](#) on your development machine. Since you are going to use this machine to test your plugin, it is recommended that this TeamCity server is of the same version as your production server. We are using TeamCity 9.0.2 installed on Windows in our setup.
3. Download and install a Java IDE; we are using [IntelliJ IDEA Community Edition](#), which has a built-in Maven integration.
4. Download and install [Apache Maven](#). Maven 3.2.x is recommended. Set the `M2_HOME` environment variable. Run `mvn -version` to verify your setup. We are using Maven 3.2.5. in our setup.

### Step 2. Generate a Maven project

We'll generate a Maven project [from an archetype](#) residing in JetBrains Maven repository. Executing the following command will produce a project for a server-side-only plugin.

You will be asked to enter the Maven `groupId`, `artifactId`, `version`, package name and `teamcityVersion` for your plugin.

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
-DarchetypeRepository=http://download.jetbrains.com/teamcity-repository
-DarchetypeArtifactId=teamcity-server-plugin -DarchetypeGroupId=org.jetbrains.teamcity.archetypes
-DarchetypeVersion=RELEASE
```

We used the following values:

groupId	com.demoDomain.teamcity.demoPlugin
artifactId	demoPlugin
version	leave the default 1.0-SNAPSHOT
packaging	leave the default package name
teamcityVersion	10.0.

i Different released versions of the TeamCity server API are listed [here](#).

demoPlugin will be used as the internal name of our plugin.

When the build finishes, you'll see that the demoPlugin directory was created in the directory where Maven was called.

## View the project structure

The root of the demoPlugin directory contains the following:

- the `readme.txt` file with minimal instructions to develop a server-side plugin
- the `pom.xml` file which is your Project Object Model
- the `teamcity-plugin.xml` file which is your **plugin descriptor** containing meta information about the plugin.
- the `demoPlugin-server` directory contains the plugin sources:
  - `\src\main\java\zip` contains the `AppServer.java` file
  - `src\main\resources` includes resources controlling the plugin look and feel.
  - `src\main\resources\META-INF` folder contains `build-server-plugin-demo-plugin.xml`, the bean definition file for our plugin. TeamCity plugins are initialized in their own Spring containers and every plugin needs a Spring bean definition file describing the main services of the plugin.
- the `build` directory contains the xml files which define how the project output is aggregated into a single distributable archive.

## Step 3. Edit the plugin descriptor

Open the `teamcity-plugin.xml` file in the project root folder with IntelliJ IDEA and add details, such as the plugin display name, description, vendor, and etc. by modifying the **corresponding attributes** in the file.

## Step 4. Create the plugin sources

Open the `pom.xml` from the project root folder with IntelliJ IDEA.

We are going to make a controller class which will return `Hello.jsp` via a specific TeamCity URL.

### A. Create the plugin web-resources

The plugin web resources (files that are accessed via hyperlinks and JSP pages) are to be placed into the `buildServerResources` subfolder of the plugin's resources.

1. First we'll create the directory for our jsp: go to the `demoPlugin-server\src\main\resources` directory in IDEA and create the `buildServerResources` directory.
2. In the newly created `demoPlugin-server\src\main\resources\buildServerResources` directory, create the `Hello.jsp` file, e.g.

```

<html>
<body>
Hello world
</body>
</html>

```

## B. Create the controller and obtain the path to the JSP

Go to `\demoPlugin\demoPlugin-server\src\main\java\com\demoDomain\teamcity\demoPlugin` and open the `AppServer.java` file to create a custom controller:

1. We'll create a simple controller which extends the TeamCity `jetbrains.buildServer.controllers.BaseController` class and implements the `BaseController.doHandle(HttpServletRequest, HttpServletResponse)` method.
2. The TeamCity open API provides the `jetbrains.buildServer.web.openapi.WebControllerManager` which allows registering custom controllers using the path to them: the path is a part of URL starting with a slash / appended to the URL of the server root.
3. Next we need to construct the path to our JSP file. When a plugin is unpacked on the TeamCity server, the paths to its resources change. To obtain valid paths to the files after the plugin is installed, use the `jetbrains.buildServer.web.openapi.PluginDescriptor` class which implements the `getPluginResourcesPath` method; otherwise TeamCity might have difficulties finding the plugin resources.

```

package com.demoDomain.teamcity.demoPlugin;

import jetbrains.buildServer.controllers.BaseController;
import jetbrains.buildServer.web.openapi.PluginDescriptor;
import jetbrains.buildServer.web.openapi.WebControllerManager;
import org.jetbrains.annotations.Nullable;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class AppServer extends BaseController {
    private PluginDescriptor myDescriptor;

    public AppServer (WebControllerManager manager, PluginDescriptor descriptor) {
        manager.registerController("/demoPlugin.html",this);
        myDescriptor=descriptor;
    }

    @Nullable
    @Override
    protected ModelAndView doHandle(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse) throws Exception {
        return new ModelAndView(myDescriptor.getPluginResourcesPath("Hello.jsp"));
    }
}

```

## C. Update the Spring bean definition

Go to the `demoPlugin-server\src\main\resources\META-INF` directory and update `build-server-plugin-demo-plugin.xml` to include our `AppServer` class.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-autowire="constructor">
    <bean class="com.demoDomain.teamcity.demoPlugin.AppServer"></bean>
</beans>

```

## Step 5. Build your project with Maven

Go to the root directory of your project and run

```
mvn package
```

The target directory of the project root will contain the `<demoPlugin>.zip` file. It is our plugin package, ready to be installed.

## Step 6. Install the plugin to TeamCity

1. Copy the plugin zip to `<TeamCity Data Directory>/plugins` directory.
2. Restart the server and locate the TeamCity Demo Plugin in the Administration|Plugins List to verify the plugin was installed correctly.



The Hello World page is available via `<TeamCity server URL>/demoPlugin.html`.

## Next Steps

[Read more](#) if you want to extend the TeamCity pages with custom elements.  
The detailed information on TeamCity plugin development is available [here](#).

You can also use the [TeamCity SDK Maven plugin](#) allowing you to control a TeamCity instance from the command line and to install a new/updated plugin created from a Maven archetype.

## Typical Plugins

TeamCity is hiring! Learn about [the available vacancies](#) on the JetBrains site. [Read](#) about working in the TeamCity team.

This section covers:

- [Build Runner Plugin](#)
- [Risk Tests Reordering in Custom Test Runner](#)
- [Custom Build Trigger](#)
- [Extending Notification Templates Model](#)
- [Issue Tracker Integration Plugin](#)
- [Version Control System Plugin](#)
- [Version Control System Plugin \(old style - prior to 4.5\)](#)
- [Custom Authentication Module](#)
- [Custom Notifier](#)
- [Custom Statistics](#)
- [Custom Server Health Report](#)
- [Extending Highlighting for Web diff view](#)
- [External Storage Implementation Guide](#)

### Build Runner Plugin

A [build runner](#) plugin consists of two parts: agent-side and server-side. The server side part of the plugin provides meta information about the build runner, the web UI for the build runner settings and the build runner properties validator. The

agent-side part launches builds.

On this page:

- Server-side part of the runner
- Agent-side part of the runner
- Extending the Ant runner
- Your Build Runner Results in TeamCity
  - Build log
  - Artifacts
  - Reports
    - XML Report processing
    - HTML Report processing

A build runner can have various settings which must be edited by the user in the web UI and passed to the agent. These settings are called runner parameters (or runner properties) and provided as a Map<String, String> to the agent part of the runner.

 Hint: some build runners whose source code can be used as a reference:

- Rake Runner
  - FxCop runner sources
- Other build runner plugins.

## Server-side part of the runner

The main entry point for the runner on the server side is `jetbrains.buildServer.serverSide.RunType`. A build runner plugin must provide its' own RunType and register it in the `jetbrains.buildServer.serverSide.RunTypeRegistry`.

RunType has a type which must be unique among all build runners and correspond to the type returned by the agent-side part of the runner (see `jetbrains.buildServer.agent.AgentBuildRunnerInfo`).

The `getEditRunnerParamsJspFilePath` and `getViewRunnerParamsJspFilePath` methods return paths to JSP files for editing and viewing runner settings. These JSP files must be bundled with plugin in `buildServerResources` subfolder, [read more](#). The paths should be relative to the `buildServerResources` folder.

 Since TeamCity 5.1, the path to the build runner resources files should be a full path without context. This path could be either a path to a .jsp file or a path that is handled by a controller. The plugin class may use `PluginDescriptor#getPluginResourcesPath()` method to create a path to a .jsp file from the `buildServerResources` folder of the plugin.

 TeamCity 5.0.x and earlier uses the following rule to compute a full path to the runner's jsp:

```
<context path>/plugins/<runType>/<returned jsp path>
```

 Hint: before writing your own JSP for a custom build runner, take a look at the JSP files of the existing runners bundled with TeamCity.

When a user fills in your runner settings and submits the form, `jetbrains.buildServer.serverSide.PropertiesProcessor` returned by the `getRunnerPropertiesProcessor` method will be called. This processor will be able to verify user settings and indicate which of them are invalid.

Usually a JSP page is simple and does not provide much controls except for fields, checkboxes and so on. But if you need more control on how the page is processed on the server side, then you should register your own extension to the runner editing controller: `jetbrains.buildServer.controllers.admin.projects.EditRunTypeControllerExtension`.

And finally if you need to prefill some settings with default values, you can do this with the help of the `getDefaultRunnerProperties` method.

## Agent-side part of the runner

The main interface for agent-side runners is `jetbrains.buildServer.agent.AgentBuildRunner`. However, if your custom runner runs an external process, it is simpler to use the following classes:

1. jetbrains.buildServer.agent.runner.CommandLineBuildServiceFactory
2. jetbrains.buildServer.agent.runner.CommandLineBuildService
3. jetbrains.buildServer.agent.runner.BuildServiceAdapter

You should implement the `CommandLineBuildServiceFactory` factory interface and make your class a Spring bean. The factory also provides some meta information about the runner via `jetbrains.buildServer.agent.AgentBuildRunnerInfo`.

`CommandLineBuildService` is an abstract class which simplifies external processes launching and allows listening for process events (output, finish and so on). Your runner should extend this class. Since TeamCity 6.0, we introduced the `jetbrains.buildServer.agent.runner.BuildServiceAdapter` class that extends `CommandLineBuildService` and provides utility methods to access build and runner context parameters.

`AgentBuildRunnerInfo` has two methods: `getType` which must return the same type as the one returned by the server-side part of the plugin, and `canRun` which is called to determine whether the custom runner can run on the agent (in the agent environment).

If the command line build service is not suitable for your needs, you can still implement the `AgentBuildRunner` interface and define it in the Spring context. Then it will be loaded automatically.

## Extending the Ant runner

The TeamCity Ant runner, while being a plugin itself, can also be extended with the help of `jetbrains.buildServer.agent.ant.AntTaskExtension`. This extension works in the same JVM where Ant is running. Using this extension, you can watch for Ant tasks, modify/patch them and log various messages to the build log.

Your class implementing `AntTaskExtension` interface must be defined in the Spring bean and it will be picked up by the Ant runner automatically. You need to add a dependency to  
`<teamcity>/webapps/ROOT/WEB-INF/plugins/ant/agent/antPlugin.zip!antPlugin/ant-runtime.jar jar`.

## Your Build Runner Results in TeamCity

### Build log

Usually a build runner starts an external process, and logging is performed from that process. The simplest way to log messages in this case is to use service messages, [read more](#). In brief, a service message is a specially formatted text with attributes; when such text is logged to the process output, it is parsed and the associated processing is performed. With the help of these messages you can create a TeamCity hierarchical build log, report tests, errors and so on.

If an external process launched by your runner is Java and you can't use service messages, it is possible to obtain `jetbrains.buildServer.agent.BuildProgressLogger` in the class running in this JVM. For this, the following jar files must be added in the classpath of the external Java process: `runtime-util.jar`, `server-logging.jar`. Then you should use the `jetbrains.buildServer.agent.LoggerFactory` method to construct the logger: `LoggerFactory.createBuildProgressLogger(parentClassLoader)`. Since this way is more involved, it is recommended to use service messages instead.

If logging of the messages is done in the agent JVM (not from within the external process started by your runner), you can obtain `jetbrains.buildServer.agent.BuildProgressLogger` from the `jetbrains.buildServer.agent.AgentRunningBuild#getBuildLogger` method.

### Artifacts

You can instruct your build runner to publish the resulting artifacts to TeamCity using service messages. Note that artifacts are uploaded to the TeamCity server in the background, so to verify that your artifacts are uploaded, you'll have to wait until your build is finished.

### Reports

#### XML Report processing

If your runner reports build results in a format supported by TeamCity, they can be displayed in the TeamCity web UI on the Build Results page. There are two ways to approach this:

- using the [XML Report Processing](#) build feature
- via [service messages](#)

#### HTML Report processing

If your build runner produces some static HTML content, it can be displayed in the TeamCity web UI. Configure a custom [report tab](#) to show the results on a project or build level.

## Risk Tests Reordering in Custom Test Runner

In TeamCity, you can instruct the system to [run risk group tests before any others](#).

To implement the risk group tests reordering feature for your own custom test runner, TeamCity provides the following special system properties:

- `teamcity.tests.runRiskGroupTestsFirst` — this system property value contains groups of tests to run before others. Accordingly, there are two groups: `recentlyFailed` and `newAndModified`. If more than one group is specified, they are separated with a comma. This property is provided only if corresponding settings are selected on the build runner page.
- `teamcity.tests.recentlyFailedTests.file` — this system property value contains the full path to a file with the recently failed tests. The property is provided only if the `recentlyFailed` group is selected. The file contains tests separated by a new line. For Java-like tests, full class names are stored in the file (without the test method name). In other cases, the full name of the test will be stored in the file as it was reported by the tests runner.
- `teamcity.build.changedFiles.file` — this system property is useful if you want to support running of new and modified tests in your tests runner. This property contains the full path to a file with the information about changed files included in the build. You can use this file to determine whether any tests were modified and run them before others. The file contains new-line separated files: each line corresponds to one file and has the following format:

```
<relative file path>:<change type>:<revision>
```

where:

- `<relative file path>` is the path to a file relative to the current checkout directory.
  - `<change type>` is a type of modification and can have the following values: `CHANGED`, `ADDED`, `REMOVED`, `NOT_CHANGED`, `DIRECTORY_CHANGED`, `DIRECTORY_ADDED`, `DIRECTORY_REMOVED`
  - `<revision>` is a file revision in the repository. If the file is a part of change list started via the [remote run](#), then the `<personal>` string will be written instead of the file revision.
- `teamcity.build.checkoutDir` — this system property contains the path to the build checkout directory. It is useful if you need to convert relative paths to modified files to absolute ones.



TeamCity will pass the `teamcity.tests.runRiskGroupTestsFirst`, `teamcity.tests.recentlyFailedTests.file` and `teamcity.build.changedFiles.file` properties to the build process, but if the process starts an additional JVM or other processes, these properties won't be passed to them automatically.

For example, if you are using an Ant runner, you will have access to these properties from the Ant build.xml. But if your build.xml starts a new JVM (or `<junit/>` task with `fork="yes"` attribute), and you want to access these properties from this JVM, you'll have to modify your build script and pass them explicitly.

## Known Limitations

If you have a package specified in the `TestNG` `xml` suite, reordering will not work: in this case `TestNG` itself searches for classes in packages and TeamCity cannot affect the way it sorts these classes. However, reordering will work if you specify concrete Test classes in the `xml` suite. Also, if you have several `xml` suites, reordering will work on the per-suite basis.

Having single classes in the XML suite may impose some inconveniences, e.g. developers have to remember to include classes in the suites. At the same time, this should speedup the tests startup, as the process of the searching classes by package is not that fast.

## Custom Build Trigger

An example of a trigger plugin can be found in [Url Build Trigger](#).

## Build Trigger Service

Build trigger is a service whose purpose is to trigger builds (add builds to the queue). Build trigger must extend `jetbrains.buildServer.buildTriggers.BuildTriggerService` abstract class. Build trigger service is uniquely identified by trigger name (see `getName` method). There is no need to register `BuildTriggerService`, instead plugin should provide a class extending the `jetbrains.buildServer.buildTriggers.BuildTriggerService` defined as a Spring bean.

## Build Trigger Settings

Build trigger settings is an object containing build trigger parameters specified by a user via the web UI. Build trigger settings are represented by `jetbrains.buildServer.buildTriggers.BuildTriggerDescriptor` class. The settings are contained within a map of string parameters. More than one instance of `jetbrains.buildServer.buildTriggers.BuildTriggerDescriptor` corresponding to the same trigger service can be added to the build configuration or a template. Instances of the `jetbrains.buildServer.buildTriggers`.

`BuildTriggerDescriptor` with the same trigger name and parameters are considered equal. With help of `jetbrains.buildServer.buildTriggers.BuildTriggerService#isMultipleTriggersPerBuildTypeAllowed()` trigger service can allow or disallow multiple trigger settings per build configuration.

Each trigger service can provide an URL to a jsp or custom controller which will show the trigger web UI. The approach is similar to the one used for VCS roots and build runners.

## Triggering Policy

Build trigger service must return a policy ( `jetbrains.buildServer.buildTriggers.BuildTriggeringPolicy`) which will be used to add builds to the queue. Currently only one policy is available: `jetbrains.buildServer.buildTriggers.PolledBuildTrigger`. More policies can be added in the future.

Trigger returning `jetbrains.buildServer.buildTriggers.PolledBuildTrigger` policy will be polled by the server with regular intervals. Trigger will receive `jetbrains.buildServer.buildTriggers.PolledTriggerContext` object which contains all information necessary to make a decision whether a build must be triggered or not. Trigger can use `jetbrains.buildServer.serverSide.SBuildType#addToQueue(java.lang.String)` method to add builds to the queue. Note that `jetbrains.buildServer.buildTriggers.PolledTriggerContext` also provides access to the custom data storage. This storage can be used for build trigger state associated with a build configuration and trigger settings. Custom storage will be automatically persisted and restored upon server restart.

## Extending Notification Templates Model

You can extend data model passed into notification templates when evaluating.

In your plugin, implement `jetbrains.buildServer.notification.TemplateProcessor` interface. The following example can be found in our sample plugin:

```
public class SampleTemplateProcessor implements TemplateProcessor {  
    public SampleTemplateProcessor() {}  
  
    @NotNull  
    public Map<String, Object> fillModel(@NotNull NotificationContext context) {  
        Map<String, Object> model = new HashMap<String, Object>();  
        model.put("users", context.getUsers());  
        model.put("event", context.getEventType());  
        return model;  
    }  
}
```

## Issue Tracker Integration Plugin

TeamCity offers integration with several issue trackers and a custom plugin can provide support for other systems.

### Issue tracker integration

To create a TeamCity plugin for custom issue tracking system (ITS), you have to implement the following interfaces (all from `jetbrains.buildServer.issueTracker` package):

- `jetbrains.buildServer.issueTracker.SIssueProvider` - represents a single provider
- `jetbrains.buildServer.issueTracker.IssueProviderFactory` - API for instantiation of issue tracker providers

The main entity is a provider (i.e. connection to the ITS), responsible for parsing, extracting and fetching issues from the ITS.

Here is a brief description of the strategy used in TeamCity in respect to ITS integration:

When the server is going to render the user comment (VCS commit, or build comment), it invokes all registered providers to parse the comment. This operation is performed by the `IssueProvider.getRelatedIssues()` method, which analyzes the comment and returns the list of the issue mentions ( `jetbrains.buildServer.issueTracker.IssueMention`). `IssueMention` just holds the information that is enough to render a popup arrow near the issue id. When the user points the mouse cursor on the arrow, the server requests the full data for this issue calling `IssueProvider.findIssueById()` method, and then displays the data in a popup. The data can be taken from the provider's cache.

The provider has a number of parameters, configured from admin UI. These parameters are passed using the properties map (a map string -> string). Commonly used properties include provider name, credentials to communicate with ITS, or regular expression to parse issue ids. You don't have to worry about storing the properties in XML files, server does that.

Provider registration is done by the TeamCity administrator in the web UI, and the responsibility for it lies mostly on TeamCity server. The plugin must only provide a JSP used for creation/editing of the provider (see details below).

## Plugin development overview

A brief summary of steps to be done to create and add a plugin to TeamCity.

- Implement factory and provider interfaces ( `jetbrains.buildServer.issueTracker.SIssueProvider` and `jetbrains.buildServer.issueTracker.IssueProviderFactory` )
- Create a JSP page for admin UI
- Install the plugin (to `.BuildServer/plugins`)

## Reusing default implementation

Common code of Jira, Bugzilla and YouTrack plugins can be found in `Abstract*` classes in the same package:

- `jetbrains.buildServer.issueTracker.AbstractIssueProviderFactory`
- `jetbrains.buildServer.issueTracker.AbstractIssueProvider`
- `jetbrains.buildServer.issueTracker.AbstractIssueFetcher` - a helper entity which encapsulates fetch-related logic

`AbstractIssueProvider` implements a simple caching provider able to extract the issues from the string based on a regexp. In most cases you just need to derive from it and override few methods. A simple derived provider class can look like this:

```
public class MyIssueProvider extends AbstractIssueProvider {  
    // Let's name the provider simple: "myName". The plugin name should be the same.  
    public MyIssueProvider(@NotNull IssueFetcher fetcher) {  
        super("myName", fetcher);  
    }  
  
    // Means that issues are in format "PREFIX-123", like in Jira or YouTrack.  
    // The prefix is configured via properties, regexp is invisible for users.  
    protected boolean useIdPrefix() {  
        return true;  
    }  
}
```

Providers like Bugzilla might need to override `extractId` method, because the mention of issue id (in comment) and the id itself can differ. For instance, suppose the issues are referenced by a hash with a number, e.g. #1234; the regexp is "#(\d{4})" (configurable); but the issues in ITS are represented as plain integers. Then the provider must extract the substrings matching "#(\d{4})" and return the first groups only. You should implement it in `extractId` method:

```
@NotNull  
protected String extractId(@NotNull String match) {  
    Matcher matcher = myPattern.matcher(match);  
    matcher.find();  
    return matcher.group(1);  
}
```

The factory code is very simple as well, for example:

```

public class MyIssueProviderFactory extends AbstractIssueProviderFactory {
    public MyIssueProviderFactory(@NotNull IssueFetcher fetcher) {
        // Type name usually starts with uppercase character because it is displayed in UI, but not
        necessarily.
        super(fetcher, "MyName");
    }

    @NotNull
    public IssueProvider createProvider() {
        return new MyIssueProvider(myFetcher);
    }
}

```

`IssueFetcher` is usually the central class performing plugin-specific logic. You have to implement `getIssue` method, which connects to the ITS remotely (via HTTP, XML-RPC, etc), passes authentication, retrieves the issue data and returns it, or reports an error. Example:

```

public IssueData getIssue(@NotNull String host, @NotNull String id,
                           @Nullable final Credentials credentials) throws Exception {
    final String url = getUrl(host, id);
    return getFromCacheOrFetch(url, new FetchFunction() {
        @NotNull
        public IssueData fetch() throws Exception {
            InputStream body = fetchHttpFile(url, credentials);
            IssueData result = null;
            if (body != null) {
                result = parseXml(body, url);
            }
            if (result == null) {
                throw new RuntimeException("Failed to fetch issue from '" + url + "'");
            }
            return result;
        }
    });
}

```

You need to implement how to compose the server URL and how do you parse the data out of XML (HTML). `AbstractIssueFetcher` will take care about caching, errors reporting and everything else.

#### Plugin UI

The only mandatory JSP required by TeamCity is `editIssueProvider.jsp` (the full path must be `/plugins/myName/admin/editIssueProvider.jsp`, that is, the plugin should have the jsp available `/admin/editIssueProvider.jsp` of its resources). This JSP is requested when the user opens the dialog for editing (or creating) the issue provider. In most cases it just renders the provider properties, or returns the form for filling them.

You can see the example in `/plugins/youtrack/admin/editIssueProvider.jsp`.

#### Version Control System Plugin

##### Overview

In TeamCity a plugin for Version Control System (VCS) is seen as a set of interface implementations grouped together by instances of

`jetbrains.buildServer.vcs.VcsSupportContext` (server-side part) and `jetbrains.buildServer.agent.vcs.AgentVcsSupportContext` (agent-side part).

The server-side part of a VCS plugin is responsible for the following major operations:

- collecting changes between versions
- building of a patch from version to version
- get content of a file (for web diff, duplicates finder and some other places)

There are also optional parts:

- labeling / tagging
- personal builds, which require corresponding support in IDE. This dependency may be eliminated in the future.

The agent-side part is optional and only responsible for checking out and updating project sources on agents. In contrast to server-side checkout it offers a traditional approach to interacting between a CI system and VCS – when source code is checked out into the same location where it's built. For pros & cons of both solutions see [VCS Checkout Mode](#).



You can use the source code of the existing VCS plugins as a reference, for example:

- [Git plug-in](#)
- [Mercurial plug-in](#)

For more information on TeamCity plugins, please refer to the [TeamCity Plugins](#) section.

Before digging into the VCS plugin development details, it's important to understand the basic terms such as a Version, Modification, Change, Patch, and Checkout Rule, which are explained below.

## Basic Terms

A Version is unambiguous representation of a particular snapshot within a repository pointed at by a VCS Root. The current version represents the head revision at the moment of obtaining.

The current version is taken by calling `jetbrains.buildServer.vcs.CollectSingleStatePolicy#getCurrentVersion(jetbrains.buildServer.vcs.VcsRoot)`. The version here is an arbitrary text. It can be a representation of a transaction number, a revision number, a date, whatever suitable enough for getting a source snapshot in a particular VCS. Usually format of the version depends on a version control system, the only requirement which comes from TeamCity — it should be possible to sort changes by version in order of their happening (see `jetbrains.buildServer.vcs.VcsSupportConfig#getVersionComparator()`).

Version is used in several places:

- for changes collecting
- for patch construction
- when content of the file is retrieved from the repository
- for labeling / tagging

TeamCity does not show Versions in the UI directly. For UI TeamCity converts a Version to its display name using `jetbrains.buildServer.vcs.VcsSupportConfig#getVersionDisplayName(String,jetbrains.buildServer.vcs.VcsRoot)`.

A Change is an atomic modification of a single file within a source repository. In other words, a change corresponds to a single increment of a file version.

A Modification is a set of changes made by some user at a certain time interval. It most closely corresponds to a single checkin transaction (commit), when a user commits all his modifications made locally to the central repository. A Modification also contains the Version of the VCS Root right after the corresponding Changes have been applied.

A collection of Modifications is what TeamCity expects as a result when asking a VCS plugin for changes.

A Patch is a set of operations to convert the directory state from one modification to another (e.g. change/add/remove file, add/remove directory).

A Checkout Rule is a way of changing default file layout.

Checkout rules allow to map the path in repository to another path on agent or to exclude some parts of repository, [read more](#).

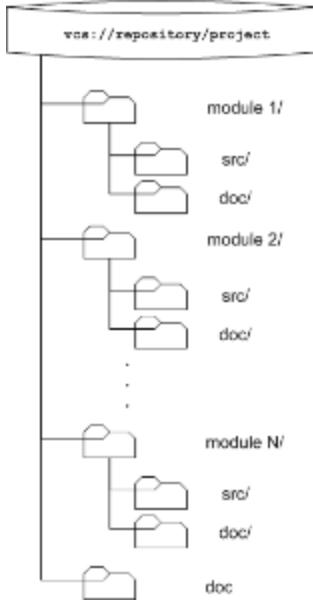
Checkout rules consist of include and exclude rules. Include rule can have "from" and "to" parts ("to" part allows to map path in repository to another path on agent). Mapping is performed by TeamCity itself and VCS plugin should not worry about it. However a VCS plugin can use checkout rules to speedup changes retrieval and patch building since checkout rules usually narrow a VCS Root to some its subset.

## Server-Side Part

## Patch Building and Change Collecting Policies

When implementing include rule policies it is important to understand how it works with Checkout Rules and paths. Let's consider an example with collecting changes.

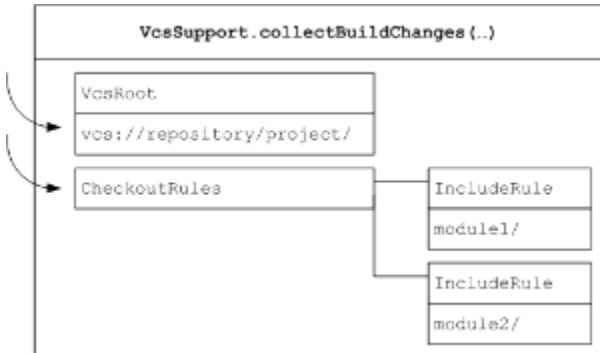
Suppose, we have a VCS Root pointing to `vcs://repository/project/`. The project root contains the following directory structure:



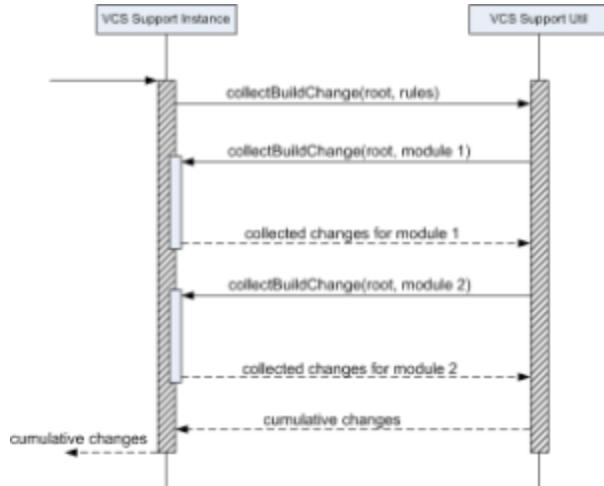
We want to monitor changes only in `module1` and `module2`. Therefore we've configured the following checkout rules:

```
\+:module1  
\+:module2
```

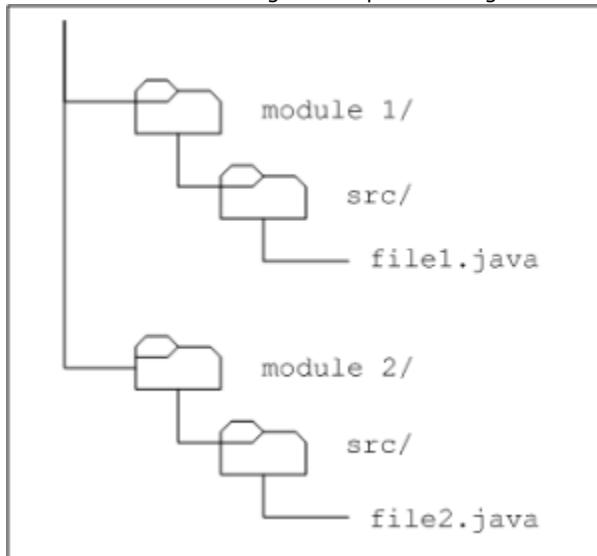
When `collectBuildChanges(...)` is invoked it will receive a `VcsRoot` instance that corresponds to `vcs://repository/project/` and a `CheckoutRules` instance with two `IncludeRules` — one for "module1" and the other for "module2".



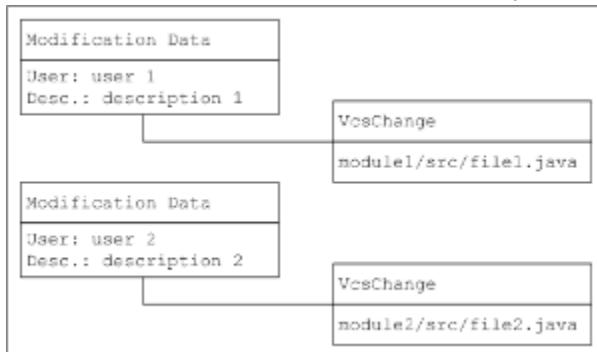
If `collectBuildChanges(...)` utilizes `VcsSupportUtil.collectBuildChanges(...)` it transforms the invocation into two separate calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. If you have implemented `CollectChangesByIncludeRule` in the way described in the listing above you will have the following interaction.



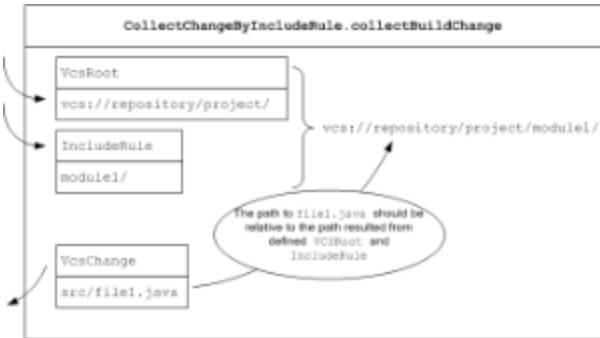
Now let's assume we've got a couple of changes in our sample repository, made by different users.



The collection of `ModificationData` returned by `vcsSupport.collectBuildChanges(...)` should then be like this:



But this is not a simple union of collections, returned by two calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. To see why let's have a closer look at the first call.

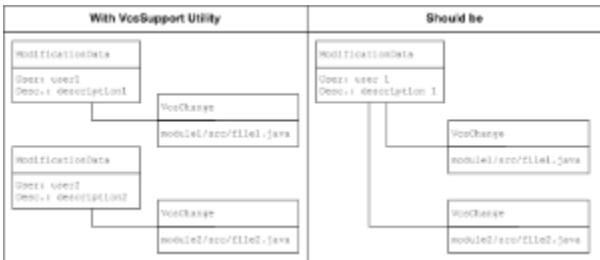


As you can see the paths in the resulting change must be relative to the path presented by the include rule, rather than the path in the VCS Root.

Then after collecting all changes for all the include rules `VcsSupportUtil` transforms the collected paths to be relative to the VCS Root's path.

Although being quite simple `VcsSupportUtil.collectBuildChanges(...)` has the following limitations.

Assume both changes in the example above are done by the same user within the same commit transaction. Logically, both corresponding `VcsChange` objects should be included into the same `ModificationData` instance. However, it's not true if you utilize `VcsSupportUtil.collectBuildChanges(...)`, since a separate call is made for each include rule.



Changes corresponding to different include rules cannot be aggregated under the same `ModificationData` instance even if they logically relate to the same commit transaction. This means a user will see these changes in separate change lists, which may be confusing. Experience shows that it's not very common situation when a user commits to directories monitored with different include rules. However, if the duplication is extremely undesirable an implementation should not utilize `VcsSupportUtil.collectBuildChanges(...)` and control `CheckoutRules` itself.

Another limitation is complete ignorance of exclude rules. As it said before this doesn't cause showing unneeded information in the UI. So an implementation can safely use `VcsSupportUtil.collectBuildChanges(...)` if this ignorance doesn't lead to significant performance problems. However, If an implementer believes the change collection speed can be significantly improved by taking into account include rules, the implementation must handle exclude rules itself.

All above is applicable to building patches using `VcsSupportUtil.buildPatch(...)` including the path relativity aspect.

## Server-Side Caching

By default, the server caches clean patches created by VCS plugins, because clean patch construction can take significant time on large repositories. If clean patches created by your VCS plugin need not to be cached, you should return `true` from the method `VcsSupport#ignoreServerCachesFor(VcsRoot)`.

## Agent-Side Part

### Agent-Side Checkout

Agent part of VCS plugin is optional, if it is provided then checkout can also be performed on the agent itself. This kind of checkout usually works faster but it may require additional configuration efforts, for example, if VCS plugin uses command line client then this client must be installed on all of the agents.

To enable agent-side checkout, be sure to include `jetbrains.buildServer.agent.vcs.AgentVcsSupportContext` into agent plugin part and also enable agent-side checkout via `jetbrains.buildServer.vcs.VcsSupportConfig#isAgentSideCheckoutAvailable()`.

## Version Control System Plugin (old style - prior to 4.5)

In TeamCity a plugin for Version Control System (VCS) is seen as an `jetbrains.buildServer.vcs.VcsSupport` instance. All VCS plugins must extend this class.

VCS plugin has a server side part and an optional agent side part. The server side part of a VCS plugin should support the following mandatory operations:

- collecting changes between versions
- building of a patch from version to version
- get content of a file (for web diff, duplicates finder, and some other places)

There are also optional parts:

- labeling / tagging
- personal builds

Personal builds require corresponding support in IDE. Chances are we will eliminate this dependency in the future.

 You can use source code of the existing VCS plugins as a reference, for example:

- <http://www.jetbrains.net/confluence/display/TW/Mercurial>
- <http://www.jetbrains.net/confluence/display/TW/AccuRev>

Before digging into the VCS plugin development details it's important to understand the basic notions such as Version, Modification, Change, Patch, Checkout Rule, which are explained below.

## Version

A Version is unambiguous representation of a particular snapshot within a repository pointed at by a VCS Root. The current version represents the head revision at the moment of obtaining.

The current version& is obtained from the `VcsSupport#getCurrentVersion(VcsRoot)`. The version here is arbitrary text. It can be transaction number, revision number, date and so on. Usually format of the version depends on a version control system, the only requirement which comes from TeamCity - it should be possible to sort changes by version in order of their appearance (see `VcsSupport#getVersionComparator()` method).

Version is used in several places:

- for changes collecting
- for patch construction
- when content of the file is retrieved from the repository
- for labeling / tagging

TeamCity does not show Versions in the UI directly. For UI, TeamCity converts a Version to its display name using `VcsSupport#getVersionDisplayName(String, VcsRoot)`.

## Collecting Changes

A Change is an atomic modification of a single file within a source repository. In other words, a Change corresponds to a single increment of the file version.

A Modification is a set of Changes made by some user at a certain moment. It most closely corresponds to a single checkin transaction, when a user commits all his modifications made locally to the central repository. A Modification also contains the Version of the VCS Root right after the corresponding Changes have been applied.

TeamCity server polls VCS for changes on a regular basis. A VCS plugin is responsible for collecting information about Changes (grouped into Modifications) between two versions.

Once a VCS Root is created the first action performed on it is determining the current Version (`VcsSupport#getCurrentVersion(VcsRoot)`). This value is stored and used during the next checking for changes as the "from" Version (`VcsSupport#collectBuildChanges(VcsRoot, String, String, CheckoutRules)`). The current Version is obtained again to be used as the "to" Version. The Modifications collected are then shown as pending changes for corresponding build configurations. After the checking for changes interval passes the server requests for next portion of changes, but this time the "from" Version is replaced with the previous "current" Version. And so on.

Obtaining the current Version may be an expensive operation for some version control systems. In this case some optimization can be done by implementing interface `CurrentVersionIsExpensiveVcsSupport`. Its method `CurrentVersionIsExpensiveVcsSupport#collectBuildChanges(VcsRoot, String, CheckoutRules)` takes only "from" Version assuming that the changes are to be collected for the head snapshot. In this case TeamCity will look for the Modification with the greatest Version in the returned Modifications and take it as the "from" parameter for the next checking cycle. If you implement `CurrentVersionIsExpensiveVcsSupport`, you can leave method `VcsSupport#collectBuildChanges(VcsRoot, String, String, CheckoutRules)` not implemented.

## Patch Construction

A Patch is the set of all modifications of a VCS Root made between two arbitrary Versions packed into a single unit. With Patches there is no need to retrieve all the sources from the repository each time a build starts. Patches are sent to agents where they are applied to the checkout directory. Patches in TeamCity have their own format, and should be constructed using `jetbrains.buildServer.vcs.patches.PatchBuilder`.

When a build is about to start, the server determines for which Versions the patch is to be constructed and passes them to `VcsSupport#buildPatch(VcsRoot, String, String, PatchBuilder, CheckoutRules)`.

There are two types of patch: clean patch (if `fromVersion` is null) and incremental patch (if `fromVersion` is provided). Clean patch is just an export of files on the specified version, while incremental patch is a more complex thing. To create incremental patch you need to determine the difference between two snapshots including files and directories creations/deletions.

## Checkout Rules

Checkout rules allow to map path in repository to another path on agent or to exclude some parts of repository, [read more](#).

Checkout rules consist of include and exclude rules. Include rule can have "from" and "to" parts ("to" part allows to map path in repository to another path on agent). Mapping is performed by TeamCity itself and VCS plugin should not worry about it. However a VCS plugin can use checkout rules to speedup changes retrieval and patch building since checkout rules usually narrow a VCS Root to some its subset.

In most cases it is simpler to collect changes or build patch separately by each include rule, for this VCS plugin can implement interface `jetbrains.buildServer.CollectChangesByIncludeRule` (as well as `jetbrains.buildServer.vcs.BuildPatchByIncludeRule`) and use `jetbrains.buildServer.vcs.VcsSupportUtil` as shown below:

```
public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String  
currentVersion, CheckoutRules checkoutRules)  
throws VcsException {  
    return VcsSupportUtil.collectBuildChanges(root, fromVersion, currentVersion, checkoutRules, this);  
}  
  
public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String  
currentVersion, IncludeRule includeRule)  
throws VcsException {  
    ... changes collecting code ...  
}
```

And for patch construction:

```
public void buildPatch(VcsRoot root, String fromVersion, String toVersion, PatchBuilder builder,  
CheckoutRules checkoutRules)  
throws IOException, VcsException {  
    VcsSupportUtil.buildPatch(root, fromVersion, toVersion, builder, checkoutRules, this);  
}  
  
public void buildPatch(VcsRoot root, String fromVersion, String toVersion, PatchBuilder builder,  
IncludeRule includeRule)  
throws IOException, VcsException {  
    ... build patch code ...  
}
```

If you want to share data between calls, this approach allows you to do it easily using anonymous classes:

```

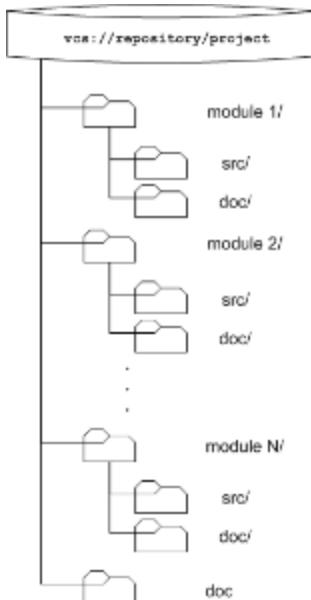
public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, CheckoutRules checkoutRules)
throws VcsException {
final MyConnection conn = obtainConnection(root); // get a connection to the repository
return VcsSupportUtil.collectBuildChanges(root, fromVersion, currentVersion, checkoutRules, new
CollectChangesByIncludeRule {
    public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, IncludeRule includeRule)
throws VcsException {
    doCollectChange(conn, includeRule); // use the same connection for all calls
}
});
}

public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, IncludeRule includeRule)
throws VcsException {
... changes collecting code ...
}

```

When using `VcsSupportUtil` it is important to understand how it works with Checkout Rules and paths. Let's consider an example with collecting changes.

Suppose, we have a VCS Root pointing to `vcs://repository/project/`. The project root contains the following directory structure:



We want to monitor changes only in `module1` and `module2`. Therefore we've configured the following checkout rules:

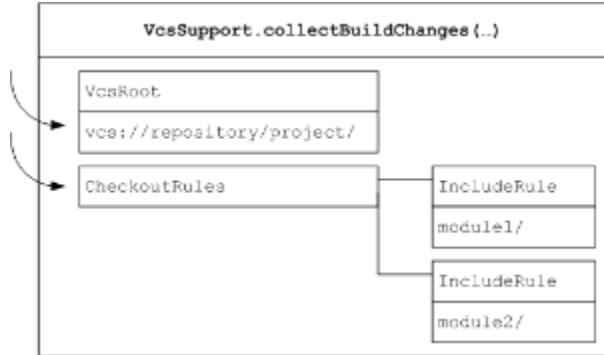
```

\+:module1
\+:module2

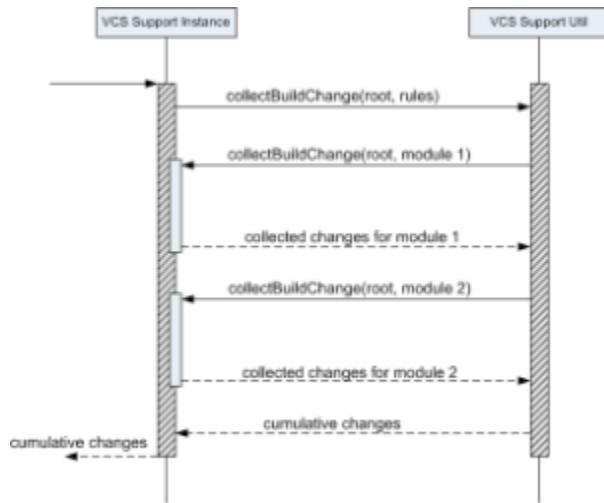
```

When `collectBuildChanges(...)` is invoked it will receive a `VcsRoot` instance that corresponds to `vcs://repository/project`

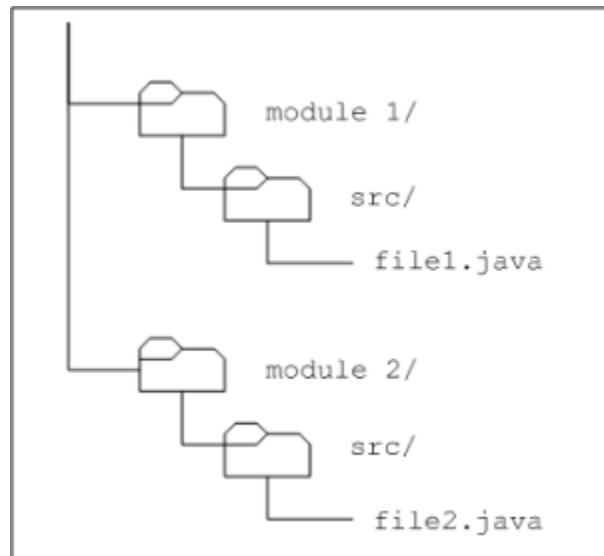
`t/||` and a `{CheckoutRules instance with two IncludeRules — one for "module1" and the other for "module2".`



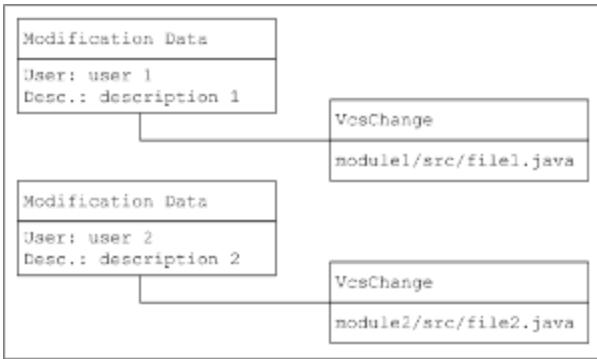
If `collectBuildChanges(...)` utilizes `VcsSupportUtil.collectBuildChanges(...)` it transforms the invocation into two separate calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. If you have implemented `CollectChangesByIncludeRule` in the way described in the listing above you will have the following interaction.



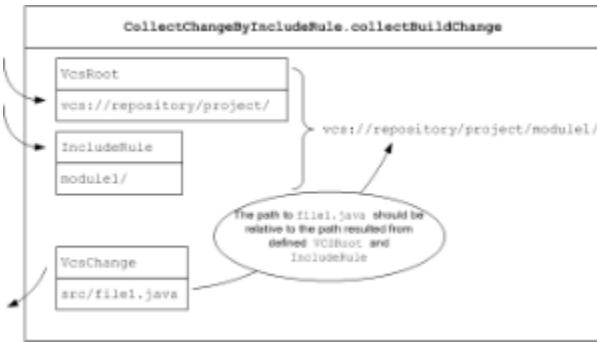
Now let's assume we've got a couple of changes in our sample repository, made by different users.



The collection of `ModificationData` returned by `VcsSupport.collectBuildChanges(...)` should then be like this:



But this is not a simple union of collections, returned by two calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. To see why let's have a closer look at the first calls.

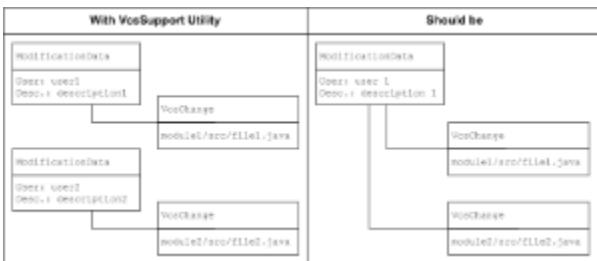


As you can see the paths in the resulting change must be relative to the path presented by the include rule, rather than the path in the VCS Root.

Then after collecting all changes for all the include rules `VcsSupportUtil` transforms the collected paths to be relative to the VCS Root's path.

Although being quite simple `VcsSupportUtil.collectBuildChanges(...)` has the following limitations.

Assume both changes in the example above are done by the same user within the same commit transaction. Logically, both corresponding `VcsChange` objects should be included into the same `ModificationData` instance. However, it's not true if you utilize `VcsSupportUtil.collectBuildChanges(...)`, since a separate call is made for each include rule.



Changes corresponding to different include rules cannot be aggregated under the same `ModificationData` instance even if they logically relate to the same commit transaction. This means a user will see these changes in separate change lists, which may be confusing. Experience shows that it's not very common situation when a user commits to directories monitored with different include rules. However if the duplication is extremely undesirable an implementation should not utilize `VcsSupportUtil.collectBuildChanges(...)` and control `CheckoutRules` itself.

Another limitation is complete ignorance of exclude rules. As it said before this doesn't cause showing unneeded information in the UI. So an implementation can safely use `VcsSupportUtil.collectBuildChanges(...)` if this ignorance doesn't lead to significant performance problems. However, If an implementer believes the change collection speed can be significantly improved by taking into account include rules, the implementation must handle exclude rules itself.

All above is applicable to building patches using `VcsSupportUtil.buildPatch(...)` including the path relativity aspect.

## Registering In TeamCity

During the server startup all VCS plugins are required to register themselves in the VCS Manager ( `jetbrains.buildServer.vcs.VcManager` ). A VCS plugin can receive the `VcsManager` instance using Spring injection:

```
class SomeVcsSupport implements VcsSupport {  
    ...  
    public SomeVcsSupport(VcsManager manager) {  
        manager.registerVcsSupport(this);  
    }  
    ...  
}
```

## Server side caches

By default, server caches clean patches created by VCS plugins, because on large repositories clean patch construction can take significant time. If clean patches created by your VCS plugin must not be cached, you should return true from the method `VcsSupport#ignoreServerCachesFor(VcsRoot)`.

## Agent side checkout

Agent part of VCS plugin is optional; if it is provided then checkout can also be performed on the agent itself. This kind of checkout usually works faster but it may require additional configuration efforts, for example, if VCS plugin uses command line client then this client must be installed on all of the agents.

To create agent side checkout, implement `jetbrains.buildServer.agent.vcs.CheckoutOnAgentVcsSupport` in the agent part of the plugin. Also server side part of your plugin must implement `jetbrains.buildServer.AgentSideCheckoutAbility` interface.

---

## See Also:

- Extending TeamCity: [Developing TeamCity Plugins | Typical Plugins](#)

## Custom Authentication Module

There are two types of custom authentication modules, which can be provided by plugins: credentials authentication modules and HTTP authentication modules. The first ones are used to check the credentials user typed in login form on the login page. The second ones are used to authenticate a user by HTTP request without showing login page at all.

- [Credentials Authentication Module](#)
- [HTTP Authentication Module](#)

## Credentials Authentication Module

Credentials authentication modules API is based on Sun JAAS API. To provide your own credentials authentication module you should provide a login module class which must implement the interface `javax.security.auth.spi.LoginModule` and register it in the `jetbrains.buildServer.serverSide.auth.LoginConfiguration`.

To make the authentication module active its type name can then be used during [Configuring Authentication Settings](#).

For example:

### CustomLoginModule.java

```
public class CustomLoginModule implements javax.security.auth.spi.LoginModule {  
    private Subject mySubject;  
    private CallbackHandler myCallbackHandler;  
    private Callback[] myCallbacks;  
    private NameCallback myNameCallback;  
    private PasswordCallback myPasswordCallback;  
  
    public void initialize(Subject subject, CallbackHandler callbackHandler, Map<String, ?> sharedState,  
    Map<String, ?> options) {  
        // We should remember callback handler and create our own callbacks.  
        // TeamCity authorization scheme supports two callbacks only: NameCallback and
```

```

PasswordCallback.

    // From these callbacks you will receive username and password entered on the login page.
    myCallbackHandler = callbackHandler;
    myNameCallback = new NameCallback("login:");
    myPasswordCallback = new PasswordCallback("password:", false);
    // remember references to newly created callbacks
    myCallbacks = new Callback[]{myNameCallback, myPasswordCallback};

    // Subject is a place where authorized entity credentials are stored.
    // When user is successfully authorized, the jetbrains.buildServer.serverSide.auth.ServerPrincipal
    // instance should be added to the subject. Based on this information the principal server will know
    a real name of
    // the authorized entity and realm where this entity was authorized.
    mySubject = subject;
}

public boolean login() throws LoginException {
    // invoke callback handler so that username and password were added
    // to the name and password callbacks
    try {
        myCallbackHandler.handle(myCallbacks);
    }
    catch (Throwable t) {
        throw new jetbrains.buildServer.serverSide.auth.TeamCityLoginException(t);
    }

    // retrieve login and password
    final String login = myNameCallback.getName();
    final String password = new String(myPasswordCallback.getPassword());

    // perform authentication
    if (checkPassword(login, password)) {
        // create ServerPrincipal and put it in the subject
        mySubject.getPrincipals().add(new ServerPrincipal(null, login));
        return true;
    }

    throw new jetbrains.buildServer.serverSide.auth.TeamCityFailedLoginException();
}

private boolean checkPassword(final String login, final String password) {
    return true;
}

public boolean commit() throws LoginException {
    // simply return true
    return true;
}

public boolean abort() throws LoginException {
    return true;
}

public boolean logout() throws LoginException {
    return true;
}

```

```
}
```

Now we should register this module in the server. To do so, we create a login module descriptor:

### CustomLoginModuleDescriptor.java

```
public class CustomLoginModuleDescriptor extends
jetbrains.buildServer.serverSide.auth.LoginModuleDescriptorAdapter {
    // Spring framework will provide reference to LoginConfiguration when
    // the CustomLoginModuleDescriptor is instantiated
    public CustomLoginModuleDescriptor(LoginConfiguration loginConfiguration) {
        // register this descriptor in the login configuration
        loginConfiguration.registerLoginModule(this);
    }

    public String getName() {
        // return unique name of this module type. e.g. a derivative id will then be used in
        "auth-config.xml" file
        return "custom";
    }

    @Override
    public String getDisplayName() {
        // return presentable name of this plugin
        return "My custom authentication plugin";
    }

    @Override
    public String getDescription() {
        // return human-readable description of this module type
        return "Authentication via custom login module";
    }

    public Class<? extends LoginModule> getLoginModuleClass() {
        // return our custom login module class
        return CustomLoginModule.class;
    }

    @Override
    public Map<String, ?> getJAASOptions(final Map<String, String> properties) {
        // return options which can be passed to our custom login module
        // for example, we could store reference to SBuildServer instance here
        return null;
    }
}
```

Finally we should create `build-server-plugin-ourUserAuth.xml` and zip archive with plugin classes as it is described [here](#) and write there `CustomLoginModuleDescriptor` bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-autowire="constructor">
  <bean id="customLoginModuleDescriptor" class="some.package.CustomLoginModuleDescriptor"/>
</beans>
```

Now you should be able to [change authentication scheme](#) to your authentication module.

#### HTTP Authentication Module

To provide your own HTTP authentication module you should provide a class which must implement the interface `jetbrains.buildServer.controllers.interceptors.auth.HttpAuthenticationScheme` and register it in the `jetbrains.buildServer.serverSide.auth.LoginConfiguration`.

To make the authentication module active its type name can then be used during [Configuring Authentication Settings](#).

For example:

## CustomHttpAuthenticationScheme.java

```
public class CustomHttpAuthenticationScheme extends
jetbrains.buildServer.controllers.interceptors.auth.HttpAuthenticationSchemeAdapter {
    // Spring framework will provide reference to LoginConfiguration when
    // the CustomLoginModuleDescriptor is instantiated
    public CustomHttpAuthenticationScheme(final LoginConfiguration loginConfiguration) {
        // register this scheme in the login configuration
        loginConfiguration.registerAuthModuleType(this);
    }

    @Override
    protected String doGetName() {
        // return unique name of this module type. e.g. a derivative id will then be used in
        "auth-config.xml" file
        return "custom";
    }

    @Override
    public String getDisplayName() {
        // return presentable name of this plugin
        return "My custom HTTP authentication plugin";
    }

    @Override
    public String getDescription() {
        // return human-readable description of this module type
        return "Authentication via custom HTTP scheme";
    }

    // main method to process HTTP authentication
    @Override
    public HttpAuthenticationResult processAuthenticationRequest(final HttpServletRequest request,
                                                               final HttpServletResponse response,
                                                               final Map<String, String> properties) throws IOException {
        if (!isAttemptToAuthenticateViaThisHTTPScheme(request)) {
            return HttpAuthenticationResult.notApplicable();
        }

        // perform authentication
        final String username = authenticate(request);
        if (username == null) {
            return HttpAuthUtil.sendUnauthorized(request, response, "Failed to authenticate user", this,
properties);
        }

        return HttpAuthenticationResult.authenticated(new ServerPrincipal(null, username), true);
    }
}
```

Finally we should create `build-server-plugin-ourUserAuth.xml` and zip archive with plugin classes as it is described [here](#) and write there `CustomHttpAuthenticationScheme` bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-autowire="constructor">
<bean id="customHttpAuthenticationScheme"
class="some.package.CustomHttpAuthenticationScheme"/>
</beans>
```

Now you should be able to [change authentication scheme](#) to your authentication module.

## Custom Notifier

Custom notifier must implement `jetbrains.buildServer.notification.Notificator` interface and register implementation in the `jetbrains.buildServer.notification.NotificatorRegistry`.

When a notifier is registered, it can provide information about additional properties that must be filled in by the user. To obtain values of these properties, use the following code:

```
String value = user.getPropertyValue(new NotificatorPropertyKey(<notifier type>, <property name>));
```

Notifier can also provide custom UI for [Notifier rules](#) and [My Settings&Tools](#) pages. See `PlaceId.NOTIFIER_SETTINGS_FRAGMENT` and `PlaceId.MY_SETTINGS_NOTIFIER_SECTION`.

Notifications are only delivered if there is at least one subscribed user for given event.



Use source code of the existing plugins as a reference:

- <http://code.google.com/p/buildbunny/wiki/CreateTeamcityNotifier> - instructions, source code at [GitHub](#)
- <https://github.com/ndrake/tcgrowl>

See also:

[Concepts: Notifier](#)

[User's Guide: Subscribing to Notifications](#)

[Administrator's Guide: Customizing Notifications](#)

## Custom Statistics

TeamCity provides a number of ways to customize statistics. You can add your own custom metrics to integrate your tools/processes, insert any statistical chart/report into statistic page extension places and so on.

This page describes programmatic approaches to statistics customization. For user-level customizations, please refer to the [Custom Chart](#) page.

On this page:

- [Customize TeamCity Statistics Page](#)
  - [Add Chart](#)
  - [Add Custom Content](#)
- [Add Statistics to your Custom Pages](#)
  - [Customize Chart Appearance](#)
- [Add Custom Build Metrics](#)

### Customize TeamCity Statistics Page

#### Add Chart

To add a chart to the Statistics tab for a project or build configuration, use the `ChartProviderRegistry` bean:

```

public MyExtension(ChartProviderRegistry registry, ProjectManager manager) {
    registry.getChartProvider("project-graphs").registerChart(manager.findProjectByExternalId("externalId"), createGraphBean());
    // "project-graphs" for Project Statistics Tab
    // "buildtype-graphs" for Build Configuration Statistics Tab
}

public GraphBean createGraphBean() {
    // creates GraphBean
}

```

#### Add Custom Content

To add custom content to the Statistics tab for a project or build configuration, use the following example [here](#) and the appropriate `PlaceId`:

- for Build Configuration Statistics tab, use `PlaceId.BUILD_CONF_STATISTICS_FRAGMENT`
- for Project Statistics tab, use `PlaceId.PROJECT_STATS_FRAGMENT`.

#### Add Statistics to your Custom Pages

To add charts to your custom JSP pages, use the `<buildGraph>` tag and a special controller accessible on `"/buildGraph.html"`. It requires the `jsp` attribute leading to your page:

```

new ModelAndView("/buildGraph.html?jsp=" +
myDescriptor.getPluginResourcesPath("sampleChartPage.jsp"))

```

To insert statistics chart into the `sampleChartPage.jsp`:

```

<%@taglib prefix="stats" tagdir="/WEB-INF/tags/graph"%>
<stats:buildGraph id="g1" valueType="BuildDuration"/>

```

#### Customize Chart Appearance

Attribute	Description	Usage
<code>width, height</code>	modify the chart image size	Integer value
<code>hideFilters</code>	suppress filter controls	Comma separated filters names: 'all', 'series', 'average', 'showFailed', 'range', 'yAxisType', 'forceZero' or 'yAxisRange'
<code>defaultFilter</code>	default filter state	Comma separated names: 'showFailed', 'averaged', 'logYAxis', 'autoscale'
<code>hints</code>	chart style	Set to 'rendererB' for a bar chart

#### Add Custom Build Metrics

To add a custom build metric, in addition to the [built-in methods](#), you can extend `BuildValueTypeBase` to define your build metric calculation method, appearance, and key. After that you can reference this metric by its key in statistics chart/report tags.

See also:

[Extending TeamCity: Build Script Interaction with TeamCity](#)

## Custom Server Health Report

To report custom server health items, do the following:

- Create your own reporter
- Create a custom page extension to render items reported by you

### Reporting Server Health Items

To make a reporter, create a subclass of `jetbrains.buildServer.serverSide.healthStatus.HealthStatusReport`.

Particularly, you must override method `jetbrains.buildServer.serverSide.healthStatus.HealthStatusReport#report(jetbrains.buildServer.serverSide.healthStatus.HealthStatusScope, jetbrains.buildServer.serverSide.healthStatus.HealthStatusItemConsumer)`.

The items should be reported according to the analysis scope passed as a parameter to this method using the appropriate method of `resultConsumer`.

If you try to consume an object which is not in the scope of the current analysis, it will be filtered out by the consumer and will not appear in the report.

This method is always called with system privileges (in all permissions mode). Any permissions checks should be avoided here.

While reporting items, additional data required for further items presentation could be provided.

### Presenting Server Health Items to User

To present reported items, provide a [custom page extension](#) and connect it to `PlaceId.HEALTH_STATUS_ITEM`.

The simplest way to do it is to create a subclass of `jetbrains.buildServer.web.openapi.healthStatus.HealthStatusItemPageExtension`.

#### Handling Current User Permissions

To handle permissions of the user viewing the reported items, use the subclass of `HealthStatusItemPageExtension`.

In order to do it, override the `jetbrains.buildServer.web.openapi.healthStatus.HealthStatusItemPageExtension#isAvailable` method.

You can also limit the set of pages where your items should be presented to the Administration area of the Web UI by setting `FALSE` to `jetbrains.buildServer.web.openapi.healthStatus.HealthStatusItemPageExtension#setVisibleOutsideAdminArea`

The particular strategy of handling permissions depends on the report details. The proposed behaviour is to completely hide items from the user only if none of related objects is available. In other cases it makes sense to show the item, but filter all inaccessible objects.

#### Switching between Display Modes

There are the following places in the Web UI where Server Health items could be presented:

- the report page (Administration | Server Health)
- the notes section at the top of all pages (global items with severity more than 'info')
- in-place (in the popups appearing on some pages).

To define in what display mode a server health item is presented, use

`jetbrains.buildServer.web.openapi.healthStatus.HealthStatusItemDisplayMode`. The `HealthStatusItemDisplayMode.GLOBAL` value is passed to the request when an item is shown on the report page, `HealthStatusItemDisplayMode.IN_PLACE` is used in other cases.

Here is an example of handling the display mode in a JSP page.

```
<jsp:useBean id="showMode"
type="jetbrains.buildServer.web.openapi.healthStatus.HealthStatusItemDisplayMode"
scope="request"/>
<c:set var="inplaceMode" value="<% =HealthStatusItemDisplayMode.IN_PLACE %>" />

<c:choose>
<c:when test="${showMode == inplaceMode}">
    //Smth about current object
</c:when>
<c:otherwise>
    //Smth about related object
</c:otherwise>
</c:choose>
```

## Presenting Results In-place

While presenting results in-place, it might be necessary to know the ID of an object being viewed at the moment. The ID can be retrieved using the following requests:

- on the Edit the VCS root settings page

```
<jsp:useBean id="vcsRootId" type="java.lang.String" scope="request"/>
```

- on the Edit the Build Configuration setting page

```
<jsp:useBean id="buildTypeId" type="java.lang.String" scope="request"/>
```

- on the Edit the Build Configuration Template settings page

```
<jsp:useBean id="templateId" type="java.lang.String" scope="request"/>
```

## Extending Highlighting for Web diff view

TeamCity uses [JHighlight](#) library to render the code on [diff view](#) page. Essentially what JHighlight is doing is it takes plain source code, recognizes the language by extension, parses it, and in case of success renders the HTML output where the tokens are highlighted according to the specified settings. Unfortunately JHighlight supports relatively small subset of languages out-of-the-box (major ones like Java, C++, XML, and several more). Here we'd like to present you a HOWTO on adding the support for more languages.



Please note that in the further versions TeamCity may switch to another highlighting engine, so the changes you make will only work while JHighlight is used by TeamCity.

As an example we are implementing a highlighting for properties files, like this one:

```
# Comment on keys and values
key1=value1
foo = bar
x=y
a b c = foo bar baz

! another comment
! more complex cases:
a\=\\fb : x\\ty\\n\\x\\uzzzz

key = multiline value \
still value \
still value
the key
```

The implementation consists of the following steps:

- Step one: Writing a lexer using flex language
- Step two: Generating a lexer on java
- Step three: The renderer class.
- Step four: Running the JHighlight
- Including JHighlight Changes into TeamCity Distribution

Step one: Writing a lexer using flex language

To understand this step you might need to familiarize yourself with a [JFlex](#) syntax.

There are several things you need to define in a flex file in order to generate a lexer. First of all, token types or, in our case, styles.

```
public static final byte PLAIN_STYLE = 1;
public static final byte NAME_STYLE = 2;
public static final byte VALUE_STYLE = 3;
public static final byte COMMENT_STYLE = 4;
```

These constants will be mapped to the lexems in a source code and to the CSS classes, so at this moment you should decide which tokens are to be highlighted.

We will highlight names, values of properties, comments and plain text, which is just '=' character.

Then you need to specify the states and actual parsing rules:

```
WhiteSpace = [ \t\f]

%state IN_VALUE

%%

/* Rules for YYINITIAL state */
<YYINITIAL> {
    "\n"           { return PLAIN_STYLE; }

    {WhiteSpace}      { return PLAIN_STYLE; }

    [Extending Highlighting for Web diff view^=\n\t\f ]+ { return NAME_STYLE; }

    "="            { yybegin(IN_VALUE); return PLAIN_STYLE; }

    [#!] [Extending Highlighting for Web diff view^\n]* \n { return COMMENT_STYLE; }

/* Rules for IN_VALUE state */
<IN_VALUE> {
    "\\\n"         { return VALUE_STYLE; }

    "\n"           { yybegin(YYINITIAL); return PLAIN_STYLE; }

    [Extending Highlighting for Web diff view^\\\n]+ { return VALUE_STYLE; }

/* error fallback */
.\n           { return PLAIN_STYLE; }
```

Our simple lexer has two states: initial (YYINITIAL - it is predefined) and IN\_VALUE. In each of these states we try to handle the next character (or a group of characters) using regexp rules.

The rules are applied from the top to the bottom, the first one that matches non-empty string is used. Each rule is associated with the action to be performed on runtime. Here we have only simple actions that return the token constant and sometimes change the state.

To end the composition of a lexer add the common part to be inserted to the Java file. It's unlikely that you need to modify it. Here's the full result code:

```
package com.uwyn.jhighlight.highlighter;
```

```

import java.io.Reader;
import java.io.IOException;

%%

%class PropertiesHighlighter
%implements ExplicitStateHighlighter

%unicode
%pack

%buffer 128

%public

%int

%{
    /* styles */

    public static final byte PLAIN_STYLE = 1;
    public static final byte NAME_STYLE = 2;
    public static final byte VALUE_STYLE = 3;
    public static final byte COMMENT_STYLE = 4;

    /* Highlighter implementation */

    public byte getStartState() {
        return YYINITIAL+1;
    }

    public byte getCurrentState() {
        return (byte) (yystate()+1);
    }

    public void setState(byte newState) {
        yybegin(newState-1);
    }

    public byte getNextToken() throws IOException {
        return (byte) yylex();
    }

    public int getTokenLength() {
        return yylength();
    }

    public void setReader(Reader r) {
        this.zzReader = r;
    }

    public PropertiesHighlighter() {
    }
}

WhiteSpace = [ \t\f]
%state IN_VALUE

```

```
%%

/* Rules for YYINITIAL state */
<YYINITIAL> {
    "\n"           { yybegin(YYINITIAL); return PLAIN_STYLE; }

    {WhiteSpace}     { return PLAIN_STYLE; }

[Extending Highlighting for Web diff view^=\n\t\f ]+      { return NAME_STYLE; }

"="             { yybegin(IN_VALUE); return PLAIN_STYLE; }

[#!] [Extending Highlighting for Web diff view^\n]* \n      { return COMMENT_STYLE; }
}

/* Rules for IN_VALUE state */
<IN_VALUE> {
    "\\n"          { return VALUE_STYLE; }

    "\n"           { yybegin(YYINITIAL); return PLAIN_STYLE; }

[Extending Highlighting for Web diff view^\\n]+      { return VALUE_STYLE; }
}
```

```
/* error fallback */
.\n        { return PLAIN_STYLE; }
```

That's it: the lexer is ready. Download the latest JHighlighter sources from the [repository](#) (version 1.0) and put this file to the `src/com/uwyn/jhighlight/highlighter` directory of JHighlight distribution.

Step two: Generating a lexer on java

You can compile the code above using a JFlex tool, or amend the build.xml file adding the following task to the "flex" target:

```
<jflex file="${src.dir}/com/uwyn/jhighlight/highlighter/PropertiesHighlighter.flex"
      destdir="${src.dir}"
      verbose="on"
      nobak="on"/>
```

After the compilation we'll have a java class `PropertiesHighlighter` implementing `ExplicitStateHighlighter` interface. If the previous steps are done right, you won't need to modify this file by hand.

Step three: The renderer class.

The only JHighlight class left is the renderer corresponding to the generated lexer. This class should extend a `XhtmlRenderer` class and provide CSS classes correspondence along with default CSS map:

```

package com.uwyn.jhighlight.renderer;

import com.uwyn.jhighlight.highlighter.ExplicitStateHighlighter;
import com.uwyn.jhighlight.highlighter.PropertiesHighlighter;
import com.uwyn.jhighlight.renderer.XhtmlRenderer;
import java.util.HashMap;
import java.util.Map;

public class PropertiesXhtmlRenderer extends XhtmlRenderer {
    // Contains the default CSS styles.
    public final static HashMap DEFAULT_CSS = new HashMap() {{
        put(".properties_plain",
            "color: rgb(0,0,0);");

        put(".properties_name",
            "color: rgb(0,0,128); " +
            "font-weight: bold;");

        put(".properties_value",
            "color: rgb(0,128,0); " +
            "font-weight: bold;");

        put(".properties_comment",
            "color: rgb(128,128,128); " +
            "background-color: rgb(247,247,247);");
    }};
}

protected Map getDefaultCssStyles() {
    return DEFAULT_CSS;
}

// Maps the token type with the CSS class. E.g. each token of a 'PLAIN_STYLE' type will be
// rendered with 'properties_plain' style (see above).
protected String getCssClass(int style) {
    switch (style) {
        case PropertiesHighlighter.PLAIN_STYLE:
            return "properties_plain";
        case PropertiesHighlighter.NAME_STYLE:
            return "properties_name";
        case PropertiesHighlighter.VALUE_STYLE:
            return "properties_value";
        case PropertiesHighlighter.COMMENT_STYLE:
            return "properties_comment";
    }

    return null;
}

protected ExplicitStateHighlighter getHighlighter() {
    return new PropertiesHighlighter();
}
}

```

You can leave DEFAULT\_CSS empty, but in this case the styles should always be present in jhighlight.properties file. But it is essential that PropertiesHighlighter token constants are mapped to the CSS styles.

Also we need to tell the factory class that a new renderer exists: for this XhtmlRendererFactory class should be updated. We don't provide the code here as it is very simple (in fact, two lines should be added).

Step four: Running the JHighlight

JHighlight patch is ready, let's check it out in action. Put the properties file to the 'examples' directory and run the commands from JHighlight home directory:

```
ant  
java -cp build/classes/ com.uwyn.jhighlight.JHighlight examples/  
firefox examples/test.properties.html
```

Voilà! Our properties file is highlighted:



Including JHighlight Changes into TeamCity Distribution

TeamCity uses only public JHighlight API, that's why if your patched JHighlight successfully generates the HTML, you have to do just few steps to integrate it to TeamCity:

- repack jhighlight.jar (call ant jar)
- replace /WEB-INF/lib/jhighlight-njcms-patch.jar with it
- restart TeamCity server

Good luck!

## External Storage Implementation Guide

Since TeamCity 2017.1, an API is provided to enable writing TeamCity plugins which can store TeamCity build artifacts in a custom storage. This guide details implementation of support for an external storage system as a TeamCity plugin.

You can use the following plugins from JetBrains as implementation examples:

- [S3 Artifact Storage](#)
- [Azure Artifact Storage](#)
- [Google Cloud Artifact Storage](#)

On this page:

- [TeamCity Artifacts Overview](#)
- [External Artifacts Storage Overview](#)
- [Implementation](#)
  - [Settings](#)
  - [Publication](#)
  - [View](#)
  - [Cleanup](#)

## TeamCity Artifacts Overview

TeamCity provides the following artifacts-related features:

- [Artifacts upload](#)
- Individual artifacts download and browsing of build artifacts in a web browser and via the [REST API](#)

- Ability to configure [artifact dependencies](#) between builds and fetching necessary dependencies on the agent

Upload to a TeamCity server is a process of storing data created by a build, so that it is available after a TeamCity agent is disconnected. The data is uploaded from the agent to the server via HTTP multipart requests. Usually the upload process starts when a build finishes on the agent, but it is also possible to initiate the upload while the build is in progress using [service messages](#). Artifacts are uploaded according to [artifacts paths](#). Uploaded artifacts are also cached on the build agent in case they are requested by another build on this agent via artifact dependencies.

Uploaded data is displayed in the TeamCity web UI as an artifacts tree in the popups or on the "Artifacts" tab of the build results. It also can be accessed by [http requests](#) via the [REST API](#) and as an [ivy-compliant repository](#).

TeamCity can also deliver artifacts of one build to another build with the help of [Artifacts dependencies](#). Artifact dependency configuration includes the source build settings (build configuration, version), artifact patterns for matching the source build artifacts, and destination paths on the target agent. Downloaded artifact dependencies are cached on build agents to reduce the download time.

Build artifacts also contain a number of [internal artifacts](#). They include (but not limited to) build logs, build properties, coverage reports, etc. These artifacts are required for TeamCity features to function properly, and unless specified explicitly, they are not removed by clean-up and not downloaded as dependencies.

Build artifacts can be cleaned up according to [Cleanup Rules](#).

## External Artifacts Storage Overview

An implementation of an external storage should be able to upload to, download, and remove artifacts from the external storage.

The external storage plugin should be able to upload artifacts to the storage during a build on agent, send them to the client on a request and handle the cleanup of the artifacts.

While delegating some of its features to the plugin, TeamCity keeps a part of its internal functions intact. Regardless of external storage settings, internal artifacts (including build logs) are still published to the TeamCity server. Besides, currently the artifacts tree is rendered in the web UI by the TeamCity server itself.

## Implementation

### Settings

The TeamCity artifacts storage is configured on the "Project Settings" page under the dedicated tab. Using the tab, a TeamCity user can choose which storage will be used for builds' artifacts. Also, the relevant settings page is displayed there. The choice will be applied to all build configurations in the project and its subprojects.

To get listed on this page, the plugin will provide a spring bean implementing the `jetbrains.buildServer.serverSide.artifacts.ArtifactStorageType` abstract class and should be registered in the `jetbrains.buildServer.serverSide.artifacts.ArtifactStorageTypeRegistry`.

### Publication

Publication is done from the build agent process. The plugin should provide a spring bean implementing the interface `jetbrains.buildServer.agent.ArtifactsPublisher` (for future compatibility we recommend extending the base implementation `jetbrains.buildServer.agent.ArtifactsPublisherBase`). The plugin should publish information about remote artifacts using `jetbrains.buildServer.agent.artifacts.AgentArtifactHelper#publishArtifactList` after a build is finished. This information will be stored in a special index file as a hidden build artifact and can be accessed later on.

### View

If the external artifacts index was created during publication using `jetbrains.buildServer.agent.artifacts.AgentArtifactHelper#publishArtifactList`, it will be used by the TeamCity server when listing build artifacts, e.g. when adding nodes to the artifacts tree.

To access artifact content via HTTP requests, the plugin should provide an implementation of the `jetbrains.buildServer.web.openapi.artifacts.ArtifactDownloadProcessor` interface.

To access artifact content for other purposes, it should provide an implementation of the `jetbrains.buildServer.serverSide.artifacts.ArtifactContentProvider` interface.

### Cleanup

For cleanup, the plugin is expected to have a Spring bean implementing `jetbrains.buildServer.serverSide.cleanup.CleanUpExtension`. It is recommended to make this bean [PositionAware](#) and place it [first](#) to make sure the extension is called before the default TeamCity clean-up procedures (that will remove builds and data stored on the disk).

The implementation should use `jetbrains.buildServer.serverSide.cleanup.BuildCleanupContext#getErrorReporter` to report errors which occurred during the clean-up, and `jetbrains.buildServer.serverSide.artifacts.ServerArtifactHelper#removeFromArtifactList` to remove artifacts which were successfully cleaned up from the storage from the artifact list stored in the TeamCity.

## Bundled Development Package

 TeamCity is hiring! Learn about [the available vacancies](#) on the JetBrains site. [Read](#) about working in the TeamCity team.

TeamCity comes bundled with a Development Package that can be used to start developing TeamCity plugins.

To get the package, use the `.tar.gz` or `.exe` distribution.

Upon installation, `<TeamCity Home Directory>` will have the `devPackage` directory which contains TeamCity open API binaries, javadoc, sources and archive with a sample plugin.

### `devPackage` directory description

There are mainly two types of plugins in TeamCity: server-side plugins and agent-side plugins.

To develop an agent-side plugin, you need the following part of the Open API:

- `serviceMessages.jar`
- `common-api.jar`
- `agent-api.jar`

Correspondingly for the server-side plugin, you need:

- `serviceMessages.jar`
- `common-api.jar`
- `server-api.jar`

Note that sometimes a part of an agent-side plugin has to work in the same JVM where the build tool is executing. For example, some custom test runner can be executed in the JVM where the tests are running. The `runtime` directory of `devPackage` contains some jars that can be used in this case.

`devPackage` also contains some base classes for tests under the `tests` directory.

## Sample Plugin

### Building and deploying sample plugin

#### Building plugin with Apache Ant

- Unpack `<TeamCity Home Directory>\devPackage\samplePlugin-src.zip` into a directory of your choice
- Edit the `build.properties` file and set the value for `path.variable.teamcitydistribution` property to the path of `<TeamCity Home Directory>`
- Run `ant dist` in the plugin directory (Ant 1.7+ is recommended). The plugin distribution should be created in the `dist` directory.

#### Building sample plugin in IntelliJ IDEA

- Unpack `<TeamCity Home Directory>\devPackage\samplePlugin-src.zip` into a directory of your choice
- Open the project in IDEA (the `.idea` project should work OK in IntelliJ IDEA 9 and later (including IntelliJ IDEA 9.0 Community Edition))
- On prompt to add the path variable, set the "TeamCityDistribution" path variable to the directory where TeamCity with `devPackage` is installed (`<TeamCity Home Directory>`).
- Open Project Structure and ensure you have Project SDK with name "1.6" pointing to Sun JDK version 1.6

#### Running the server with plugin from IDEA

- Either edit the `build.properties` file to set the `path.variable.teamcitydistribution` property or regenerate the build script from IDEA (execute "Generate Ant Build" with the settings: single file, all other options unchecked).

If you use the Ultimate edition of IntelliJ IDEA, you can start TeamCity right from the IDE:

- Go to the "server" run configuration settings and configure Application Server pointing it to <TeamCity Home Directory>
- Run the "server" run configuration. It will run Ant create distribution task, deploy the plugin into \${user.home}/.BuildServer directory and run the TeamCity server.

If you use the Community edition, see [#Building plugin with Apache Ant](#) - you can run "deploy" Ant build target right from Ant Build IDEA tool window and then start TeamCity manually.

## Sample Plugin Functionality

The sample plugin adds "Click me!" button in the bottom of "Projects" page. Click it to navigate to the plugin description page.

## Open API Changes



TeamCity is hiring! Learn about [the available vacancies](#) on the JetBrains site. [Read](#) about working in the TeamCity team.

### Changes from 2017.1 to 2017.2

- With the introduction of the ability to attach a build configuration to multiple templates the following changes have been made:
  - In BuildTypeSettings:
    - getTemplate() and getTemplateId were deprecated
    - getTemplates() and getTemplateIds() were introduced instead
  - in SBuildType:
    - attachToTemplate(BuildTypeTemplate) and detachFromTemplate() were deprecated
    - addTemplate(BuildTypeTemplate, boolean), and removeTemplates(Collection<? extends BuildTypeTemplate>, boolean) were introduced instead
    - also setTemplates(List<? extends BuildTypeTemplate>, boolean) and setTemplatesOrder(List<String>) were added
- With the introduction of default templates the following changes have been made:
  - in SProject:
    - getDefaultTemplate(boolean) has been added
  - in SBuildType:
    - getOwnTemplates() has been added
  - in BuildTypeTemplate:
    - getUsagesAsDefaultTemplate() and getNumberOfUsagesAsDefaultTemplate() have been added

### Changes from 10.0 to 2017.1

- ArtifactsURLProvider is renamed to ArtifactAccessor. Several methods were added to support artifact retrieval using this interface instances.
- ServerVersionInfo.getDisplayVersionMajor and ServerVersionInfo.getDisplayVersionMinor now return int instead of byte

### Changes from 9.1 to 10.0

- UptodateValue:TimeToLiveProvider#getTimeToLiveMillis since 10.0 has a parameter - value to be cached, so the cache time can depend on the cached value
- AgentLifeCycleListener and AgentLifeCycleAdapter have two new methods:
  - dependenciesDownloaded - called when all artifact dependencies of the build have been successfully resolved and downloaded
  - preparationFinished - called when all preparations for the build are finished (sources checkout, personal patch, artifact dependencies, free disk space requirement, etc)
- jetbrains.buildServer.web.openapi.healthStatus.suggestions.ProjectSuggestion class can be used as base class for project-level suggestions
- Service messages-related classes are no longer available in common-api.jar. serviceMessages.jar is now essential part of Common API.

This change only affects compile time (already compiled binaries will work as is). To fix compile-time "NoClassDefFound" errors, add serviceMessages.jar to your project's library.

- Methods SRunningBuild.addBuildMessage and SRunningBuild.addBuildMessages are deprecated and cannot be used in version 10.

Plugins, using these methods should be rewritten to use jetbrains.buildServer.serverSide.buildLog.BuildLog methods (see SRunningBuild.getBuildLog).

- BuildServerListener.messageReceived event will not work in two-node configuration, and will be removed in the future.

Consider using some other approach. For instance, a plugin can obtain an Iterator from build log (`BuildLog.getMessagesIterator`) and tail it periodically in background.

- Signature of `jetbrains.buildServer.messages.BuildMessagesTranslator.translateMessages` method has been changed, now it accepts list of messages instead of single message: `List<BuildMessage1> translateMessages(SRunningBuild build, List<BuildMessage1> messages)`
- `jetbrains.buildServer.BuildAgent` has new method `int getAgentPoolId()`
- `jetbrains.buildServer.clouds.CloudImage` has new method `Integer getAgentPoolId()` - it represents the agent pool that instances from this image will fall into.

The value is nullable, which means that agent pool for the instances can be configured manually in Agents->Pools UI

Changes from 9.0 to 9.1

### Issue Trackers integration API changes

- `IssueProviderType` has been extracted as a separate class. It serves as issue tracker integration descriptor and contains id, display name and URLs to controller and issue rendering pages
- `AbstractIssueProviderFactory` now takes `IssueProviderType` as an argument rather type as a String  
To be compatible with 9.1, existing plugins must implement `IssueProviderType` and change the corresponding provider factory according to base class interface. [This change](#) in Github integration plugin can be taken as an example

Changes from 8.1.x to 9.0

### Server API changes

- `jetbrains.buildServer.serverSide.dependency.Dependent#getDependencyReferences` and `jetbrains.buildServer.serverSide.dependency.Dependent#getNumberOfDependencyReferences` were moved to `jetbrains.buildServer.serverSide.SBuildType`
- `jetbrains.buildServer.issueTracker.IssueProviderFactory#getDisplayName` display name for issue tracker in UI.

Changes from 7.1.x to 8.0

### External ID -related changes

- `jetbrains.buildServer.serverSide.dependency.DependencyFactory#createDependency` now accepts external ID instead of internal one, use `jetbrains.buildServer.serverSide.dependency.DependencyFactory#createDependencyByInternalId` for internal ID.
- `jetbrains.buildServer.serverSide.ArtifactDependencyFactory#createArtifactDependency` now accepts external ID instead of internal one, use `jetbrains.buildServer.serverSide.ArtifactDependencyFactory#createArtifactDependencyByInternalId` for internal ID.

### Common API changes

- `SimpleCommandLineProcessRunner.RunCommandEvent` => `SimpleCommandLineProcessRunner.ProcessRunCallback`
- added `jetbrains.buildServer.serverSide.CachePaths` for plugins to get cache directory on server
- `jetbrains.buildServer.serverSide.statistics.ValueType#getFormat` now returns String constant representing format style
- `jetbrains.buildServer.serverSide.statistics.ValueType#getColor` now returns String containing Web Color

### Server API changes

- Added `jetbrains.buildServer.serverSide.SProject#getPluginDataDirectory` that returns per-project plugin data directory
- `jetbrains.buildServer.serverSide.BuildTypeSettings#addBuildRunner` not accepts `jetbrains.buildServer.serverSide.BuildRunnerDescriptor` instead of `*S*BuildRunnerDescriptor`
- `jetbrains.buildServer.serverSide.TeamCityProperties` no longer contains static methods to compute TeamCity Data Directory. Use `jetbrains.buildServer.serverSide.ServerPaths` spring bean instead
- `jetbrains.buildServer.serverSide.buildDistribution.AagentsFilterContext` now contains `getCustomData` and `setCustomData` methods. Agent filters can now store data there to be used during distribution/filtering process
- added `jetbrains.buildServer.serverSide.buildDistribution.DefaultAgentsFilterContext`. Contains default implementation of custom data storage

### Authentication API changes

- Changes in `jetbrains.buildServer.serverSide.auth.LoginConfiguration` class:
  - `registerLoginModule(LoginModuleDescriptor)` method is deprecated, use `registerAuthMethodType(AuthMethodType)` instead
  - `getSelectedLoginModuleDescriptor()` method is deprecated, use `getConfiguredLoginModules()` instead
  - `createJAASConfiguration()` method is deprecated, use `createJAASConfiguration(AuthMethod)` instead
  - `getAuthType()` method now always returns the value "mixed" and is deprecated, use `getConfiguredAuthMethods(Class)` or `isAuthMethodConfigured(Class)` instead
- Changes in `jetbrains.buildServer.serverSide.auth.LoginModuleDescriptor` class:
  - `jetbrains.buildServer.serverSide.auth.LoginModuleDescriptorAdapter` class was added, extend your implementation from this class to not depend on future changes in `LoginModuleDescriptor`
  - `getOptions()` method is deprecated, you need to implement `getJAASOptions(Map)` method
  - `LoginModuleDescriptor` interface now extends `jetbrains.buildServer.serverSide.auth.AuthMethodType` interface and it contains some new methods you need to implement (or just use the adapter mentioned above)
- Changes in `javax.security.auth.spi.LoginModule` class:
  - Message from `javax.security.auth.login.FailedLoginException` thrown from `javax.security.auth.spi.LoginModule` is now visible to user as is on login page
  - Login module should now store own user name in TeamCity user's properties if it can differ from TeamCity's login. On login attempt login module must find existing user with the specified value of that property and return TeamCity's login for that user or return own user name if user does not exist yet. Use `jetbrains.buildServer.serverSide.auth.LoginModuleUtil#getUserModel(Map)` to get `jetbrains.buildServer.users.UserModel` in login module.
- You need now call `jetbrains.buildServer.serverSide.auth.ServerPrincipal#setCreatingNewUserAllowed(true)` if you want TeamCity to create the specified user in case he/she does not exist yet.

## VCS API changes

### General

- Non-required `VcsManager::registerVcsSupport` method have been removed.
- tests-related constructors from `jetbrains.buildServer.vcs.ModificationData` were moved to `jetbrains.buildServer.vcs.ModificationDataForTest`
- most methods from `jetbrains.buildServer.vcs.VcsSupportUtil` moved to parent class `jetbrains.buildServer.vcs.utils.VcsSupportUtil`
- `VcsException` class no longer have `setRoot`, `getRoot`, `prependMessage` methods that are not designed to be used for vcs-plugins, in core-related tasks use `jetbrains.buildServer.vcs.VcsRootVcsException`
- added method `jetbrains.buildServer.vcs.VcsSupportContext#getVcsExtension` for Vcs plugin context, override this method to provide additional services from plugin
- `jetbrains.buildServer.vcs.VcsSupport#ignoreServerCachesFor` no longer be called, please migrate to post TeamCity 4.5 API

### Patch building

- `jetbrains.buildServer.vcs.patches.PatchBuilder` no longer extends `jetbrains.buildServer.vcs.patches.PatchBuilderBase`. All methods from `jetbrains.buildServer.vcs.patches.PatchBuilderBase` were moved into `jetbrains.buildServer.vcs.patches.PatchBuilder`
- `jetbrains.buildServer.vcs.patches.PatchBuilderEx#setTimeStamp` was removed, use `jetbrains.buildServer.vcs.patches.PatchBuilder#setLastModified`
- `jetbrains.buildServer.vcs.patches.PatchBuilder` code was covered with `@NotNull/@Nullable` annotations

### Access to VCS Services

Vcs API is split into two parts: VCS plugin api, which is used to implement VCS services in TeamCity, and VCS usage api, or just VCS API, which is used to work with VCS services from within TeamCity.

- `jetbrains.buildServer.vcs.VcsManager#getAllVcs` is replaced with `jetbrains.buildServer.vcs.VcsManager#getAllVcsCore`
- Introduced `jetbrains.buildServer.vcs.VcsRootInstance#findService` method to obtain a `VcsService`
- `jetbrains.buildServer.vcs.VcsManager#getVcsUsernames` return type has changed from `VcsSupportContext` to `VcsSupportCore` in the key of the returned Map.

## Changes from 7.0 to 7.1

- new API calls `AgentRunningBuild#stopBuild` and `AgentRunningBuild#getInterruptReason()`. (Those methods were in `AgentRunningBuildEx` since 6.5)
- Responsibility API changes:
  - Added:

- jetbrains.buildServer.responsibility.ResponsibilityEntry
  - enum RemoveMethod
- jetbrains.buildServer.responsibility.ResponsibilityEntry
  - jetbrains.buildServer.serverSide.ResponsibilityInfo
  - jetbrains.buildServer.serverSide.ResponsibilityInfoData
  - jetbrains.buildServer.tests.TestResponsibilityData
    - getRemoveMethod()
- jetbrains.buildServer.responsibility.ResponsibilityEntryFactory
  - createEntry(BuildType)
- jetbrains.buildServer.responsibility.impl.BuildTypeResponsibilityEntryImpl
  - constructor(BuildType)
- jetbrains.buildServer.web.functions.user.ResponsibilityFunctions
  - isUserResponsible(ResponsibilityEntry, User)
- Changed:
  - jetbrains.buildServer.responsibility.impl.BuildTypeResponsibilityEntryImpl
    - constructor(BuildType, State, User, User, Date, String, RemoveMethod)
  - jetbrains.buildServer.responsibility.ResponsibilityEntryFactory
    - createEntry(BuildType, State, User, User, Date, String, RemoveMethod)
    - createEntry(TestName, long, State, User, User, Date, String, String, RemoveMethod)
  - jetbrains.buildServer.BuildType
    - getResponsibilityInfo() now returns ResponsibilityEntry
  - jetbrains.buildServer.serverProxy.RemoteBuildServer
    - updateResponsibility(Vector, String, String, String, String, String)
- Removed (deprecated):
  - jetbrains.buildServer.serverSide.ResponsibilityInfo
    - createInactive()
    - createInactive(String, boolean, User)
    - getSince()
    - getUser()
    - getUserWhoPerformsTheAction()
    - isActive()
    - isFixed()
    - setUser(User)
  - jetbrains.buildServer.serverSide.ResponsibilityInfoData
    - isActive()
    - isFixed()
  - jetbrains.buildServer.BuildType
    - removeResponsible(boolean, User, String)
    - setResponsible(User, String)
  - jetbrains.buildServer.serverProxy.RemoteBuildServer
    - removeResponsible(String, boolean, String)
    - removeResponsible(String, boolean, String, String)
    - resetResponsible(String, String)
    - resetResponsible(Vector, String, boolean, String, String, String)
    - setIsFixed(String, String, String)
    - setResponsible(String, String, String)
    - setResponsible(String, String, String, String)
    - setResponsible(Vector, String, String, String, String)
  - jetbrains.buildServer.serverSide.BuildServerListener
  - jetbrains.buildServer.serverSide.BuildServerAdapter
    - responsibleChanged(SBuildType, ResponsibilityInfo, ResponsibilityInfo, boolean)
  - jetbrains.buildServer.responsibility.SBuildTypeResponsibilityFacade
  - jetbrains.buildServer.responsibility.STestNameResponsibilityFacade
- Removed:
  - jetbrains.buildServer.serverSide.problems.BuildProblem and all implementations
  - jetbrains.buildServer.serverSide.problems.BuildProblemsProvider and all implementations
  - jetbrains.buildServer.serverSide.problems.BuildProblemsVisitor
  - jetbrains.buildServer.serverSide.SBuild
    - getBuildProblems()
    - visitBuildProblems(BuildProblemsVisitor)
- JavaScript: Activator is now BS.Activator and its source file has been moved from js/activation.js to js-bs/activation.js

## Changes from 6.5 to 7.0

- new API calls: `BuildStatistics.findTestBy(TestName)` and `BuildStatistics.getAllTests()`
- event-method `projectCreated` of `j.b.serverSide.BuildServerListener` and `j.b.serverSide.BuildServerAdapter` now receives two parameters: `projectId` and `user`.
- no longer publish `AntTaskExtension*`, `AntUtil`, `TestNGUtil`, `ElementPatch`, `JavaTaskExtensionHelper` classes to `openapi` package. Those classes can still be found in

<teamcity>/webapps/ROOT/WEB-INF/plugins/ant/agent/antPlugin.zip!antPlugin/ant-runtime.jar

- Notificator interface: methods `notifyResponsibleChanged` and `notifyResponsibleAssigned` changed second parameter from `j.b.serverSide.ResponsibilityInfo` to `j.b.responsibility.ResponsibilityEntry` (due to `ResponsibilityInfo` deprecation).
- `j.b.serverSide.BuildServerListener` - we've deprecated `responsibleChanged` method which used `j.b.serverSide.ResponsibilityInfo` parameter and added a similar method which uses `j.b.responsibility.ResponsibilityEntry`
- new API calls: `j.b.agent.AgentRunningBuild.getBuildFeatures()` and `j.b.agent.AgentRunningBuild.getBuildFeaturesOfType(String)`. With help of these methods you can access build features enabled for the current build with all parameters properly resolved.
- new API calls: `j.b.serverSide.BuildTypeSettings.isEnabled(String)` and `j.b.serverSide.BuildTypeSettings.setEnabled(String, boolean)`. These calls allow to enable / disable a setting with specified id (build runner, trigger or build feature), or check if it is enabled.
- Classes from `serviceMessages.jar` no longer depend on `j.b.messages.Status` class. If you used some of the classes (for example, `j.b.messages.serviceMessages.BuildStatus` class) and want to make your code compatible with TeamCity versions 6.0 - 7.0, please use `j.b.messages.serviceMessages.ServiceMessage.asString(...)` methods.
- new API extension point to filter all build messages: `j.b.messages.BuildMessagesTranslator`
- `j.b.serverSide.BuildServerListener` - we've removed `beforeBuildFinish(SRunningBuild, boolean)` method which was deprecated since TeamCity 3.1, there is another method `beforeBuildFinish(SRunningBuild)` which can be used instead.

## Changes from 6.0 to 6.5

- Classes `j.b.serverSide.TestBlockBean`, `j.b.serverSide.TestInProject`, `j.b.serverSide.FailedTestBean`, `j.b.TestNameBean` are removed from the Open API. Interfaces `j.b.serverSide.STest`, `j.b.serverSide.STestRun` should be used instead.
- `j.b.serverSide.ShortStatistics.getFailedTests()`, `j.b.serverSide.BuildStatistics.getIgnoredTests()` and `j.b.serverSide.BuildStatistics.getPassedTests()` return the list of `j.b.serverSide.STestRun` accordingly.
- Classes `j.b.tests.TestName` and `j.b.tests.SuiteTestName` are combined together into `j.b.tests.TestName`.

## Changes from 5.1.2 to 6.0

- `j.b.vcs.TriggerRules` class was removed from Open API as part of API cleanup. Please let us know if your plugin is affected by the change.

## New responsibility event methods added:

- `j.b.serverSide.BuildServerListener.responsibleChanged(SProject, Collection<SuiteTestName>, ResponsibilityEntry, boolean)`.
- `j.b.notification.Notificator.notifyResponsibleChanged(Collection<SuiteTestName>, ResponsibilityEntry, SProject, Set<SUser>)`, `j.b.notification.Notificator.notifyResponsibleAssigned(Collection<SuiteTestName>, ResponsibilityEntry, SProject, Set<SUser>)`.
- `j.b.notification.NotificationEventListener.responsibleChanged(SProject, Collection<SuiteTestName>, ResponsibilityEntry, boolean)`.
- `j.b.messages.ServiceMessageTranslator` is reworked to allow binding to arbitrary message type by name instead of only known types

Most methods in `j.b.agent.AgentLifeCycleListener` interface were extended to receive `j.b.agent.BuildRunnerContext`.

`j.b.agent.AgentLifeCycleListener#runnerFinished(...)` method added. It is called after build step is finished.

`j.b.agent.duplicates.DuplicatesReporter` and `j.b.duplicator.DuplicateInfo` are added for reporting code duplicates on agent side.

## Build Agent changes:

- `j.b.agent.AgentRunningBuild` does not extend `j.b.agent.AgentBuildInfo`, `j.b.agent.ResolvedParameters`. All methods from those interfaces were inlined into `AgentRunningBuild` interface.

Most methods from `j.b.agent.AgentRunningBuild` were splitted into `j.b.agent.BuildRunnerContext` and `j.b.agent.BuildContext`. We have added

Parameters required for build runner are represented with `j.b.agent.BuildRunnerContext` interface.

Every time `AgentRunningBuild` and `BuildRunnerContext` return resolved parameters back.

j.b.agent.BuildRunnerContext represents the context of current build runner. All add\* methods modifies context for the runner. Those changes will be reverted when context is switched to next runner.

j.b.agent.AgentRunningBuild provides a context of a build (i.e. shared between all runners). All addShared\* methods modifies the build context (and thus all build runner contexts).

j.b.agent.BuildAgentConfiguration now contains getBuildParameters() and getConfigParameters() methods to access parameters. Configuration parameters here are formed from properties from buildAgent.properties that does not start from 'system.' or 'env.' prefix. All parameters are returned with all references resolved.

j.b.agent.AgentBuildRunner#createBuildProcess method signature has been changed to receive j.b.agent.BuildRunnerContext.

j.b.agent.CommandLineBuildService#initialize(...) method signature has been changed to receive j.b.agent.BuildRunnerContext.

j.b.agent.CommandLineBuildService#getRunnerContext(...) added

j.b.agent.CommandLineBuildService#afterProcessSuccessfullyFinished() added

j.b.agent.BuildServiceAdapter is added to simplify as proposed base class for commandline base build runner service.

## Changes from 5.0 to 5.1

### Web extensions:

- deprecated method removed:  
j.b.web.openapi.WebControllerManager.addPageExtension(final WebPlace addTo, final WebExtension extension, Anchor<WebExtension> anchor)
- deprecated class removed: j.b.serverSide.Anchor
- deprecated class removed: j.b.notification.TemplatePatternProcessor; j.b.notification.TemplateProcessor added instead, see [Extending Notification Templates Model](#)
- method removed: j.b.notification.TemplateMessageBuilder.setPatternProcessor()
- several methods in j.b.serverSide.SBuildType now return boolean instead of void. You will probably need to recompile your plugins that use the interface.

## Changes from 4.5.5 to 5.0

### Parameters

j.b.serverSide.parameters.AbstractBuildParameterReferencesProvider is renamed to j.b.serverSide.parameters.AbstractBuildParametersProvider  
j.b.serverSide.parameters.BuildParameterReferencesProvider is renamed into j.b.serverSide.parameters.BuildParametersProvider  
BuildParameterReferencesProvider.getParameters(@NotNull final SBuild build) changed signature to getParameters(@NotNull final SBuild build, final boolean emulationMode)  
j.b.agent.BuildAgentConfiguration#getCacheDirectory now receives String as argument  
j.b.serverSide.buildDistribution.StartBuildPrecondition#canStart second parameters (Map<QueuedBuildInfo, BuildAgent>) may contain null values for some queued builds

### Miscellaneous

#### Added new build server events:

j.b.serverSide.BuildServerListener.vcsRootRemoved(SVcsRoot),  
j.b.serverSide.BuildServerListener.responsibleChanged(SProject, TestNameResponsibilityEntry,  
TestNameResponsibilityEntry, boolean)

#### Added three notification methods:

j.b.notification.Notifier.notifyResponsibleAssigned(SBuildType, Set<SUser>),  
j.b.notification.Notifier.notifyResponsibleChanged(TestNameResponsibilityEntry,  
TestNameResponsibilityEntry, SProject, Set<SUser>), j.b.notification.Notifier.notifyResponsibleAssigned(T  
estNameResponsibilityEntry, TestNameResponsibilityEntry, SProject, Set<SUser>)

## Changes prior to 4.5.5

### Not documented

preparationFinished

# Plugin Types in TeamCity

 TeamCity is hiring! Learn about [the available vacancies on the JetBrains site](#). Read about working in the TeamCity team.

TeamCity build system consists of two parts:

1. The server that gathers information while builds are running
2. Agents that run builds and send information to the server

Consequently, depending on where the code runs, there are

- server-side plugins
- agent-side plugins.

Besides that, plugins are divided into the following types :

- Build runners
- VCS plugins
- Notifiers
- User authentication plugins
- Build Triggers
- Extensions, which can modify some aspects of TeamCity behavior. There are several extension points on the server and on the agent, allowing you, for example, to format the stack trace on the web the way you need or modify the build status text. [Read more](#).

Plugins can also modify the TeamCity web UI. They can provide custom content to the existing pages (again, there are several extension points provided for that), or create new pages with their own UI. [Read more](#).

## Plugins Packaging

 TeamCity is hiring! Learn about [the available vacancies on the JetBrains site](#). Read about working in the TeamCity team.

This page is intended for plugin developers and explains how to package TeamCity plugins and agent tools. See [Installing Additional Plugins](#) and [Installing Agent Tools](#) for installation instructions.

On this page:

- [Introduction](#)
- [Plugins Location](#)
- [Plugins Loading](#)
- [Server-Side Plugins](#)
  - [Plugin Structure](#)
    - [Web resources packaging](#)
  - [Plugin Descriptor](#)
- [Agent-Side Plugins](#)
  - [Plugin Structure](#)
    - [Deprecated Plugin Structure](#)
    - [New Plugins](#)
  - [Plugin Descriptor](#)
    - [Plugins](#)
    - [Tools](#)
      - [Making File Executable](#)
- [Plugin Dependencies](#)
- [Agent Upgrade on Updating Plugins](#)

### Introduction

To write a TeamCity plugin, the knowledge of [Spring Framework](#) is beneficial.

There are [server-side](#) and [agent-side](#) plugins in TeamCity. Server-side and agent-side plugins are initialized in their own Spring containers; this means that every plugin needs a Spring bean definition file describing the main services of the plugin. Bean definition files are to be placed into the `META-INF` folder of the JAR archive containing the plugin classes.

There is a convention for naming the definition file:

- `build-server-plugin-<plugin name>*.xml` — for server-side plugins
- `build-agent-plugin-<plugin name>*.xml` — for agent-side plugins

where the asterisk can be replaced with any text, for example: `build-server-plugin-cvs.xml`.

**i** If you want to get started with an empty plugin quickly, try the template plugin in the JetBrains Subversion repository <http://svn.jetbrains.org/teamcity/plugins/template-plugin/templateProject>. Refer to `readme.txt` for instructions.

## Plugins Location

TeamCity is able to load plugin from the following directories:

- `<TeamCity data directory>/plugins` – **user-installed** plugins
- `<TeamCity web application>/WEB-INF/plugins` — default directory for bundled TeamCity plugins

Plugins with the same name (for example, a newer version) located in `<TeamCity data directory>/plugins` will override the plugins in the `<TeamCity web application>/WEB-INF/plugins` directory.

## Plugins Loading

TeamCity creates a child Spring Framework context per plugin. There are two options to load plugins classes: standalone and shared:

- Standalone classloading (recommended) allows loading every plugin to a separate classloader. This approach allows a plugin to have additional libraries without the risk of affecting the server or other plugins.
- Shared classloading allows loading all plugins into same classloader. It is not allowed to override any libraries here.

You may specify desired the classloading mode in the `teamcity-plugin.xml` file, see the [section below](#).

**!** The TeamCity plugin loader supports plugin dependencies, described [below](#).

**!** To load your plugin, the server must be restarted.

## Server-Side Plugins

A server-side plugin may affect the server only, or may include a number of agent-side plugins that will be automatically distributed to all build agents.

## Plugin Structure

A plugin can be a zip archive (recommended) or a separate folder.

If you use a zip file:

- TeamCity will use the name of the zip file as the plugin name
- The plugin zip file will be automatically unpacked to a temporary directory on the server start-up

If you use a separate folder:

- TeamCity will use the folder name as the plugin name

The plugin zip archive/directory includes:

- `teamcity-plugin.xml` containing meta information about the plugin, like its name and version, see the [section below](#).
- the `server` directory containing the server-side part of the plugin, i.e., a number of jar files.
- the `agent` directory containing `<agent plugin zip>` if your plugin affects agents too, see the [section below](#).

The plugin directory should have the following structure:

The server-only plugin:

```
server
  |
--> <server plugin jar files>
teamcity-plugin.xml
```

The plugin affecting the server and agents:

```

agent
|
--> <agent plugin zip files> (see [below]#agentDirectory)
server
|
--> <server plugin jar files>
teamcity-plugin.xml

```

## Web resources packaging

In most cases a plugin is just a number of classes packed into a JAR file.

If you wish to write a custom page for TeamCity, most likely you'll need to place images, CSS, JavaScript, JSP files or binaries somewhere. The files that you want to access via hyperlinks and JSP pages are to be placed into the `buildServerResources` subfolder of the plugin's .jar file. Upon the server startup, these files will be extracted from the archive. You may use `jetbrains.buildServer.web.openapi.PluginDescriptor` spring bean to get the paths to the extracted resources ([read more](#) on how to construct paths to your JSP files).

It is a good practice to put all resources into a separate.jar file.

## Plugin Descriptor

The `teamcity-plugin.xml` file must be located in the root of the plugin directory or .zip file. You can refer to the XSD schema for this file which is unpacked to `<TeamCity data directory>/config/teamcity-plugin-descriptor.xsd`

An example of `teamcity-plugin.xml`:

```

<?xml version="1.0" encoding="UTF-8"?>
<teamcity-plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:noNamespaceSchemaLocation="urn:shemas-jetbrains-com:teamcity-plugin-v1-xml">
    <info>
        <name>PluginName</name> <!-- the name of plugin used in teamcity -->
        <display-name>This name may be used for UI</display-name>
        <description>Some description goes here</description>
        <version>0.239.42</version>
    </info>
    <requirements min-build="46654" max-build="57000" /> <!-- specify compatible TeamCity server
builds -->
    <deployment use-separate-classloader="true" /> <!-- load server plugin's classes in separate
classloader-->
    <parameters>
        <parameter name="key">value</parameter>
        <!-- ... -->
    </parameters>
</teamcity-plugin>

```

 It is recommended to set the `use-separate-classloader="true"` parameter to `true` for server-side plugins.

The plugin parameters can be accessed via the `jetbrains.buildServer.web.openapi.PluginDescriptor#getParameterValue(String)` method.

## Agent-Side Plugins

TeamCity build agents support the following plugin structures:

- new plugins (with the `teamcity-plugin.xml` descriptor), including tool plugins
  - tool plugins (with the `teamcity-plugin.xml` descriptor). This is a kind of plugin without any classes loaded into the runtime. Tool plugins for agents are used to only distribute binary files to agents, e.g. the NuGet plugin for

TeamCity creates a tool plugin for agents to redistribute the downloaded NuGet.exe to TeamCity agents. See more at [Installing Agent Tools](#).

- deprecated plugins (with the plugin name folder in the .zip file)

## Plugin Structure

The `agent` directory must have one file only: <agent plugin zip> structured the following way:

### Deprecated Plugin Structure

The old plugin structure implied that all plugin files and directories were placed into the single root directory, i.e. there had to be one root directory in the archive, the \_<plugin name directory>\_, and no other files at the top level. All .jar files required by the plugin on agents were placed into the `lib` subfolder:

```
<plugin name directory>
  |
  --> lib
  |
  --> <jar files>
```

There must be no other items in the root of .zip but the directory with the plugin name. TeamCity build agent detects and loads such plugins using the shared classloader.

### New Plugins

Now a new, more flexible schema of packing is recommended. The plugin name root directory inside the plugin archive is no longer required. The agent plugin name now is obtained from the `PluginName.zip` file name. The archive needs to include the plugin descriptor, `teamcity-plugin.xml`, see below.

```
agent-plugin-name.zip
  |
  - teamcity-plugin.xml
  - lib
  |
  plugin.jar
  plugin.lib
```

## Plugin Descriptor

It is required to have the `teamcity-plugin.xml` file under the root of the agent plugin .zip file. The agent tries to validate the plugin-provided `teamcity-plugin.xml` file against the xml schema. If `teamcity-plugin.xml` is not valid, the plugin will be loaded, but some data from the descriptor may be lost.

### Plugins

This `teamcity-plugin.xml` file provides the plugin description (same as it is done on the server-side):

```
<?xml version="1.0" encoding="UTF-8"?>
<teamcity-agent-plugin
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="urn:schemas-jetbrains-com:teamcity-agent-plugin-v1-xml">
  <plugin-deployment use-separate-classloader="true"/>
</teamcity-agent-plugin>
```

## Tools

To deploy a tool, use the following `teamcity-plugin.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<teamcity-agent-plugin
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:schemas-jetbrains-com:teamcity-agent-plugin-v1-xml">
    <tool-deployment/>
</teamcity-agent-plugin>
```

#### Making File Executable

There is experimental ability (can be removed in the future versions!) to set executable bit to some files after unpacking on the agent. Watch [TW-21673](#) for proper solution.

To make some files of a tool executable, use the following `teamcity-plugin.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<teamcity-agent-plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:schemas-jetbrains-com:teamcity-agent-plugin-v1-xml">
    <tool-deployment>
        <layout>
            <executable-files>
                <include name='path_to_a_file' />
            </executable-files>
        </layout>
    </tool-deployment>
</teamcity-agent-plugin>
```

where `<include name='path_to_a_file' />` relative to your tool folder ( e.g. `<Agent home>/tools/<your tool name>`) specifies the list of files to be made executable on Linux/Unix/Mac.  Note that wildcards are not supported.

See [Installing Agent Tools](#) for installation instructions.

#### Plugin Dependencies

Plugin dependencies are present on both the server and agent side: some components are separated from the core into separate bundled plugins: Ant runner, IDEA runner, .NET runners, JUnit, and TestNG support. If you need some functionality of one of these plugins, use the plugin dependencies feature.

To use plugin dependencies, add the `dependencies` tag into the plugin xml descriptor:

Example of the server-side plugin descriptor using plugin dependencies:

```
<?xml version="1.0" encoding="UTF-8"?>
<teamcity-plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:schemas-jetbrains-com:teamcity-plugin-v1-xml">
    <info>
        <name>Plugin Name</name>
        <!-- Some tags skipped -->
    </info>
    <deployment use-separate-classloader="true"/>
    <dependencies>
        <plugin name="dotNetRunners"/>
    </dependencies>
</teamcity-plugin>
```

Example of agent-side plugin descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<teamcity-agent-plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:schemas-jetbrains-com:teamcity-agent-plugin-v1-xml">
<plugin-deployment use-separate-classloader="true"/>
<dependencies>
<tool name="ant"/>
<plugin name="ant-runner"/>
</dependencies>
</teamcity-agent-plugin>
```



Using separate classloader is required (and will be enforced) to use dependencies.  
Transitive dependencies are not supported, you should specify all dependencies.

The names of the bundled tools and plugins are just the names of the corresponding folders in `TeamCity Home/webapps/ROOT/WEB-INF/plugins` for the server-side plugins and `<Agent home>/plugins/` or `<Agent home>/tools/` for the agent-side plugins and tools.



Some bundled plugin names may change in future releases.

## Agent Upgrade on Updating Plugins

TeamCity server monitors all agent plugins .zip files for a change (plugin files changed, added or removed). Once a change is detected, agents receive the upgrade command from the server and download the updated files automatically. It means that if you want to deploy an updated agent part of your plugin without the server restart, you can put your agent plugin into this folder.

After a successful upgrade, your plugin will be unpacked into the `<Agent home>/plugins/` or `<Agent home>/tools/` folders. Note that if an agent is busy running a build, it will upgrade only after the build finishes. No new builds will start on the agent if it is to be upgraded.

## Server-side Object Model



TeamCity is hiring! Learn about the available vacancies on the JetBrains site. Read about working in the TeamCity team.

### Project model

The main entry point for project model is `jetbrains.buildServer.serverSide.ProjectManager`. With help of this class you can obtain projects and build configurations, create new projects or remove them.

On the server side projects are represented by `jetbrains.buildServer.serverSide.SProject` interface. Project has unique id (`projectId`). Any change in the project will not be persisted automatically. If you need to persist project configuration on disk use `SProject.persist()` method.

Build configurations are represented by `jetbrains.buildServer.serverSide.SBuildType`. As with projects any change in the build configuration settings is not saved on disk automatically. Since build configurations are stored in the projects, you should persist corresponding project after the build configuration modification.



Note: interfaces available on the server side only have prefix S in their names, like `SProject`, `SBuildType` and so on.

### Build lifecycle

When build is triggered it is added to the build queue (`jetbrains.buildServer.serverSide.BuildQueue`). While staying in the queue and waiting for a free agent it is represented by `jetbrains.buildServer.serverSide.SQueuedBuild` interface. Builds in the queue can be reordered or removed. To add new build in the queue use `addToQueue()` method of the `jetbrains.buildServer.serverSide.SBuildType`.

A separate thread periodically tries to start builds added to the queue on a free agent. A started build is removed from the queue and appears in the model as `jetbrains.buildServer.serverSide.SRunningBuild`. After the build finishes it becomes `jetbrains.buildServer.serverSide.SFinishedBuild` and is added to the build history. Both `SRunningBuild` and `SFinishedBuild` extend

common interface: `jetbrains.buildServer.serverSide.SBuild`.

There is another entity `jetbrains.buildServer.serverSide.BuildPromotion` which is associated with build during the whole build lifecycle. This entity contains all information necessary to reproduce this build, i.e. build parameters (properties and environment variables), VCS root revisions, VCS root settings with checkout rules and dependencies. `BuildPromotion` can be obtained on the any stage: when build is in the queue, running or finished, and it always be the same object.

## Accessing builds

A started build (running or finished) can be found by its' id (`buildId`). For this you should use `jetbrains.buildServer.serverSide.SBuildServer#findBuildInstanceById(long)` method.

It is also possible to find build in the build history, or to retrieve all builds from the history. Take a look at `SBuildType#getHistory()` method and at `jetbrains.buildServer.serverSide.BuildHistory` service.



Note: if not mentioned specifically the returned collections of builds are always sorted by start time in reverse order, i.e. most recent build comes first.

## Listening for server events

A lot of events are generated by the server during its lifecycle, these are events like `buildStarted`, `buildFinished`, `changeAdded` and so on. Most of these events are defined in the `jetbrains.buildServer.serverSide.BuildServerListener` interface. There is corresponding adapter class `jetbrains.buildServer.serverSide.BuildServerAdapter` which you can extend.

To register your listener you should obtain reference to `EventDispatcher<BuildServerListener>`. Since this dispatcher is defined as a Spring bean, you can obtain reference with help of Spring autowiring feature.

## User model events

You can also watch for events from TeamCity user model. For example, you can track new user accounts registration, removing of the users or changing of the user settings. You should use `jetbrains.buildServer.serverSide.UserModelListener` interface and register your listeners in the `jetbrains.buildServer.users.UserModel`.

## VCS changes

TeamCity server constantly polls version control systems to determine whether a new change occurred. Polling is done per VCS root (`jetbrains.buildServer.vcs.SVcsRoot`). Each VCS root has unique id, VCS specific properties, scope (shared or project local) and version. Every change in VCS root creates a new version of the root, but VCS root id remains the same. VCS roots can be obtained from `SBuildType` or found by id with help of `jetbrains.buildServer.vcs.VcsManager`.

A change is represented by `jetbrains.buildServer.vcs.SVcsModification` class. Each detected change has unique id and is associated with concrete version of the VCS root. A change also belongs to one or more build configurations (these are build configurations where VCS root was attached when change was detected), see `getRelatedConfigurations()` method.

There are several methods allowing to obtain VCS changes:

1. `SBuildType#getPendingChanges()` - use this method to find pending changes of the some build configuration (i.e. changes which are not yet associated with a build)
2. `SBuild#getContainingChanges()` - use this method to obtain changes associated with a build, i.e. changes since previous build
3. `jetbrains.buildServer.vcs.VcsModificationHistory` - use this service to obtain arbitrary changes stored in the changes history, find change by id and so on.



Note: if not mentioned specifically the returned collections of changes are always sorted in reverse order, with the most recent change coming first.

## Agents

Agent is represented by `jetbrains.buildServer.serverSide.SBuildAgent` interface. Agents have unique id and name, and can be found by name or by id with help of `jetbrains.buildServer.serverSide.BuildAgentManager`. Agent can have various states:

1. registered / unregistered: agent is registered if it is connected to the server.
2. authorized / unauthorized: authorized agent can run builds, unauthorized can't. It is impossible to run build on unauthorized agent even manually. A number of authorized agents depends on entered license keys.
3. enabled / disabled: builds won't run automatically on disabled agents, but it is possible to start build manually on such agent if user has required permission.
4. outdated / up to date: agent is outdated if its' version does not match server version or if some of its' plugins should be updated. New builds will not start on an outdated agent until it upgrades, but already running builds will continue to run

as usual.

## Agent-side Object Model

 TeamCity is hiring! Learn about the available vacancies on the JetBrains site. Read about working in the TeamCity team.

On the agent side agent is represented by `jetbrains.buildServer.agent.BuildAgent` interface. `BuildAgent` is available as a Spring bean and can be obtained by autowiring.

Build agent configuration can be read from the `jetbrains.buildServer.agent.BuildAgentConfiguration`, it can be obtained from the `BuildAgent#getConfig()` method.

### Agent side events

There is `jetbrains.buildServer.agent.AgentLifeCycleListener` interface and corresponding adapter class `jetbrains.buildServer.agent.AgentLifeCycleAdapter` which can be used to receive notifications about agent side events, like starting of the build, build finishing and so on. Your listener must be registered in the `jetbrains.buildServer.util.EventDispatcher`. This service is also defined in the Spring context.

### Build

Each build on the agent is represented by `jetbrains.buildServer.agent.AgentRunningBuild` interface. You can obtain instance of `AgentRunningBuild` by listening for `buildStarted(AgentRunningBuild)` event in `AgentLifeCycleListener`.

### Logging to build log

Messages to build log can be sent only when a build is running. Internally agent sends messages to server by packing them into the `jetbrains.buildServer.messages.BuildMessage1` structures. However instead of creating `BuildMessage1` structures it is better and easier to use corresponding methods in `jetbrains.buildServer.agent.BuildProgressLogger` which can be obtained from the `AgentRunningBuild`.

If you want to construct your own messages you can use static methods of `jetbrains.buildServer.messages.DefaultMessagesInfo` class for that.

## Extensions

 TeamCity is hiring! Learn about the available vacancies on the JetBrains site. Read about working in the TeamCity team.

Extension in TeamCity is a point where standard TeamCity behavior can be changed. There are three marker interfaces for TeamCity extensions:

- `jetbrains.buildServer.serverSide.ServerExtension`
- `jetbrains.buildServer.agent.AgentExtension`
- `jetbrains.buildServer.TeamCityExtension`

Extension interface implements one of these marker interfaces. `ServerExtension` and `AgentExtension` are used to mark server and agent side extensions correspondingly. `TeamCityExtension` is the base interface for `ServerExtension` and `AgentExtension`. Thus you can take a list of all available extensions in TeamCity by taking a look at interfaces which extend these marker interfaces.

### Registering custom extension

There are two ways to register custom extension:

1. define a bean in the Spring context which implements extension interface, in this case your extension will be loaded automatically
2. register your extension at runtime in the `jetbrains.buildServer.ExtensionHolder` service (can be obtained by Spring autowiring feature)

### Available extensions

#### Server-side extensions

Extension	Since	Description
-----------	-------	-------------

jetbrains.buildServer.serverSide.TextStatusBuilder	3.0	Allows customizing text status line of the build, i.e. the build description which usually contains text like "Tests passed: 234, failed: 4 (2 new)".
jetbrains.buildServer.serverSide.TriggeredByProcessor	4.0	Similar to TextStatusBuilder but affects "Triggered by" value shown in the UI.
jetbrains.buildServer.serverSide.FailedTestOutputFormatter	4.0	This extension allows applying custom formatting to test stacktrace to be shown in the UI.
jetbrains.buildServer.serverSide.buildDistribution.StartBuildPrecondition	4.5	Allows defining preconditions for starting a build on an agent, that is, you can instruct TeamCity to delay a build till some condition is met.
jetbrains.buildServer.serverSide.GeneralDataCleaner	2.0	This extension is called when the cleanup process is going to finish, plugins can clean their data with the help of this extension.
jetbrains.buildServer.serverSide.DataCleaner	2.0	This extension is called when the cleanup process is going to clean up data of a build, plugins can remove their data associated with this build with help of this extension.
jetbrains.buildServer.serverSide.ParametersPreprocessor	3.0	Allows modifying build parameters right before they are sent to an agent.
jetbrains.buildServer.serverSide.parameters.BuildParametersProvider	5.0	Allows adding additional parameters available for a build. It differs from ParametersPreprocessor in the way that the parameters added by BuildParametersProvider will be available in a popup showing available parameters, and will be considered when requirements are calculated.
jetbrains.buildServer.serverSide.parameters.ParameterDescriptionProvider	5.0	Provides a human-readable description for a parameter, see also BuildParametersProvider.
jetbrains.buildServer.messages.serviceMessages.ServiceMessageTranslator	4.0	Translator for specific type of service messages.
jetbrains.buildServer.usageStatistics.UsageStatisticsProvider	6.0	Provides a custom usage statistics.

See also:

[Extending TeamCity: Developing TeamCity Plugins | Web UI Extensions](#)

## Web UI Extensions



TeamCity is hiring! Learn about [the available vacancies on the JetBrains site](#). [Read](#) about working in the TeamCity team.

This section covers:

- [Getting Started](#)
- [Under the Hood](#)
- [Developing a Custom Controller](#)
- [Obtaining paths to JSP files](#)
- [Classes and interfaces from TeamCity web open API](#)



Hint: you can use source code of the existing plugins as a reference, for example:

- <http://www.jetbrains.net/confluence/display/TW/Server+Profiling>

## Getting Started

The simplest way of adding your own custom tab is to derive from one of the classes:

- `jetbrains.buildServer.web.openapi.project.ProjectTab`
- `jetbrains.buildServer.web.openapi.buildType.BuildTypeTab`
- `jetbrains.buildServer.web.openapi.ViewLogTab`
- `jetbrains.buildServer.controllers.admin.AdminPage`

This will add your tab to the project, build type (build configuration), build or administration page respectively. Here's an example of the Diagnostics admin page:

```
public class DiagnosticsAdminPage extends AdminPage {
    public DiagnosticsAdminPage(@NotNull PagePlaces pagePlaces, @NotNull PluginDescriptor descriptor) {
        super(pagePlaces);
        setPluginName("diagnostics");
        setIncludeUrl(descriptor.getPluginResourcesPath("/admin/diagnosticsAdminPage.jsp"));
        setTabTitle("Diagnostics");
        setPosition(PositionConstraint.after("clouds", "email", "jabber"));
        register();
    }

    @Override
    public boolean isAvailable(@NotNull HttpServletRequest request) {
        return super.isAvailable(request) && checkHasGlobalPermission(request,
Permission.CHANGE_SERVER_SETTINGS);
    }

    @NotNull
    public String getGroup() {
        return SERVER RELATED GROUP;
    }
}
```

There are a couple of things to note here:

- it is important to call "register" method; ProjectTab, BuildTypeTab and ViewLogTab will do that for you automatically, AdminPage won't, that's why the call is there;
- TeamCity might have difficulties with finding your resources (JSP, CSS, JS) if you don't refer to your resources through the PluginDescriptor.
- the page above doesn't provide any model to the JSP. If you need one, just override the "fillModel" method.

Here's another example of the project tab:

```

public class CurrentProblemsTab extends ProjectTab {
    public CurrentProblemsTab(@NotNull PagePlaces pagePlaces,
                             @NotNull ProjectManager projectManager,
                             @NotNull PluginDescriptor descriptor) {
        super("problems", "Current Problems", pagePlaces, projectManager,
              descriptor.getPluginResourcesPath("problems.jsp"));
        // add your CSS/JS here
    }

    @Override
    protected void fillModel(@NotNull Map<String, Object> model, @NotNull HttpServletRequest request,
                            @NotNull SProject project, @Nullable SUser user) {
        // add your data here
    }
}

```

That's it! Just specify your tab as a Spring bean, and you'll be able to see your tab in TeamCity.

 We are using [Spring MVC](#) web framework.

## Under the Hood

If you download and take a look at the TeamCity open API sources, you'll notice that all tabs above derive from the `jetbrains.buildServer.web.openapi.SimpleCustomTab`. And the only major difference between them all is a `jetbrains.buildServer.web.openapi.PlaceId` they specify in constructor.

Here's what they use:

- `PlaceId.PROJECT_TAB`
- `PlaceId.BUILD_CONF_TAB`
- `PlaceId.BUILD_RESULTS_TAB`
- `PlaceId.ADMIN_SERVER_CONFIGURATION_TAB`

Don't get confused by the variety of names, it's a long story. The main thing is there are more than 30 other place ids that you can hook into!

- `PlaceId.ALL_PAGES_HEADER`
- `PlaceId.AGENT_DETAILS_TAB`
- `PlaceId.LOGIN_PAGE`
- ...

There is a convention that a place id named as a TAB can be used with the `SimpleCustomTab`. Others cannot, and to use them you will have to deal with low level `jetbrains.buildServer.web.openapi.SimplePageExtension`. But that's pretty much the only change, take a look at the example:

```

public class ChangedFileExtension extends SimplePageExtension {
    public ChangedFileExtension(@NotNull PagePlaces pagePlaces,
                               @NotNull PluginDescriptor descriptor) {
        super(pagePlaces, PlaceId.CHANGED_FILE_LINK, "changeViewers",
              descriptor.getPluginResourcesPath("changedFileLink.jsp"));
        register();
    }

    @Override
    public boolean isAvailable(@NotNull HttpServletRequest request) {
        return super.isAvailable(request);
    }

    @Override
    public void fillModel(@NotNull Map<String, Object> model, @NotNull HttpServletRequest request) {
        // fill model
    }
}

```

This extension provides a custom HTML (usually a link) near the each file in the modification's list.

We use it to add "Open in IDE", "Open in External Change Viewer", etc links.

In this particular case the file itself is passed via "changedFile" attribute of the request, but this is different for different extensions.

A couple of useful notes:

- `isAvailable(HttpServletRequest)` method is called to determine whether page extension content should be shown or not.
- in case `isAvailable(HttpServletRequest)` is true, the `fillModel(Map, HttpServletRequest)` method will always be called and JSP will be rendered in UI. You cannot abort the process after `isAvailable(HttpServletRequest)` is done, that's why it's usually inconvenient to handle POST requests in extensions. Use a custom controller for that (see below).
- One more case when you might need a custom controller is when you need to process HTTP response manually, e.g. stream a file content. `fillModel(Map, HttpServletRequest)` won't allow you to do that.

## Developing a Custom Controller

Sometimes page extensions provide interaction with user and require communication with server. For example, your page extension can show a form with a "Submit" button. In this case in addition to writing your own page extension, you should provide a controller which will process requests from such forms, and use path to this controller in the form action attribute (the path is a part of URL without context path and query string).

Example:

```

public class ServerConfigGeneralController extends BaseFormXmlController {
    public ServerConfigGeneralController(@NotNull SBuildServer server,
                                         @NotNull WebControllerManager webControllerManager) {
        super(server);
        webControllerManager.registerController("/my/path/", this);
    }

    @Override
    @Nullable
    protected ModelAndView doGet(@NotNull final HttpServletRequest request, @NotNull final
HttpServletResponse response) {
        return null;
    }

    @Override
    protected void doPost(@NotNull final HttpServletRequest request, @NotNull final
HttpServletResponse response, @NotNull final Element xmlResponse) {
        return null;
    }
}

```

To simplify things your controller can extend our `jetbrains.buildServer.controllers.BaseController` class and implement `BaseController.doHandle(HttpServletRequest, HttpServletResponse)` method.

With the custom controller you can provide completely new pages.

#### Obtaining paths to JSP files

Plugin resources are unpacked to <TeamCity web application>/plugins directory when server starts. However to construct paths to your JSP or images in Java it is recommended to use `jetbrains.buildServer.web.openapi.PluginDescriptor`. This descriptor can be obtained as any other Spring service.

In JSP files to construct paths to your resources you can use  `${teamcityPluginResourcesPath}`. This attribute is provided by TeamCity automatically, you can use it like this:

```

```

Note: `<c:url/>` is required to construct correct URL in case if TeamCity is deployed under the non root context.

#### Classes and interfaces from TeamCity web open API

Class / Interface	Description
<code>jetbrains.buildServer.web.openapi.PlaceId</code>	A list of page place identifiers / extension points
<code>jetbrains.buildServer.web.openapi.PagePlace</code>	A single page place associated with PlaceId, allows to add / remove extensions
<code>jetbrains.buildServer.web.openapi.PageExtension</code>	Page extension interface
<code>jetbrains.buildServer.web.openapi.SimplePageExtension</code>	Base class for page extensions
<code>jetbrains.buildServer.web.openapi.CustomTab</code>	Custom tab extension interface
<code>jetbrains.buildServer.web.openapi.PagePlaces</code>	Maintains a collection of page places and allows to locate PagePlace by PlaceId

<code>jetbrains.buildServer.web.openapi.WebControllerManager</code>	Maintains a collection of custom controllers, allows to register custom controllers
<code>jetbrains.buildServer.controllers.BaseController</code>	Base class for controllers

## Plugin Settings

 TeamCity is hiring! Learn about [the available vacancies on the JetBrains site](#). [Read](#) about working in the TeamCity team.

### Server-wide settings

A plugin can store server-wide setting in the `main-config.xml` file (stored in `TEAMCITY_DATA_PATH/config` directory). To use this file, the plugin should register an [extension](#) which implements `jetbrains.buildServer.serverSide.MainConfigProcessor`. This interface has methods which allow loading and saving some data in the XML format (via JDOM). Please note, that the plugin will be asked to reinitialize data if the file has been changed on the disk while TeamCity is up and running.

### Project-wide settings

Per-project settings can be stored in the `TEAMCITY_DATA_PATH/config/<project-name>/plugin-settings.xml` directory.

To manage the settings in this file, you should implement a `jetbrains.buildServer.serverSide.settings.ProjectSettingsFactory` interface and register this implementation in `jetbrains.buildServer.serverSide.settings.ProjectSettingsManager` (which can be obtained via the constructor injection). Upon registration, you should specify the name of the XML node under where the settings will be stored.

Your settings should be serialized to the XML format by your implementation of the `jetbrains.buildServer.serverSide.settings.ProjectSettings` interface. The `{readFrom}` and `writeTo` methods should be implemented consistently.

When your code needs the stored XML settings, they should be loaded via `ProjectSettingsManager#getSettings` call. Your registered factory will create these settings in memory.

You can save this project's settings explicitly via the `jetbrains.buildServer.serverSide.SProject#persist()` call, or via `ProjectManager#persistAllProjects`. This can be done, for instance, upon some event (see `jetbrains.buildServer.serverSide.BuildServerAdapter#serverStartup()`).

## Development Environment

 TeamCity is hiring! Learn about [the available vacancies on the JetBrains site](#). [Read](#) about working in the TeamCity team.

### Plugin Debugging

You can debug your plugin in a running TeamCity just like a regular Java application debug: start TeamCity server with debug-enabling JVM options and then connect to a remote debug port from the IDE. If you start TeamCity server from outside of your IDE, in IntelliJ IDE you can use "Remote" run configuration, check related [external blog post](#). The JVM options for the server can be set via `TEAMCITY_SERVER_OPTS` environment variable.

### Plugin Reloading

If you make changes to a plugin, you will generally need to shut down the server, update the plugin, and start the server again.

To enable TeamCity development mode, pass the `"teamcity.development.mode=true"` [internal property](#). Using the option you will:

- Enforce application server to quicker recompile changed `.jsp` classes
- Disable JS and CSS resources merging/caching

The following hints can help you eliminate the restart in the certain cases:

- if you do not change code affecting plugin initialization and change only body of the methods, you can attach to the server process with a debugger and use Java hotswap to reload the changed classes from your IDE without web server restart. Note that the standard hotswap does not allow you to change method signatures.
- if you make a change in some resource (`jsp`, `js`, `images`) you can copy the resources to `webapps/ROOT/plugins/<plugin-name>` directory to allow Tomcat to reload them.

- change in build agent part of plugin will initiate build agents upgrade.

If you replace a deployed plugin .zip file with changed class files while TeamCity server is running, this can lead to NoClassDefFound errors.

To avoid this, set "teamcity.development.shadowCopyClasses=true" internal property. This will result in:

- creating ".teamcity\_shadow" directory for each plugin .jar file;
- avoid .jar files update on plugin archive change.

See also:

[Extending TeamCity: Developing TeamCity Plugins | Plugins Packaging](#)

## Developing Plugins Using Maven

 TeamCity is hiring! Learn about [the available vacancies on the JetBrains site](#). [Read about working in the TeamCity team](#).

You can easily develop TeamCity plugins with Maven.

On this page:

- [Supported Maven versions](#)
- [Open API in Maven Repository](#)
- [Maven Archetypes](#)
- [TeamCity SDK Maven plugin](#)

### Supported Maven versions

Both Maven 2 (2.2.1+) and Maven 3 (3.0.4+) are supported.

### Open API in Maven Repository

TeamCity Open API is available as a set of Maven artifacts residing in the JetBrains Maven repository (<http://download.jetbrains.com/teamcity-repository>). Add this fragment to the <repositories> section of your pom file to access it:

```
<repository>
  <id>jetbrains-all</id>
  <url>http://download.jetbrains.com/teamcity-repository</url>
</repository>
```

Please note that only open API artifacts are present in the repository. If your plugin needs to use the not-open API, the corresponding jars should then be added to the project from the TeamCity distribution as they are not provided in the repository.

The open API in the repository is split into two main parts:

The server-side API:

```
<dependency>
  <groupId>org.jetbrains.teamcity</groupId>
  <artifactId>server-api</artifactId>
  <version>10.0</version>
  <scope>provided</scope>
</dependency>
```

The agent-side API:

```
<dependency>
<groupId>org.jetbrains.teamcity</groupId>
<artifactId>agent-api</artifactId>
<version>10.0</version>
<scope>provided</scope>
</dependency>
```

**i** Note that API dependencies are used with the `provided` scope. This way you will avoid adding the API and its transitive dependencies to the target distribution.

There is also an artifact to support plugin tests:

```
<dependency>
<groupId>org.jetbrains.teamcity</groupId>
<artifactId>tests-support</artifactId>
<version>10.0</version>
<scope>test</scope>
</dependency>
```

## Maven Archetypes

For a quick start with a plugin, there are three [Maven archetypes](#) in the `org.jetbrains.teamcity.archetypes` group:

- `teamcity-plugin` - an empty plugin, includes both the server and the agent plugin parts
- `teamcity-server-plugin` - an empty plugin, includes the server plugin part only
- `teamcity-sample-plugin` - the plugin with the sample code (adds a "Click me" button to the bottom of the TeamCity project Overview page)

Different released versions of the TeamCity server API are listed [here](#).

Here is the Maven command that will generate a project for a server-side-only plugin depending on 10.0 TeamCity version:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
-DarchetypeRepository=http://download.jetbrains.com/teamcity-repository
-DarchetypeArtifactId=teamcity-server-plugin -DarchetypeGroupId=org.jetbrains.teamcity.archetypes
-DarchetypeVersion=RELEASE -DteamcityVersion=10.0
```

Here is the Maven command that will generate a project that contains both, the server and agent parts of a plugin and depends on 10.0 TeamCity version:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
-DarchetypeRepository=http://download.jetbrains.com/teamcity-repository
-DarchetypeArtifactId=teamcity-plugin -DarchetypeGroupId=org.jetbrains.teamcity.archetypes
-DarchetypeVersion=RELEASE -DteamcityVersion=10.0
```

Here is the Maven command that will generate a sample project on 10.0 TeamCity version:

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate
-DarchetypeRepository=http://download.jetbrains.com/teamcity-repository
-DarchetypeArtifactId=teamcity-sample-plugin -DarchetypeGroupId=org.jetbrains.teamcity.archetypes
-DarchetypeVersion=RELEASE -DteamcityVersion=10.0
```

You will be asked to enter the usual Maven `groupId`, `artifactId` and `version` for your plugin. Please note, that `artifactId` will

be used as your plugin (internal) name.

After the project is generated, you may want to update `teamcity-plugin.xml` in the root directory: enter display name, description, author e-mail and other information.

Finally, change the directory to the root of the generated project and run

```
mvn package
```

The `target` directory of the project root will contain the `<artifactId>.zip` file. It is your plugin package. You can [install it to TeamCity](#) or use the TeamCity Maven plugin.

#### TeamCity SDK Maven plugin

You can also use the [TeamCity SDK Maven plugin](#) allowing you to control a TeamCity instance from the command line and to install a new/updated plugin created from a Maven archetype

## Plugin Development FAQ

 TeamCity is hiring! Learn about [the available vacancies](#) on the JetBrains site. [Read](#) about working in the TeamCity team.

- [How to Use Logging](#)

#### How to Use Logging

The TeamCity code uses the Log4j logging library with a centralized configuration on the `server` and `agent`.

Logging is usually done via a utility wrapper `com.intellij.openapi.diagnostic.Logger` rather than the default Log4j classes. You can use the `jetbrains.buildServer.log.Loggers` class to get instances of the `Loggers`, e.g. use `jetbrains.buildServer.log.Loggers.SERVER` to add a message to the `teamcity-server.log` file.

For plugin-specific logging it is recommended to log into a log category matching the full name of your class. This is usually achieved by defining the logger field in a class as `private static Logger LOG =`

```
Logger.getInstance(YourClass.class.getName());
```

If your plugin source code is located under the `jetbrains.buildServer` package, the logging will automatically go into `teamcity-server.log`.

If you use another package, you might need to add a corresponding category handling into the `conf/teamcity-server-log4j.xml` file (mentioned at [TeamCity Server Logs](#)) or the corresponding agent file.

For debugging you might consider creating a customized Log4j configuration file and put it as a logging preset into `<TeamCity Data Directory>\config\_logging` directory. This way one will be able to activate the preset via the Administration | Diagnostics page, Troubleshooting tab.

## How To...

In this section:

- [Choose OS/Platform for TeamCity Server](#)
- [Estimate Hardware Requirements for TeamCity](#)
  - [Network Traffic between the Server and the Agents](#)
- [Configuring TeamCity Server for Performance](#)
- [Retrieve Administrator Password](#)
- [Estimate External Database Capacity](#)
- [Estimate the Number of Required Build Agents](#)
- [Setup TeamCity in Replication/Clustering Environment](#)
- [TeamCity Security Notes](#)
- [What Encryption is Used by TeamCity](#)
- [Configure Newly Installed MySQL Server](#)
  - [InnoDB database engine](#)
  - [max\\_connections](#)
  - [innodb\\_buffer\\_pool\\_size and innodb\\_log\\_file\\_size](#)
  - [innodb\\_file\\_per\\_table](#)
  - [innodb\\_flush\\_log\\_at\\_trx\\_commit](#)
  - [log files on different disk](#)

- Setting The Binary Log Format
- Enable additional diagnostics
- Configure Newly Installed PostgreSQL Server
  - shared\_buffers
  - checkpoint-related parameters
  - synchronous\_commit
- Set Up TeamCity behind a Proxy Server
  - Proxy Server Setup
    - Apache
    - NGINX
    - Other servers
  - TeamCity Tomcat Configuration
    - Dedicated "Connector" Node Approach
    - "RemoteIpValve" Approach
- Configure TeamCity to Use Proxy Server for Outgoing Connections
- Configure TeamCity Agent to Use Proxy To Connect to TeamCity Server
- Install Multiple Agents on the Same Machine
- Change Server Port
- Test-drive Newer TeamCity Version before Upgrade
- Create a Copy of TeamCity Server with All Data
  - Create a Server Copy
    - Use TeamCity Backup
    - Copy Manually
  - Environment transferring
  - Licensing issues
  - Copied Server Checklist
- Move TeamCity Projects from One Server to Another
- Move TeamCity Installation to a New Machine
- Move TeamCity Agent
- Share the Build number for Builds in a Chain Build
- Make Temporary Build Files Erased between the Builds
- Clear Build Queue if It Has Too Many Builds due to a Configuration Error
- Automatically create or change TeamCity build configuration settings
- Attach Cucumber Reporter to Ant Build
- Get Last Successful Build Number
- Set up Deployment for My Application in TeamCity
- Use an External Tool that My Build Relies on
- Integrate with Build and Reporting Tools
- Restore Just Deleted Project
- Transfer 3 Default Agents to Another Server
- Import coverage results in TeamCity
- Recover from "Data format of the data directory (NNN) and the database (MMM) do not match" error
- Debug a Build on a Specific Agent
- Debug a Part of the Build (a build step)
- Vulnerabilities
  - Heartbleed, ShellShock
  - POODLE
  - GHOST
  - FREAK
  - Apache Struts
  - Tomcat Under Windows
  - Tomcat CVE-2018-8037
- Watch Several TeamCity Servers with Windows Tray Notifier
- Personal User Data Processing
  - TeamCity and Users' Personal Data
  - Deleting the User Data
  - User Agreement
  - Encryption
  - Logs and Debugging Data
  - Customizations
- TeamCity Release Cycle

## Choose OS/Platform for TeamCity Server

Once the server/OS fulfills the [requirements](#), TeamCity can run on any system.

Please also review the [requirements](#) for the integrations you plan to use, for example the following functionality requires or works better when TeamCity server is installed under Windows:

- VCS integration with TFS
- VCS integration with VSS
- Windows domain logins (can also work under Linux, but may be less stable), especially NTLM HTTP authentication

- NuGet feed on the server (can also work under Linux, but may be less stable)
- Agent push to Windows machines

If you have no preference, Linux platforms may be more preferable due to more effective file system operations and the level of required general OS maintenance.

Final Operating System choice should probably depend more on the available resources and established practices in your organization.

If you choose to install 64 bit OS, TeamCity can run under 64 bit JDK (both server and agent).

However, unless you need to provide more than 1Gb memory for TeamCity, the recommended approach is to use 32 bit JVM even under 64 bit OS. Our experience suggests that using 64 bit JVM does not increase performance a great deal. At the same time it does increase memory requirements to almost the scale of 2. See a [note](#) on memory configuration.

## Estimate Hardware Requirements for TeamCity

The hardware requirements differ for the server and the agents.

The agent hardware requirements are basically determined by the builds that are run. Running TeamCity agent software introduces a requirement for additional CPU time (but it can usually be neglected comparing to the build process CPU requirements) and additional memory: about 500Mb. The disk space required corresponds to the disk usage by the builds running on the agent (sources checkouts, downloaded artifacts, the disk space consumed during the build; all that combined for the regularly occurring builds).

Although you can run a build agent on the same machine as the TeamCity server, the recommended approach is to use a separate machine (it may be virtual) for each build agent. If you chose to install several agents [on the same machine](#), please consider the possible CPU, disk, memory or network bottlenecks that might occur. The [Performance Monitor](#) build feature can help you in analyzing live data.

The server hardware requirements depend on the server load, which in its turn depends significantly on the type of the builds and server usage. Consider the following general guidelines.



- If you decide to run an [external database](#) on the same machine as the server, take into account hardware requirements with database engine requirements in mind.
- If you face some TeamCity-related performance issues, they should probably be investigated and addressed individually. e.g. if builds generate too much data, the server disk system might be needing an upgrade both in size and speed characteristics.

### Database Note:

When using the server extensively, the database performance starts to play a greater role.

For reliability and performance reasons you should use external database.

Please see the [notes](#) on choosing external database.

The database size requirements naturally vary based on the amount of data stored (number of builds, number of tests, etc.)

The active server database usage can be estimated at several gigabytes of data per year.

### Overview of the TeamCity hardware resources usage:

- CPU: TeamCity utilizes multiple cores of the CPU, so increasing number of cores makes sense. For non-trivial TeamCity usage at least 4 CPU cores are recommended.
- Memory: See a [note](#) on memory usage. Consider also that required memory may depend on the JVM used (32 bit or 64 bit). Generally, you will probably not need to dedicate more than 4G of memory to TeamCity server if you do not plan to run more than 100 concurrent builds (agents) and more than 200 online users.
- HDD/disk usage: This sums up mainly from the temp directory usage (<[TeamCity Home](#)>/temp and OS temp directory) and .BuildServer/system usage. Performance of the TeamCity server highly depends on the disk system performance. As TeamCity stores large amounts of data under .BuildServer/system (most notably, VCS caches and build results) it is important that the access to the disk is fast (in particular reading/writing files in multiple threads, listing files with attributes). Ensuring disk has good performance is especially important if you plan to store the data directory on a network drive.
- Network: This mainly sums up from the traffic from VCS servers, to clients (web browsers, IDE, etc.) and to/from build agents (send sources, receive build results, logs and artifacts).

### The load on the server depends on:

- number of build configurations;
- number of builds in the history;
- number of the builds running daily;
- amount of data consumed and produced by the builds (size of the used sources and artifacts, size of the build log, number and output size of unit tests, number of inspections and duplicates hits, size and number of produced artifacts, etc.);
- cleanup rules configured
- number of agents and their utilization percentage;
- number of users having TeamCity web pages open;

- number of users logged in from IDE plugin;
- number and type of VCS roots as well as checking for changes interval for the VCS roots. VCS checkout mode is relevant too: server checkout mode generates greater server load. Specific types of VCS also affect server load, but they can be roughly estimated based on native VCS client performance;
- number of changes detected by TeamCity per day in all the VCS roots;
- total size of the sources checked out by TeamCity daily.

A general example of hardware configuration capable to handle up to 100 concurrently running builds and running only TeamCity server can be:

Server-suitable modern multi-core CPU, 8Gb of memory, fast network connection, fast and reliable HDD, fast external database access

Based on our experience, a modest hardware like

Intel 3.2 GHz dual core CPU, 3.2Gb memory under Windows, 1Gb network adapter, single HDD can provide acceptable performance for the following setup:

- 60 projects and 300 build configurations (with one forth being active and running regularly);
- more than 300 builds a day;
- about 2Mb log per build;
- 50 build agents;
- 50 web users and 30 IDE users;
- 100 VCS roots (mainly Perforce and Subversion using server checkout), average checking for changes interval is 120 seconds;
- more than 150 changes per day;
- the database (MySQL) is running on the same machine;
- TeamCity server process has `-Xmx1100m` JVM setting.

The following configuration can provide acceptable performance for a more loaded TeamCity server:

Intel Xeon E5520 2.2 GHz CPU (4 cores, 8 threads), 12Gb memory under Windows Server 2008 R2 x64, 1Gb network adapter, 3 HDD RAID1 disks (general, one for artifacts, logs and caches storage, and one for the database storage)

Server load characteristics:

- 150 projects and 1500 build configurations (with one third being active and running regularly);
- more than 1500 builds a day;
- about 4Mb log per build;
- 100 build agents;
- 150 web users and 40 IDE users;
- 250 VCS roots (mainly Git, Hg, Perforce and Subversion using agent-side checkout), average checking for changes interval is 180 seconds;
- more than 1000 changes per day;
- the database (MySQL) is running on the same machine;
- TeamCity server process has `-Xmx3700m` x64 JVM setting.

However, to ensure peak load can be handled well, more powerful hardware is recommended.

HDD free space requirements are mainly determined by the number of builds stored on the server and the artifacts size/build log size in each. Server disk storage is also used to store VCS-related caches and you can estimate that at double the checkout size of all the VCS roots configured on the server.

If the builds generate large number of data (artifacts/build log/test data), using fast hard disk for storing .BuildServer/system directory and fast network between agents and server are recommended.

The general recommendation for deploying large-scale TeamCity installation is to start with a reasonable hardware while considering hardware upgrade.

Then increase the load on the server (e.g. add more projects) gradually, monitoring the performance characteristics and deciding on necessary hardware or software improvements. There is also a [benchmark plugin](#) which can be used to estimate the number of simultaneous build the current server installation can handle. Anyway, best administration practices are recommended like keeping adequate disk defragmentation level, etc.

Starting with an adequately loaded system, if you then increase the number of concurrently running builds (agents) by some factor, be prepared to increase CPU, database and HDD access speeds, amount of memory by the same factor to achieve the same performance.

If you increase the number of builds per day, be prepared to increase the disk size.

If you consider cloud deployment for TeamCity agents (e.g. on Amazon EC2), please also review [Setting Up TeamCity for Amazon EC2#Estimating EC2 Costs](#)

A note on agents setup in JetBrains internal TeamCity installation:

We use both separate machines each running a single agent and dedicated "servers" running several virtual machines each of them having a single agent installed. Experimenting with the hardware and software we settled on a configuration when each core i7 physical machine runs 3 virtual agents, each using a separate hard disk. This stems from the fact that our (mostly Java) builds depend on HDD performance in the first place. But YMMV.

TeamCity is known to work well with 500+ build agents (500 concurrently running builds actively logging build run-time data). In synthetic tests the server was functioning OK with as many as 1000 concurrent builds (the server with 8 cores, 32Gb of total

memory running under Linux, and MySQL server running on a separate comparable machine). The load on the server produced by each build depends on the amount of data the build produces (build log, tests number and failure details, inspections/duplicates issues number, etc.). Keeping the amount of data reasonably constrained (publishing large outputs as build artifacts, not printing those into standard output; tweaking inspection profiles to report limited set of the most important inspection hits, etc.) will help scale the server to handle more concurrent builds.

If you need much more agents/parallel builds, it is recommended to use [several nodes setup](#). If a substantially large amount of agents is required, it is recommended to consider using several separate TeamCity instances and distributing the projects between them. We constantly work on TeamCity performance improvements and are willing to work closely with organizations running large TeamCity installations to study any performance issues and improve TeamCity to handle larger loads. See also a related post on the [maximum number of agents which TeamCity can handle](#)

See also a related post: [description of a substantial TeamCity setup](#).

## Network Traffic between the Server and the Agents

The traffic mostly depends on the settings as some of them include transferring binaries between the agent and the server. The most important flows of traffic between the agent and the server are:

- agent retrieves commands from the server: these are typically build start tasks which basically include a dump of the build configuration settings and the full set of build parameters. The latter can be large (e.g. megabytes) in case of a large build chain. The parameters can be reviewed on the build's [Parameters tab](#);
- agent periodically sends current status data to the server (this includes all the agents parameters which can be reviewed on the agent's [Agent Parameters tab](#));
- during the build, the agent sends build log messages and parameters data back to the server. These can be reviewed on the [Build Log](#) and [Parameters](#) tabs of the build;
- (when the server-side checkout mode is used) the agent downloads the sources before the build (as a full or incremental patch) from the server;
- (when an [artifact dependency](#) is configured) the agent downloads build artifacts of other builds from the server before starting a build;
- (when artifacts are configured for a build) the agent uploads build artifacts to the server;
- some runners (like coverage or code analysis) include automatic uploading of their results' reports to the server.

## Configuring TeamCity Server for Performance

Here are some recommendations to tweak TeamCity server setup for better performance. The list for [production server use](#) is a prerequisite:

- Regularly review reported Server Health reports (including hidden ones)
- Use a separate [reverse proxy](#) server (e.g. NGINX) to handle HTTPS
- Use a separate server for the external database and monitor the database performance
- Monitor the server's CPU and IO performance, increase hardware resources as necessary (see also [hardware notes](#))
- Make sure clean-up is configured for all the projects with a due retention policy, make sure clean-up completely finishes regularly (check Administration / Clean-Up page)
- Consider ensuring good IO performance for the <TeamCity Data Directory>/system/caches directory, e.g. by moving it to a separate local drive (or storing on a local drive you choose to store the TeamCity Data Directory on a network storage)
- Regularly archive obsolete projects
- Regularly review the installed not bundled plugins and remove those not essential for the server functioning
- Consider using agent-side checkout whenever possible
- Make sure the build logs are not huge (tens megabytes at most, better less than 10 Mb)
- If lots VCS roots are configured on the server, consider configuring [repository commit hooks](#) instead of using polling for changes or at least increase [VCS polling interval](#) to 300 seconds or more
- If the server is often used by large number of users (e.g. more than 1000), consider reducing the frequency of background UI requests their by increasing [UI refresh intervals](#)
- When regularly exceeding 500 concurrently running builds which log a lot of data, consider using [Several Nodes Setup](#)

## Retrieve Administrator Password

On the first start with the empty database, TeamCity displays the Administrator Setup page which allows creating a user with full administrative permissions (assigning the [System Administrator](#) role).

If you want to regain access to the system and you cannot log in as a user with the System Administrator role, you can log in as a [super user](#) and change the existing administrator account password or create a new account with the System Administrator role.

It is also possible to use [REST API](#) to add the System Administrator role to any existing user.

If you use built-in authentication and have correct email specified, you can [reset the password](#) from the login page.

## Estimate External Database Capacity

It is quite hard to provide the exact numbers when setting up or migrating to an external database, as the required capacity varies greatly depending on how TeamCity is used.

The database size and database performance are crucial aspects to consider.

### Database Size

The size of the database will depend on:

- how many builds are started every day
- how many test are reported from builds
- [clean-up](#) rules (retention policy)
- cleanup schedule

We recommend the initial size of data spaces to be 4 GB. When migrating from the internal database, we suggest at least doubling the size of the current internal database. For example, the size of the external database (without the Redo Log files) of the internal TeamCity server in JetBrains is about 50 GB. Setting your database to grow automatically helps to increase file sizes to a pre-determined limit when necessary, which minimizes the effort to monitor disk space.

Allocating 1 GB for the redo log (see the table [below](#)) and undo files is sufficient in most cases.

### Database Performance

The following factors are to be taken into account:

- type of database (RDBMS)
- number of agents (which actually means the number of builds running in parallel)
- number of web pages opened by all users
- [clean-up](#) rules (retention policy)

It is advised to place the [TeamCity Data directory](#) and database data files on physically different hard disks (even when both the TeamCity server and RDBMS share the same host).

Placing redo logs on a separate physical disk is also recommended especially in case of the high number of agents (50 and more).

### Database-specific considerations

 The redo log (or a similar entity) naming for different RDBMS:

RDBMS	Log name
Oracle	Redo Log
MS SQL Server	Transaction Log
PostgreSQL	WAL (write ahead log)
MySQL + InnoDB and Percona	Redo Log

PostgreSQL: We recommend using version 9.2+, which has a lot of query optimization features. Also see the information on the write-ahead-log (WAL) in the [PostgreSQL documentation](#)

Oracle: it is recommended to keep statistics on: all automatically gathered statistics should be enabled (since Oracle 10.0, this is the default set-up). Also see the information on redo log files in the [Oracle documentation](#).

MS SQL Server: it is NOT recommended to use the jTDS driver: it does not work with nchar/nvarchar, and to preserve unicode streams it may cause queries to take a long time and consume a lot of IO. Also see the information on redo log in the [Microsoft Knowledge base](#). If you use jTDS, please migrate.

MySQL: the query optimizer might be inefficient: some queries may get a wrong execution plan causing them to take a long time and consume huge IO.

## Estimate the Number of Required Build Agents

There are no precise data and the number of required build agents depends a lot on the server usage pattern, type of builds, team size, commitment of the team to CI process, etc.

The best way is to start with the default 3 agents and see how that plays with the projects configured, then estimate further

based on that.

You might want to increase the number of agents when you see:

- builds waiting for an idle agent in the build queue;
- more changes included into each build than you find comfortable (e.g. for build failures analysis);
- necessity for different environments.

We've seen patterns of having an agent per each 20 build configurations (types of builds). Or a build agent per 1-2 developers.

See also [notes](#) on maximum supported number of agents.

## Setup TeamCity in Replication/Clustering Environment

TeamCity only supports a single instance of the main server, but it is possible to add [read-only node](#) (to provide view-only UI over the current data) and [running builds node](#) to collect the running builds data from the agents. All three nodes need to be connected to the same [TeamCity Data Directory](#) and the database.

To address fast disaster recovery scenarios, TeamCity supports active - failover (cold standby) approach: the data that the TeamCity server uses can be replicated and a solution put in place to start a new server using the same data if the currently active server malfunctions.

As to the data, the TeamCity server uses both database and file storage (Data Directory). You can browse through [TeamCity Data Backup](#) and [TeamCity Data Directory](#) pages in to get more information on TeamCity data storing.

Basically, both TeamCity data directory on the disk and the database which TeamCity uses must remain in a consistent state and thus must be replicated together.

Only a single TeamCity server instance should use the database and data directory at any time.

Ensure that the distribution of the TeamCity failover/backup server is of exactly the same version as the main server. It is also important to ensure the same server environment/startup options like memory settings, etc.

TeamCity agents farm can be reused between the main and the failover servers. Agents will automatically connect to the new server if you make the failover server to be resolved via the old server DNS name and agents connect to the server using the DNS name. See also information on [switching](#) from one server to another.

If appropriate, the agents can be replicated just as the server. However, there is no need to replicate any TeamCity-specific data on the agents except for the `conf/buildAgent.properties` file as all the rest of the data can typically be renewed from the server. In case of replicated agents farm, the replica agents just need to be connected to the failover server.

In case of two servers installations for redundancy purposes, they can use the same set of licenses as only one of them is running at any given moment.

## TeamCity Security Notes

The following notes are provided for your reference only and are not meant to be complete or accurate in their entirety.

TeamCity is developed with security concerns in mind, and reasonable efforts are made to make the system invulnerable to different types of attacks. We work with third-parties on assessing TeamCity security using security scanners and penetration tests.

We aim to promptly address newly discovered security issues in the nearest bug-fix releases for the most recent TeamCity major version.

However, the general assumption and recommended setup is to deploy TeamCity in a trusted environment with no possibility for it to be accessed by malicious users.

Along with these guidelines, please review [notes](#) on configuring the TeamCity server for production use. For the list of disclosed security-related issues, see our [public issue tracker](#) and the "Security" section in the release notes. It is recommended to upgrade to newly released TeamCity versions as soon as they become available as they can contain security-related fixes.

Note that since TeamCity 2017.2 the TeamCity Windows installer modifies permissions of the [TeamCity installation directory](#) not to use inheritable permissions and explicitly grants access to the directory to the Administrators user group and the account under which the service is configured to run.

It is strongly recommended to restrict permissions to the [TeamCity Data Directory](#) in the same way.

### Additional security-related settings

Consider adding the `"teamcity.installation.completed=true"` line into the `<TeamCity Home Directory>\conf\teamcity-startup.properties` file - this will prevent the server started with the empty database from granting access to the machine for the first coming user.

TeamCity has no built-in protection against DoS (Denial-of-service) attack: high rate of requests can overload the server and make it not responsive. If your TeamCity instance is deployed in the environment which allows such service abuse, implement the protection on the reverse proxy level.

Short checklist (see below for full notes)

- You are running the latest released TeamCity version and are ready to upgrade to the newly released versions within weeks
- Access to the TeamCity web interface is secured using HTTPS (e.g. with the help of a proxying server like NGINX). Best practices for securing web applications are employed for the TeamCity web interface. e.g.: It is not possible to access the server using HTTP protocol. Reverse proxy does not strip Referer request header
- The TeamCity server machine does not run agents (at least under the user permitted to read the TeamCity server's [home directory](#) and [TeamCity Data Directory](#))
- TeamCity server and agents processes are run under limited users with minimal required permissions. Installation directories are readable and writable only by a limited set of OS users. The `conf\buildAgent.properties` file and server logs as well as the Data Directory are only readable by OS users who represent administrators of the services, because reading those locations may allow taking over the agent or server respectively.
- Guest user and user registration is disabled or roles are reviewed for guest and the [All Users](#) group
- TeamCity users with administrative permissions have non-trivial passwords
- If you have external authentication configured (such as LDAP), the [built-in authentication](#) module is disabled
- Passwords are not printed into the build log, not stored in build artifacts, nor are they stored in non-[password parameters](#)

## Security-related Risks Evaluation

Here are some notes on different security-related aspects:

- man-in-the-middle concerns
  - between the TeamCity server and the user's web browser: It is advised to [use HTTPS](#) for the TeamCity server. During login, TeamCity transmits the user login password in an encrypted form with a moderate encryption level.
  - between a TeamCity agent and the TeamCity server: see the section.
  - between the TeamCity server and other external servers (version control, issue tracker, etc.): the general rules apply as for a client (the TeamCity server in the case) connecting to the external server, see the guidelines for the server in question.
- users that have access to the TeamCity web UI: the specific information accessible to the user is defined via TeamCity [user roles](#).
- users who can change the code that is used in the builds run by TeamCity (including committers in any branches/pull requests if they are built on TeamCity):
  - can do everything what can do the system user under whom the TeamCity agent is running, have access to OS resources and other applications installed on the agent machines where their builds can run.
  - can access and change source code of other projects built on the same agent, modify the TeamCity agent code, publish any files as artifacts for the builds run on the agent (which means the files can be then displayed in the TeamCity web UI and expose web vulnerabilities or can be used in other builds), etc.
  - can impersonate a TeamCity agent (run a new agent looking the same to the TeamCity server).
  - can do everything that users with the "View build configuration settings" permission for all the projects on the server can do (see below).
  - can retrieve settings of the build configurations where the builds are run, including the values of the password fields.
  - can download artifacts from any build on the server.

Hence, it is advised to run TeamCity agents under an OS account with only [necessary set of permissions](#) and use the [agent pools](#) feature to ensure that projects requiring a different set of access are not built on the same agents.
- users with the "View build configuration settings" permission (the "Project developer" TeamCity role by default) can view all the projects on the server, but since TeamCity 9.0 there is a way to restrict this, see details in the corresponding issue [TW-24904](#).
- users with the "Edit project" permission (the "Project Administrator" TeamCity role by default) in one project, by changing settings can retrieve artifacts and trigger builds from any build configuration they have only the view permission for ([TW-39209](#)). The users might also be able to make the TeamCity server run any executable located on the server.
- users with the "Change server settings" permission (the "System Administrator" TeamCity role by default): It is assumed that the users also have access to the computer on which the TeamCity server is running under the user account used to run the server process. Thus, the users can get full access to the machine under that OS user account: browse file system, change files, run arbitrary commands, etc.
- TeamCity server computer administrators: have full access to TeamCity stored data and can affect TeamCity executed processes. Passwords that are necessary to authenticate in external systems (like VCS, issue trackers, etc.) are stored in a scrambled form in [TeamCity Data Directory](#) and can also be stored in the database. However, the values are only scrambled, which means they can be retrieved by the users who have access to the server file system or database.
- Users who have read access for TeamCity server logs (TeamCity server home directory) can escalate their access to TeamCity server administrator
- Users who have read access to [TeamCity Data Directory](#) can access all the settings on the server, including configured passwords
- Users who have read access to the build artifacts in [TeamCity Data Directory](#) (<TeamCity Data Directory>\system\artifacts) get the same permissions as users with the "View build runtime parameters and data" permission (in particular, with access to the values of all the password parameters used in the build)
- TeamCity agent computer administrators: same as "users who can change code that is used in the builds run by TeamCity".
- It is recommended to distribute projects among agents, so that one TeamCity agent would not run builds of several projects whose developers and administrators should not get access to each other's projects. The recommended way to

distribute projects is to use the [agent pools](#) feature and make sure that the "Default" agent pool has no agents as a project can be assigned to the Default pool after a certain reconfiguration (i.e. when there is no other pool the project is assigned to).

- When [storing settings in VCS](#) is enabled:
  - any user who can access the settings repository (including users with "View file content" permission for the build configurations using the same VCS root) can see the settings and retrieve the actual passwords based on their stored scrambled form
  - any user who can modify settings in VCS for a single build configuration built on the server, via changing settings can retrieve artifacts and trigger builds from any build configuration they have only view permission for ([TW-39192](#)).
  - users who can customize build configuration settings on a per-build basis (e.g. one who can run personal builds when versioned settings are set to "use settings from VCS") via changing settings in a build can retrieve artifacts and trigger builds from any build configuration they have only view permission for ([TW-46065](#)).
- Other:
  - TeamCity web application vulnerabilities: the TeamCity development team makes a reasonable effort to fix any significant vulnerabilities (like cross-site scripting possibilities) once they are uncovered. Please note that any user that can affect build files ("users who can change code that is used in the builds run by TeamCity" or "TeamCity agent computer administrators") can make a malicious file available as a build artifact that will then exploit cross-site scripting vulnerability. ([TW-27206](#))
  - TeamCity agent is fully controlled by the TeamCity server: since TeamCity agents support automatic updates download from the server, agents should only connect to a trusted server. An administrator of the server computer can force execution of arbitrary code on a connected agent.
  - Binaries of the agent plugins installed on the server are available to anyone who can access the server URL

## What Encryption is Used by TeamCity

TeamCity tries not to pass password values in the web UI (from a browser to the server) in clear text and uses RSA with 1024-bit key to encrypt them. However, it is recommended to use the TeamCity web UI only via HTTPS so this precaution should not be relevant.

TeamCity stores passwords in the settings (where the original password value is necessary to perform authentication in other systems) in a scrambled form. The scrambling is done using 3DES with a fixed key.

## Configure Newly Installed MySQL Server

If MySQL server is going to be used with TeamCity in addition to the [basic setup](#), you should review and probably change some of the MySQL server settings.

If MySQL is installed on Windows, the settings are located in `my.ini` file which usually can be found under MySQL installation directory. For Unix-like systems the file is called `my.cnf` and can be placed somewhere under `/etc` directory. Read more about configuration file location in [MySQL documentation](#). Note: you'll need to restart MySQL server after changing settings in `my.ini` | `my.cnf`.

The following settings should be reviewed and/or changed:

### InnoDB database engine

Make sure you're using InnoDB database engine for tables in TeamCity database. You can check what engine is used with help of this command:

```
show table status like '<table name>';
```

or for all tables at once:

```
show table status like '%';
```

### max\_connections

You should ensure `max_connections` parameter has bigger value than the one specified in TeamCity <TeamCity data directory>/config/database.properties file.

### innodb\_buffer\_pool\_size and innodb\_log\_file\_size

Specifying a too small value in `innodb_buffer_pool_size` may significantly affect performance:

```
# InnoDB, unlike MyISAM, uses a buffer pool to cache both indexes and
# row data. The bigger you set this the less disk I/O is needed to
# access data in tables. On a dedicated database server you may set this
# parameter up to 80% of the machine physical memory size. Do not set it
# too large, though, because competition of the physical memory may
# cause paging in the operating system. Note that on 32bit systems you
# might be limited to 2-3.5G of user level memory per process, so do not
# set it too high.
innodb_buffer_pool_size=2000M
```

We recommend to start with 2Gb and increase it if you experience slowness and have enough memory. After increasing buffer pool size you should also change size of the `innodb_log_file_size` setting (its value can be calculated as `innodb_buffer_pool_size/N`, where N is the number of log files in the group (2 by default)):

```
innodb_log_file_size=1024M
```

#### innodb\_file\_per\_table

For better performance you can enable the so-called [per-table tablespaces](#).

Note that once you add `innodb_file_per_table` option new tables will be created and placed in separate files, but tables created before enabling this option will still be in the shared tablespace.  
You'll need to re-import database for them to be placed in separate files.

#### innodb\_flush\_log\_at\_trx\_commit

If TeamCity is the only application using MySQL database then you can improve performance by setting `innodb_flush_log_at_trx_commit` variable to 2 or 0:

```
# If set to 1, InnoDB will flush (fsync) the transaction logs to the
# disk at each commit, which offers full ACID behavior. If you are
# willing to compromise this safety, and you are running small
# transactions, you may set this to 0 or 2 to reduce disk I/O to the
# logs. Value 0 means that the log is only written to the log file and
# the log file flushed to disk approximately once per second. Value 2
# means the log is written to the log file at each commit, but the log
# file is only flushed to disk approximately once per second.
innodb_flush_log_at_trx_commit=2
```

Note: it is not important for TeamCity that database offers full ACID behavior, so you can safely change this variable.

#### log files on different disk

Placing the MySQL log files on different disk sometimes helps improving performance. You can read about it in [MySQL documentation](#).

#### Setting The Binary Log Format

If the default MySQL binary logging format is not MIXED (it depends on the [version](#) of MySQL you are using), then it should be explicitly set to MIXED:

```
binlog-format=mixed
```

#### Enable additional diagnostics

To get additional diagnostics data in case of some database-specific errors, grant more permissions for a TeamCity database user via SQL command:

```
GRANT PROCESS ON *.* TO <teamcity-user-name>;
```

## Configure Newly Installed PostgreSQL Server

For better TeamCity server performance, it is recommended to change some of the parameters of the newly installed PostgreSQL server. You can read more about PostgreSQL performance optimization in [PostgreSQL Wiki](#).

The parameters below can be changed in the `postgresql.conf` file located in the in PostgreSQL's data directory.

### shared\_buffers

The default value of `shared_buffers` parameter is too small and should be increased:

```
shared_buffers=512MB
```

### checkpoint-related parameters

For write-intensive applications such as TeamCity, it is recommended to change some of the `checkpoint-related` parameters:

For PostgreSQL 9.5 and later:

```
max_wal_size = 1500MB
checkpoint_completion_target=0.9
```

For versions prior to PostgreSQL 9.5:

```
checkpoint_segments=32
checkpoint_completion_target=0.9
```

### synchronous\_commit

If TeamCity is the only application using the PostgreSQL database, we recommend disabling the `synchronous_commit` parameter:

```
synchronous_commit=off
```

## Set Up TeamCity behind a Proxy Server

This section covers the recommended setup of reverse-proxy servers installed in front of the TeamCity server web UI. Configuring HTTPS on the proxy level is recommended, but is out of the scope of these instructions - refer to the documentation of the proxy server for that.

Consider the example:

TeamCity server is installed at URL (local URL): `http://teamcity.local:8111/tc`

It is visible to the outside world as URL (public URL): `http://teamcity.public:400/tc`

You need to set up a reverse proxy (see [Proxy Server Setup](#) below) and also configure TeamCity's bundled Tomcat server (see [TeamCity Tomcat Configuration](#) below) to make sure TeamCity "knows" the actual absolute URL used by the client to access the resources. These URLs are then used to generate absolute URLs in client redirects and other responses.

Note: An internal TeamCity server should work under the same context (i.e. part of the URL after the host name) as it is visible from outside by an external address. See also TeamCity server [context changing instructions](#). If you still need to run the server under a different context, note that context changing proxy should conceal this fact from the TeamCity: e.g. it should map server redirect URLs as well as cookies setting paths to the original (external) context.

## Proxy Server Setup

The proxy should be configured with generic web security in mind. e.g. headers like Referer and Origin should usually be passed to the TeamCity web application in the unmodified form.

Also, unknown HTTP request headers should be passed to the TeamCity web application unmodified. For example TeamCity relies on "X-TC-CSRF-Token" header added by the clients.

### Apache

Versions 2.4.5+ are recommended. Earlier versions do not support the WebSocket protocol, so use the settings noted in [the previous documentation version](#).

When using Apache, make sure to use the Dedicated "Connector" node approach for configuring TeamCity server.

```
LoadModule proxy_module      /usr/lib/apache2/modules/mod_proxy.so
LoadModule proxy_http_module /usr/lib/apache2/modules/mod_proxy_http.so
LoadModule headers_module   /usr/lib/apache2/modules/mod_headers.so
LoadModule proxy_wstunnel_module
/usr/lib/apache2/modules/mod_proxy_wstunnel.so

ProxyRequests Off
ProxyPreserveHost On
ProxyPass      /tc/app/subscriptions
ws://teamcity.local:8111/tc/app/subscriptions connectiontimeout=240
timeout=1200
ProxyPassReverse /tc/app/subscriptions
ws://teamcity.local:8111/tc/app/subscriptions

ProxyPass      /tc http://teamcity.local:8111/tc connectiontimeout=240
timeout=1200
ProxyPassReverse /tc http://teamcity.local:8111/tc
```

Please note the order of [ProxyPass](#) rules: conflicting ProxyPass rules must be sorted starting with the longest URLs first.

**i** Note that by default Apache allows only a limited number of parallel connections that may be insufficient when using the WebSocket protocol. For instance, it may result in the TeamCity server not responding when a lot of clients open the Web UI. To fix it, you may need to fine-tune the Apache configuration.

For example, on Unix you should switch to [mpm\\_worker](#) and configure the maximum number of simultaneous connections:

```

<IfModule mpm_worker_module>
    ServerLimit 100
    StartServers 3
    MinSpareThreads 25
    MaxSpareThreads 75
    ThreadLimit 64
    ThreadsPerChild 25
    MaxClients 2500
    MaxRequestsPerChild 0
</IfModule>

```

On Windows you may need to increase the `ThreadsPerChild` value as described [in the Apache documentation](#).

## NGINX

For the NGINX configuration below, use the "RemoteIpValve" Approach for configuring TeamCity server.

Versions 1.3+ are recommended. Earlier versions do not support the WebSocket protocol, so use the settings noted in [the previous documentation version](#).

```

http {
    # ... default settings here
    proxy_read_timeout 1200;
    proxy_connect_timeout 240;
    client_max_body_size 0; # maximum size of an HTTP request. 0 allows uploading large artifacts
    to TeamCity

    map $http_upgrade $connection_upgrade { # WebSocket support
        default upgrade;
        '' '';
    }

    server {
        listen 400; # public server port
        server_name teamcity.public; # public server host name

        location /tc { # public context (should be the same as internal context)
            proxy_pass http://teamcity.local:8111/tc; # full internal address
            proxy_http_version 1.1;
            proxy_set_header Host $server_name:$server_port;
            proxy_set_header X-Forwarded-Host $http_host; # necessary for proper absolute
            redirects and TeamCity CSRF check
            proxy_set_header X-Forwarded-Proto $scheme;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_set_header Upgrade $http_upgrade; # WebSocket support
            proxy_set_header Connection $connection_upgrade; # WebSocket support
        }
    }
}

```

## Other servers

Make sure to use a performant proxy with due (high) limits on request (upload) and response (download) size and timeouts (at least tens of minutes and gigabyte, according to the sizes of the codebase and artifacts).

It is recommended to use a proxy capable of working with the WebSocket protocol as that helps UI to refresh sooner.

Generally, you need to configure the TeamCity server so that it "knows" about the original URL used by the client and it can generate correct absolute URLs accessible for the client. Preferred way to achieve that is to pass original "Host" header to TeamCity. Alternative is to set "X-Forwarded-Host" header to the original "Host" header value.

Note that whenever the value of the "Host" header is changed by the proxy (while it is recommended to preserve original Host header value), the values of the Origin and Referer headers should be mapped correspondingly if they contain the original Host header value (if they do not, they should not be set in order not to circumvent TeamCity [CSRF protection](#)).

Select a proper approach from the section below and configure the proxy accordingly.

## TeamCity Tomcat Configuration

For a due TeamCity Tomcat configuration, there are two options:

- use a dedicated "Connector" node in the server configuration with hard-coded public URL details and make sure that the port configured in the connector is used only by the requests to the public URL configured.
- configure proxy to pass due request headers: "Host" or "X-Forwarded-Host" (original request host), "X-Forwarded-Proto" (original request protocol), "X-Forwarded-Port" (original request port) and configure "RemoteIpValve" for the TeamCity Tomcat configuration.

### Dedicated "Connector" Node Approach

This approach can be used with any proxy configuration, provided the configured port is receiving requests only to the configured public URL.

Set up the proxying server to redirect all requests to `teamcity.public:400` to a dedicated port on the TeamCity server (`8111` in the example below) and change "Connector" node in `<TeamCity Home>\conf\server.xml` file as below. Note that the "Connector" port configured this way should only be accessible via the configured proxy's address. If you also want to make TeamCity port accessible directly, use a separate "Connector" node with a dedicated `port` value for that.

```
<Connector port="8111" protocol="org.apache.coyote.http11.Http11NioProtocol"
           connectionTimeout="60000"
           useBodyEncodingForURI="true"
           socket.txBufSize="64000"
           socket.rxBufSize="64000"
           tcpNoDelay="1"
           proxyName="teamcity.public"
           proxyPort="400"
           secure="false"
           scheme="http"
           />
```

When the public server address is HTTPS, use the `secure="true"` and `scheme="https"` attributes.

### "RemoteIpValve" Approach

This approach can be used when the proxy server sets "X-Forwarded-Proto", "X-Forwarded-Port" request headers to the values of the original URL. Also, while not critical for the most setups, this approach can be used to make sure the original client IP is passed to the TeamCity server correctly. This is important for legacy agents' [bidirectional communication](#).

Add the following into the Tomcat main `<Host>` node of the `conf\server.xml` file (see also [Tomcat doc](#)):

```
<Valve
    className="org.apache.catalina.valves.RemoteIpValve"
    remoteIpHeader="x-forwarded-for"
    protocolHeader="x-forwarded-proto"
    portHeader="x-forwarded-port"
    />
```

It is also recommended to specify `internalProxies` attribute with the regular expression matching only IP address of the proxy server. e.g. `internalProxies="192\.168\.0\.1"`

## Configure TeamCity to Use Proxy Server for Outgoing Connections

This section describes configuring TeamCity to use proxy server for certain outgoing HTTP connections. To connect TeamCity behind a proxy to Amazon EC2 cloud agents, see [this section](#).

TeamCity can use proxy server for certain outgoing HTTP connections made by the TeamCity server to other services like issues trackers, etc.

To point TeamCity to your proxy server:

Since TeamCity 2017.1.5 the following server [internal properties](#) are available (see the section [below](#) for previous version):

```
# For HTTP protocol
## The domain name or the IP address of the proxy host and the port:
teamcity.http.proxyHost=proxy.domain.com
teamcity.http.proxyPort=8080

## The hosts that should be accessed without going through the proxy, usually internal hosts. Provide
a list of hosts, separated by the '|' character. The wildcard '*' can be used:
teamcity.http.nonProxyHosts=localhost|*.mydomain.com

## For an authenticated proxy add the following properties:
### Authentication type. "basic" and "ntlm" values are supported. The default is basic.
teamcity.http.proxyAuthenticationType=basic
### Login and Password for the proxy:
teamcity.http.proxyLogin=username
teamcity.http.proxyPassword=password

# For HTTPS protocol
## The domain name or the IP address of the proxy host and the port:
teamcity.https.proxyHost=proxy.domain.com
teamcity.https.proxyPort=8080

## The hosts that should be accessed without going through the proxy, usually internal hosts. Provide
a list of hosts, separated by the '|' character. The wildcard '*' can be used:
teamcity.https.nonProxyHosts=localhost|*.mydomain.com

## For an authenticated proxy add the following properties:
### Authentication type. "basic" and "ntlm" values are supported. The default is basic.
teamcity.https.proxyAuthenticationType=basic
### Login and Password for the proxy:
teamcity.https.proxyLogin=login
teamcity.https.proxyPassword=password
```

The alternative approach, which will work for any TeamCity version, is to pass additional space-delimited [additional JVM options](#) to the TeamCity server on the start up:

```
-Dproxyset=true
-Dhttp.proxyHost=proxy.domain.com
-Dhttp.proxyPort=8080
-Dhttp.nonProxyHosts=domain.com
-Dhttps.proxyHost=proxy.domain.com
-Dhttps.proxyPort=8080
-Dhttps.nonProxyHosts=domain.com
```

## Configure TeamCity Agent to Use Proxy To Connect to TeamCity Server

This section covers the configuration of a proxy server for TeamCity agent-to-server connections (since TeamCity 2017.1) On the TeamCity agent side, specify the proxy to connect to TeamCity server using the following properties in the `buildAgent.properties` file:

```
## The domain name or the IP address of the proxy host and the port  
teamcity.http.proxyHost=123.45.678.9  
teamcity.http.proxyPort=8080  
  
## If the proxy requires authentication, specify the login and password  
teamcity.http.proxyLogin=login  
teamcity.http.proxyPassword=password
```

Note that the proxy has to be configured not to cache any TeamCity server responses; e.g. if you use Squid, add "cache deny all" line to the `squid.conf` file.

## Install Multiple Agents on the Same Machine

See the [corresponding section](#) under agent installation documentation.

## Change Server Port

See [corresponding section](#) in server installation instructions.

## Test-drive Newer TeamCity Version before Upgrade

It's advised to try a new TeamCity version before upgrading your production server. The usual procedure is to [create a copy](#) of your production TeamCity installation, then [upgrade](#) it, try the things out, and, when everything is checked, drop the test server and upgrade the main one.

When you start the test server, remember to change the Server URL, disable Email and Jabber notifiers as well as [other features](#) on the new server.

## Create a Copy of TeamCity Server with All Data

In case you want to preserve the original server as well as the copy, make sure to check the [licensing considerations](#).

### Create a Server Copy

You can create a copy of the server using TeamCity backup functionality or manually. Manual approach is recommended for the complete server move.

### Use TeamCity Backup

1. Create a [backup](#).
2. Install a new TeamCity server of the same version that you are already running. Ensure that:
  - the appropriate environment is configured (see the notes below)
  - the server uses its own [TeamCity Data Directory](#) and its own [database](#) (check `config/database.properties` in the Data Directory)
3. [Restore the backup](#).
4. Perform the necessary environment transfer.



The new server won't get build artifacts and some other data. If you need them, you will need to copy appropriate directories (e.g. the entire "artifacts" directory) from `.BuildServer/system` from the original to the copied server. Make sure to finish copying over the artifacts before starting the new server.

### Copy Manually

If you do not want to use bundled backup functionality or need manual control over the process, here is a description of the general steps one would need to perform to manually create copy of the server:

1. Create a [backup](#) so that you can restore it if anything goes wrong,
2. Ensure the server is not running,
3. Either perform clean [installation](#) or copy the TeamCity binaries ([TeamCity Home Directory](#)) into a new place (the `temp` and `work` subdirectories can be omitted during copying). ⚠ Use exactly the same TeamCity version. If you plan to upgrade after copying, perform the upgrade only after you have the existing version up and running.
4. Copy [TeamCity Data Directory](#). If you do not need the full copy, refer to the items below for options.
  - `.BuildServer/config` to preserve projects and build configurations settings
  - `.BuildServer/lib` and `.BuildServer/plugins` if you have them
  - files from the root of `.BuildServer/system` if you use internal database and you do not want to perform database move.
  - `.BuildServer/system/artifacts` (optional) if you want build artifacts and build logs (including tests failure details) preserved on the new server
  - `.BuildServer/system/changes` (optional) if you want personal changes preserved on the new server
  - `.BuildServer/system/pluginData` (optional) if you want to preserve state of various plugins, build triggers and settings audit diff
  - `.BuildServer/system/caches` and `.BuildServer/system/caches` (optional) are not necessary to copy to the new server, they will be recreated on startup, but can take some time to be rebuilt (expect some slow down).
5. Artifacts directory is usually large and if you need to minimize the downtime of the server in case of the server move, you can use the generic approach for copying the data: use `rsync`, `robocopy` or alike tool to copy the data while the original server is running. Repeat the sync run several times until the amount of data synced reduces. Run the final sync after the original server shut down. Alternative solution for the server move is to make the old data artifacts directory accessible to the new server and configure it as second [location of artifacts](#). Then copy the files over from this second location to the main one while the server is running, restart the server after copying completion.
6. Create copy of the [database](#) that your TeamCity installation is using in new schema or new database server. This can be done with database-specific tools or with the bundled `maintainDB` tool by [backing up](#) database data and then [restoring](#) it.
7. Configure new TeamCity installation to use proper [TeamCity Data Directory](#) and [database](#) (`.BuildServer/config/database.properties` points to a copy of the database)
8. Perform the necessary [environment transfer](#).

i If you want to do a quick check and do not want to preserve builds history on the new server you can skip step 6 (cloning database) and all items of the step 5 marked as optional.

## Environment transferring

Consider transferring relevant environment if it was specially modified for an existing TeamCity installation. This might include:

- use the appropriate OS user account for running the TeamCity server process with properly configured settings, global and file system permissions
- use the same [TeamCity process launching options](#), specifically check/copy environment variables starting with `TEAMCITY`
- ensure any files/settings that were configured in the TeamCity web UI with absolute paths are accessible
- if relying on the OS-level user/machine settings like default ssh keys, cached VCS access credentials, transfer them as well
- consider replicating any special settings or exceptions related to the machine in the network configuration, etc.
- If the TeamCity installation was patched in any way (GroovyPlug plugin, native driver for MS SQL Server integrated security authentication), apply the same modifications to the installation copy
- if you run TeamCity with the OS startup (e.g. Windows service), make sure the same configuration is performed on the new machine
- review and transfer settings in the `<TeamCity home directory>\conf\teamcity-startup.properties` file
- consider any custom settings in `<TeamCity home directory>\conf\server.xml`

## Licensing issues

A single TeamCity license cannot be used on two running servers at the same time.

- A copy of the server created for redundancy/backup purposes can use the same license as only one of the servers will be running at a time.
- A copy of the server created for testing purposes requires an additional license. You can get the time-limited TeamCity [evaluation license](#) once from the official TeamCity [download page](#). If you need an extension of the license or you have already evaluated the same TeamCity version, please contact our [sales department](#).
- A copy of the server intended to run at the same time as the main one regularly/for production purposes requires a separate license.

## Copied Server Checklist

If you are creating a copy (as opposed to moving the server this way), it is important to go through the checklist below:

- ensure the new server is configured to use another data directory and another database than the original server; check also "Artifact directories" setting on server's Global Settings;
- change server unique id by removing "uuid" attribute from XML of <TeamCity Data Directory>\config\main-config.xml file before the first start;
- ensure the same license keys are not used on several servers ([more on licensing](#));
- update [Server URL](#) on Administration | Global Settings page to the actual URL of the server;
- check that you can successfully authenticate on the new server, use [super user](#) access if necessary;
- check that VCS servers, issue tracker servers, email and Jabber server and other server-accessed systems are accessible;
- check that any systems configured to push events to TeamCity server (like VCS hooks, automated build triggering, monitors, etc.) are updated to know about the new server;
- review the list of installed plugins to determine if their settings need changes;
- install new agents (or select some from the existing ones) and configure them to connect to the new server (using the new server URL);
- check that clients reading from the server (downloading artifact, using server's REST API, NuGet feed, etc.) are reconfigured, if necessary.

If you are creating a test server, you need to ensure that the users and production systems are not affected. Typically, this means you need to:

- disable Email, Jabber (in the "Administration > Notifier" sections) and possibly also custom notifiers or change their settings to prevent the new server from sending out notifications;
- disable email verification (in the "Administration > Authentication" section);
- be sure not to run any builds which change (e.g. deploy to) production environments. This also typically includes Maven builds deploying to non-local repositories. You can prevent any builds from starting by pausing the [build queue](#);
- disable cloud integration (so that it does not interfere with the main server);
- disable external artifact storage (as otherwise running/deleting builds and server cleanup will affect the storage which might be used by the production server);
- disable Git registry cleanup (or just disable cleanup on the server);
- disable Commit Status Publishing;
- disable any plugins which push data into other non-copied systems based on the TeamCity events;
- disable functionality to [store project settings in VCS](#): set `teamcity.versionedSettings.enabled=false` internal property;
- consider significantly increasing [VCS checking for changes interval](#) (server-wide default and overridden in the VCS roots) or changing settings of the VCS roots to prevent them from contacting production servers. Since TeamCity 10.0.3, see also [TW-47324](#).

See also the section below on [moving the server](#) from one machine to another.

## Move TeamCity Projects from One Server to Another

If you need to move data to a fresh server without existing data, it is recommended to [move the server](#) or [copy](#) it and then delete the data which is not necessary on the new server.

If you need to join the data with already existing set, there is a dedicated feature to move projects with most of the associated data from one server to another: [Projects Import](#).

### Notes on manual move of the settings in case you ever want to perform it

Since TeamCity 8.0 it is possible to move settings of a project or a build configuration to another server with simple file copying. For earlier TeamCity versions see the [comment](#).

The two TeamCity servers (source and target) should be of exactly the same version (same build).

All the [identifiers](#) throughout all the projects, build configurations and VCS roots of both servers should be unique. If they are not, you can change them via web UI.

If entities with the same id are present on different servers, the entities are assumed to be the same. For example this is useful for having global set of VCS roots on all the servers.

To move settings of the project and all its build configuration from one server to another:

From the [TeamCity Data Directory](#), copy the directories of corresponding projects (.BuildServer\config\projects\<id>) and all its parent projects to .BuildServer\config\projects of the target server.

This moves project settings, build configuration settings, VCS roots defined in the projects preserving the links between them.

If there are same-named files on the target server as those copied, this can happen in case of

- a) id match: same entities already exist on the target server, in which case the clashing files can be excluded from copying, or
- b) id clash: different entities happen to have same ids. In this case it should be resolved either by changing entity id on the source or target server to fulfill the uniqueness requirement.

The set of parent projects is to be identified manually based on the web UI or the directory names on disk (which by default will have the same prefix).

Note: It might make sense to keep the settings of the [root project](#) synchronized between all the servers (by synchronizing content of `.BuildServer\config\projects_Root` directory). For example, this will ensure same settings for the default cleanup policy on all the servers.

Further steps after projects copying might be:

- delete unused data in the copied parent projects (if any) on the target server
- use "server health" reports to identify duplicate VCS roots appeared in result of copying, if any
- archive the projects on the source server and adjust cleanup rules (to be able to see build's history, if necessary)

What is not copied by the approach above:

- pausing comment and user of the paused build configurations
- archiving user of the archived projects
- global server settings (e.g. Maven settings.xml profiles, tools (e.g. handle.exe), external change viewers, build queue priorities, issue trackers). These are stored under various files under `.BuildServer\config` directory and should be synchronized either on the file level or by configuring the same settings in the server administration UI.
- project association with agent pools
- templates from other projects which are not parents of the copied one. This configuration is actually deprecated in TeamCity 8.0 and is only supported as legacy. Templates used in several projects should be moved to the common parent project or root project.
- no data configured for the agents (build configurations allowed to run on the agent).
- no user-related or user group-related settings (like roles and notification rules)
- no state-related data like mutes and investigations, etc.

## Move TeamCity Installation to a New Machine

If you need to move the existing TeamCity installation to a new hardware or clean OS, it is recommended to follow the [instructions on copying](#) the server from one machine to another and then [switch](#) from the old server to a new one. If you plan to use the same database instance on the new server, just skip the items on copying the database. If you are sure you do not need to preserve the old server, you can perform move operations instead of copying in those instructions.

You can use the existing [license keys](#) when you move the server from one machine to another (as long as there are no two servers running at the same time). As license keys are stored under [TeamCity Data Directory](#), you transfer the license keys with all the other TeamCity settings data.

It is usually advised not to combine TeamCity update with any other actions, like environment or hardware changes, and perform the changes one at a time so that, if something goes wrong, the cause can be easily tracked.

### Switching from one server to another

Please note that [TeamCity Data Directory](#) and database should be used by a single TeamCity instance at any given moment. If you configured a new TeamCity instance to use the same data, please ensure you shutdown and disable the old TeamCity instance before starting the new one.

Generally it is recommended to use a domain name to access the server (in the agent configuration and when users access the TeamCity web UI). This way you can update the DNS entry to make the address resolve to the IP address of the new server and after all cached DNS results expire, all clients will automatically use the new server. You might need to reduce the DNS server cache/lease time in advance before the change to make the clients "understand" the change fast.

However, if you need to use another server domain address, you will need to:

- Switch agents to the new URL (requires updating the `serverUrl` property in `buildAgent.properties` on each agent). If you want to install agents anew but preserve agent's name and authentication status, you can install a new agent and copy the `conf\ buildAgent.properties` file from an old agent (checking that any paths in it are updated accordingly).
- Upon the new server startup, remember to update the [Server URL](#) on Administration | Global Settings page.
- Notify all TeamCity users on the new address

## Move TeamCity Agent

Apart from the binaries, TeamCity agent stores its configuration and data left from the builds it run. Usually the data from the previous builds makes preparation for the future builds a bit faster, but it can be deleted if necessary.  
The configuration is stored under `conf` and `launcher\conf` directories.

The data collected by previous build is stored under `work` and `system` directories.

The most simple way to move agent installation into a new machine or new location is to:

- stop existing agent
- [install](#) a new agent
- copy `conf/buildAgent.properties` from the old installation to a new one
- start the new agent.

With these steps the agent will be recognized by TeamCity server as the same and will perform [clean checkout](#) for all the

builds.

Please also review the [section](#) for a list of directories that can be deleted without affecting builds consistency.

## Share the Build number for Builds in a Chain Build

A build number can be shared for builds connected by a [snapshot dependency](#) or an [artifact dependency](#) using a reference to the following dependency property: `%dep.<btID>.system.build.number%`.

For example, you have build configurations A and B that you want to build in sync: use the same sources and take the same build number.

Do the following:

1. Create build configuration C, then snapshot dependencies: A on C and B on C.
2. Set the [Build number format](#) in A and B to:

```
%dep.<btID>.system.build.number%
```

Where `<btID>` is the [ID of the build configuration](#) C. The approach works best when builds reuse is turned off via the [D o not run new build if there is a suitable one](#) snapshot dependency option set to off.

[Read more](#) about dependency properties.

Please watch/comment the issue related to sharing a build number [TW-7745](#).

## Make Temporary Build Files Erased between the Builds

Update your build script to use path stored in  `${teamcity.build.tempDir}` (Ant's style name) property as the temp directory. TeamCity agent creates the [directory](#) before the build and deletes it right after the build.

## Clear Build Queue if It Has Too Many Builds due to a Configuration Error

Try pausing the build configuration that has the builds queued. On build configuration pausing all its builds are removed from the queue.

Also there is an ability to delete many builds from the build queue in a single dialog.

## Automatically create or change TeamCity build configuration settings

If you need a level of automation and web administration UI does not suite your needs, there several possibilities:

- use [REST API](#)
- change configuration files directly on disk (see more at [TeamCity Data Directory](#))
- write a TeamCity Java plugin that will perform the tasks using [open API](#).

## Attach Cucumber Reporter to Ant Build

If you use Cucumber for Java applications testing you should run cucumber with --expand and special --format options. Moreover you should specify RUBYLIB environment variable pointing on necessary TeamCity Rake Runner ruby scripts:

```

<target name="features">
  <java classname="org.jruby.Main" fork="true" failonerror="true">
    <classpath>
      <pathelement path="${jruby.home}/lib/jruby.jar"/>
      <pathelement path="${jruby.home}/lib/ruby/gems/1.8/gems/jyaml-0.0.1/lib/jyamlb.jar"/>
      ...
    </classpath>
    <jvmarg value="-Xmx512m"/>
    <jvmarg value="-XX:+HeapDumpOnOutOfMemoryError"/>
    <jvmarg value="-ea"/>
    <jvmarg value="-Djruby.home=${jruby.home}" />
    <arg value="-S"/>
    <arg value="cucumber"/>
    <arg value="--format"/>
    <arg value="Teamcity::Cucumber::Formatter"/>
    <arg value="--expand"/>
    <arg value=". "/>
    <env key="RUBYLIB"
         value="${agent.home.dir}/plugins/rake-runner/rb/patch/common${path.separator}${agent.home.dir}
         /plugins/rake-runner/rb/patch/bdd"/>
      <env key="TEAMCITY_RAKE_RUNNER_MODE" value="buildserver"/>
    </java>
  </target>

```

Please check the RUBYLIB path separator (';' for Windows, ':' for Linux, or '\${path.separator}' in ant for safety). If you are launching Cucumber tests using the Rake build language, TC will add all necessary command line parameters and environment variables automatically.

 This tip works in TeamCity version >= 5.0.

## Get Last Successful Build Number

Use URL like this:

```
http://<your TeamCity server>/app/rest/buildTypes/id:<ID of build configuration>/builds/status:SUCCESS/number
```

The build number will be returned as a plain-text response.

For <ID of build configuration>, see [Identifier](#).

This functionality is provided by [REST API](#)

## Set up Deployment for My Application in TeamCity

TeamCity has enough features to handle orchestration part of the deployments with the actual deployment logic configured in the build script / build runner. TeamCity supports a variety of generic build tools, so any specific tool can be run from within TeamCity. To ease specific tool usage, it is possible to wrap it into a meta-runner or write a custom plugin for that.

In general, setup steps for configuring deployments are:

1. Write a build script that will perform the deployment task for the binary files available on the disk. (e.g. use Ant or MSBuild for this. For typical deployment transports use [Deployer runners](#)). See also [#Integrate with Build and Reporting Tools](#). You can use [Meta-Runner](#) to reuse a script with convenient UI.
2. Create a build configuration in TeamCity that will execute the build script and perform the actual deployment. If the deployment is to be visible or startable only by the limited set of users, place the build configuration in a separate TeamCity project and make sure the users have appropriate permissions in the project.
3. In this build configuration configure [artifact dependency](#) on a build configuration that produces binaries that need to be deployed.
4. Configure one of the available triggers in the deploying build configuration if you need the deployment to be triggered automatically (e.g. to deploy last successful or last pinned build), or use "Promote" action in the build that produced the

- binaries to be deployed.
5. Consider using [snapshot dependencies](#) in addition to artifact ones and check [Build Chains](#) tab to get the overview of the builds. In this case artifact dependency should use "Build from the same chain" option.
  6. If you need to parametrize the deployment (e.g. specify different target machines in different runs), pass parameters to the build script using [custom build run dialog](#). Consider using [Typed Parameters](#) to make the custom run dialog easier to use or handle passwords.
  7. If the deploying build is triggered manually consider also adding commands in the build script to pin and tag the build being deployed (via sending a [REST API](#) request).
- You can also [use a build number](#) from the build that generated the artifact.

Further recommendations:

- Setup a separate build configurations for each target environment
- Use build's Dependencies tab for navigation between build producing the binaries and deploying builds/tasks
- If necessary, use parameter with "prompt" display mode to ask for "[confirmation](#)" on running a build
- [Change title](#) of the build configuration "Run" button

Related section on the official site: [Continuous Deployment with TeamCity](#)

## Use an External Tool that My Build Relies on

If you need to use specific external tool to be installed on a build agent to run your builds, you have the following options:

- Install and register the tool in TeamCity:
  1. Install the tool on all the agents that will run the build. This can be done manually or via an automated script. For simple file distribution also see [Installing Agent Tools](#)
  2. Add a property into `buildAgent.properties` file (or add environment variable to the system) with the tool home location as the value.
  3. Add agent requirement for the property in the build configuration.
  4. Use the property in the build script.
- Check in the tool into the version control and use relative paths.
- Add environment preparation stage into the build script to get the tool from elsewhere.
- Create a separate build configuration with a single "fake" build which would contain required files as artifacts, then use artifact dependencies to send files to the target build.

## Integrate with Build and Reporting Tools

If you have a build tool or a tool that generates some report/provides code metrics which is not yet [supported](#) by TeamCity or any of the [plugins](#), most probably you can use it in TeamCity even without dedicated integration.

The integration tasks involved are collecting the data in the scope of a build and then reporting the data to TeamCity so that they can be presented in the build results or in other ways.

### Data collection

The easiest way for a start is to modify your build scripts to make use of the selected tool and collect all the required data. If you can run the tool from a command line console, then you can run it in TeamCity with a [command line runner](#). This will give you detection of the messages printed into standard error output. The build can be marked as failed if the exit code is not zero or there is output to standard error via [build failure condition](#).

If the tool has launchers for any of the supported build scripting engines like Ant, Maven or MSBuild, then you can use corresponding runner in TeamCity to start the tool.

See also [#Use an External Tool that My Build Relies on](#) for the recommendations on how to run an external tool.

You can also consider creating a [Meta Runner](#) to let the tool have dedicated UI in TeamCity.

For an advanced integration a custom [TeamCity plugin](#) can be developed in Java to ease tool configuration and running.

### Presenting data in TeamCity

The build progress can be reported to TeamCity via [service messages](#) and build status text can also be [updated](#).

For testing tools, if they are not yet [supported](#) you can report tests progress to TeamCity from the build via [test-related service messages](#) or generate one of the supported [XML reports](#) in the build and let it be imported via a service message of configured XML Report Processing build feature.

To present the results for a generic report, the approach might be to generate HTML report in the build script, pack it into archive and publish as a build artifact. Then configure a [report tab](#) to display the HTML report as a tab on build's results.

A metrics value can be published as TeamCity statistics via [service message](#) and then displayed in a [custom chart](#). You can also configure [build failure condition](#) based on the metric.

If the tool reports code-attributing information like Inspections or Duplicates, TeamCity-bundled report can be used to display the results. A custom plugin will be necessary to process the tool-specific report into TeamCity-specific data model. Example of this can be found in [XML Test Reporting plugin](#) and [FXCop plugin](#) (see a link on [Open-source Bundled Plugins](#)).

See also [#Import coverage results in TeamCity](#).

For advanced integration, a custom plugin will be necessary to store and present the data as required. See [Developing TeamCity Plugins](#) for more information on plugin development.

## Restore Just Deleted Project

TeamCity moves deleted projects settings directories (which are named after the project id) to `<TeamCity Data Directory>/config/_trash` directory adding ".<internal ID>" suffix to the directories.

To restore a project, find the project directory in the `_trash` directory and move it into regular projects settings directory: `<TeamCity Data Directory>/config/projects` while removing the ".projectN" suffix from the directory name.

You can do this while server is running, it should pick up the restored project automatically.

Note that TeamCity preserves builds history and other data stored in the database for deleted projects/build configurations for 5 days after the deletion time. All the associated data (builds and test history, changes, etc.) is removed during the next clean-up after the [configurable](#) (5 days by default) timeout elapses.

The `config/_trash` directory is not cleaned automatically and can be emptied manually if you are sure you do not need the deleted projects. No server restart is required.

## Transfer 3 Default Agents to Another Server

This is not possible.

Each TeamCity server (Professional and Enterprise) allows using 3 or more agents bound to the server without any agent licenses.

In case of the Professional server, by default 3 agents are bound to the server instance: users do not pay for these agents, there is no license key for them.

In case of the Enterprise server, the number of agents depends on your package and the agents are bound to the server license key.

So, the agents bound to the server cannot be transferred to another server.

If you need more build agents that are included with your TeamCity server, you can purchase additional build agent licenses and connect more agents in addition to those that come bound with the server.

See [more](#) on licensing.

## Import coverage results in TeamCity

TeamCity comes bundled with IntelliJ IDEA/Emma and, JaCoCo coverage engines for Java and dotCover/NCover/PartCover for .NET.

However, there are plenty of other coverage tools out there, like Cobertura and others which are not directly supported by TeamCity.

In order to achieve similar experience with these tools you can:

- publish coverage HTML report as TeamCity artifact: most of the tools produce coverage report in HTML format, you can publish it as artifact and [configure report tab](#) to show it in TeamCity. If artifact is published in the root artifact directory and its name is `coverage.zip` and there is `index.html` file in it, report tab will be shown automatically. As to running an external tool, check [#Integrate with Build and Reporting Tools](#).
- extract coverage statistics from coverage report and publish [statistics values](#) to TeamCity with help of [service message](#): if you do so, you'll see coverage chart on build configuration Statistics tab and also you'll be able to fail a build with the help of a build failure condition on a metric change (for example, you can fail build if the coverage drops).



### Percentage values

You should not publish values `CodeCoverageB`, `CodeCoverageL`, `CodeCoverageM`, `CodeCoverageC` standing for block/line/method/class coverage percentage. TeamCity will calculate these values using their absolute parts. E.g. `CodeCoverageL` will be calculated as `CodeCoverageAbsLCovered` divided by `CodeCoverageAbsLTotal`.

You could publish these values but in this case they will lack decimal parts and will not be useful.

## Recover from "Data format of the data directory (NNN) and the database (MMM) do not match" error

If you get "Data format of the data directory (NNN) and the database (MMM) do not match." error on starting TeamCity, it means either the database or the TeamCity Data Directory were recently changed to an inconsistent state so they cannot be used together.

Double-check the database and data directory locations and change them if they are not those where the server used to store

the data.

If they are right, most probably it means that the server was upgraded with another database or data directory and the [consistent upgrade](#) requirement was not met for your main data directory and the database.

To recover from the state you will need backup of the consistent state made prior to the upgrade. You will need to restore that backup, ensure the right locations are used for the data directory and the database and perform the TeamCity upgrade.

## Debug a Build on a Specific Agent

In case a build fails on some agent, it is possible to debug it on this very agent to investigate agent-specific issues. Do the following:

1. Go to the Agents page in the TeamCity Web UI and [select the agent](#).
2. [Disable the agent](#) to temporarily remove it from the [build grid](#). Add a comment (optional). To enable the agent automatically after a certain time period, check the corresponding box and specify the time.
3. [Select the build](#) to debug.
4. Open the [Custom Run](#) dialog and specify the following options:
  - a. In the Agent drop-down, select the disabled agent.
  - b. It is recommended to select the run as [Personal Build](#) option to avoid intersection with regular builds.
5. When debugging is complete, enable the agent manually if automatic re-enabling has not been configured.

You can also perform [remote debugging](#) of tests on an agent via the [IntelliJ IDEA plugin](#) for TeamCity.

## Debug a Part of the Build (a build step)

If a build containing several steps fails at a certain step, it is possible to debug the step that breaks. Do the following:

1. Go to the build configuration and disable the build steps up to the one you want to debug.
2. [Select the build](#) to debug.
3. Open the [Custom Run](#) dialog and select the put the build to the [queue top](#) to give you build the priority.
4. When debugging is complete, re-enable the build steps.

## Vulnerabilities

This section describes effect and necessary protection steps related to the announced security vulnerabilities.

### Heartbleed, ShellShock

TeamCity distributions provided by JetBrains do not contain software/libraries and do not use technologies affected by Heartbleed and Shell shock vulnerabilities.

What might still need assessment is the specific TeamCity installation implementation which might use the components behind those provided/recommended by JetBrains and which can be vulnerable to the mentioned exploits.

### POODLE

If you configured HTTPS access to the TeamCity server, inspect the solution used for HTTPS as that might be affected (e.g. Tomcat seems to be [affected](#)). At this time none of TeamCity distributions include HTTPS access by default and investigating/eliminating HTTPS-related vulnerability is out of scope of TeamCity.

Depending on the settings used, TeamCity server (and agent) can establish HTTPS connections to other servers (e.g. Subversion). Depending on the server settings, those connections might fall back to using SSL 3.0 protocol. The recommended solution is not TeamCity specific and it is to disable SSLv3 on the target SSL-server side.

### GHOST

CVE-2015-0235 vulnerability is found in glibc library which is not directly used by TeamCity code. It is used by the Java/JRE used by TeamCity under \*nix platforms. As Java is not bundled with TeamCity distributions, you should apply the security measures recommended by the vendor of the Java you use. At this time there are no related Java-specific security advisories released, so updating the OS should be enough to eliminate the risk of the vulnerability exploitation.

### FREAK

CVE-2015-0204 vulnerability is found in OpenSSL implementation and TeamCity does not bundle any parts of OpenSSL product and so is not vulnerable. You might still need to review the environment in which TeamCity server and agents are installed as well as tools installed in addition to TeamCity for possible vulnerability mitigation steps necessary.

### Apache Struts

CVE-2017-5638 affects Jakarta Multipart parser in Apache Struts. CVE-2016-1181 also affects multipart requests processing in

some older versions of Apache Struts.

TeamCity bundles IntelliJ IDEA which contains jars from both: Apache Struts 1.x and Apache Struts 2.x. These jars are only used by IntelliJ IDEA Struts plugin when IntelliJ IDEA collects inspections for a project on a TeamCity agent.

But under no circumstances these versions of Apache Struts are used to handle any HTTP requests. Thus neither TeamCity server, nor TeamCity agent are affected by these vulnerabilities.

## Tomcat Under Windows

Based on the wording of the description of CVE-2017-12615, CVE-2017-12616 and CVE-2017-12617 TeamCity server installed under Windows is a potential subject for the attack. However, our analysis of the vulnerabilities indicates that these potential vulnerabilities cannot be exploited in the default TeamCity installation as the related configuration of Tomcat is inactive in all the TeamCity versions.

If necessary, Tomcat bundled with TeamCity can be [upgraded](#) to the version 7.0.82 which also removes the vulnerability from the Tomcat code.

## Tomcat CVE-2018-8037

TeamCity version 2018.1 is not vulnerable to the issue as it was identified and addressed in TeamCity codebase before official Tomcat announcement. Earlier TeamCity versions are vulnerable, so upgrade to TeamCity 2018.1+ is necessary.

## Watch Several TeamCity Servers with Windows Tray Notifier

TeamCity Tray Notifier is used normally to watch builds and receive notifications from a single TeamCity server. However, if you have more than one TeamCity server and want to monitor them with Windows Tray Notifier simultaneously, you need to start a separate instance of Tray Notifier for each of the servers from the command line with the `/allowMultiple` option:

- From the TeamCity Tray Notifier installation folder (by default, it's `C:\Program Files\JetBrains\TeamCity`) run the following command:

```
JetBrains.TrayNotifier.exe /allowMultiple
```

Optionally, for each of the Tray Notifier instances you can explicitly specify the URL of the server to connect using the `/server` option. Otherwise, for each further tray notifier instance you will need to log out and change the server's URL via the UI.

```
JetBrains.TrayNotifier.exe /allowMultiple /server:http://myTeamCityServer
```

See also [details](#) in the issue tracker.

## Personal User Data Processing

In relation to the TeamCity product, JetBrains does not collect any personal data of the on-premises TeamCity installation users. The related documents governing relationship of customers with JetBrains are available on the official web site: [privacy policy](#), [terms of purchase](#), [TeamCity license agreement](#).

The notes below can be useful when assessing how your usage of TeamCity complies with the General Data Protection Regulation (GDPR) (EU) 2016/679 regulation. These notes are meant to address the most basic questions and can serve as an input to the assessment of your specific TeamCity installation.

The notes are based on TeamCity 2017.2.4 which is actual at the moment of GDPR enforcement date. Please update your TeamCity instance at least to the version as previous versions might contain issues not in line with the notes below.

## TeamCity and Users' Personal Data

The most important user-related data stored by TeamCity is:

- full name and username - stored in the database and shown on the user's profile and whenever the user is referenced. When a user triggers a build, these are also stored in the build's parameters and passed into the build
- user's email - stored in the database and shown on the user's profile, used to send out notifications
- IP address of the clients accessing the server - can appear in the [internal logs](#)

TeamCity internal logs can also record some unstructured user-related information (e.g. submitted by the user or sent by the browser with the HTTP requests, retrieved according to the configured settings from the users source like LDAP)

## Deleting the User Data

When you want to delete personal data of a specific user, the best way to do it is to delete the user in TeamCity. This way all the references to the user will continue to store the numeric user id, while all the other user information will not be stored anymore. Note that [Audit](#) records will mention internal numeric user id after the user deletion.

If the user triggered any builds (i.e. had the "Run build" permission in any of the projects which are still present on the server), the user's username and full name were recorded in the build's "teamcity.build.triggeredBy" parameters as text values as those were part of the build's "environment". If you need to remove those, you can either delete the related builds (and all the builds which artifact- or snapshot-depend on them), or delete parameters of those affected builds (the parameters are stored in archived files under <TeamCity Data directory>\system\artifacts\*\*\*\teamcity\properties directories).

After the user deletion and other data cleaning, make sure to [reset search index](#) to prune possibly cached data of the deleted user from the search index.

### There are several other places which can hold user-related data

If user had "Edit project" permission, the full name / username can appear in:

- some audit entries (saved with TeamCity versions before 2017.2.1) - [TW-52215](#)
- some build logs in "Wait for pending persist tasks to complete" build log line (saved with TeamCity versions before 2017.2.1) - [TW-52872](#)
- commit comments in the repository storing versioned settings store name of the user doing the change in UI

VCS usernames in VCS-related data :

- in VCS changes visible in UI or stored in the database
- in the local .git repository clones on the server and agents

Username can also appear in access credentials configured in different integrations like VCS roots, issue tracker, database access, etc. (these are stored in the settings files and audit diff files in the TeamCity Data Directory and VCS roots usernames are also stored in the database for the current and previous versions of the VCS roots)

To ensure user's details are not stored by TeamCity you might want to check the TeamCity-backing storage that no occurrences of the data are stored: the database, Data Directory and the TeamCity home directory (logs, and memory dumps which are regularly placed under the "bin" directory).

## User Agreement

If you want the users to accept a special agreement before using your TeamCity instance, you can install a [dedicated plugin](#) developed by JetBrains for this purpose. Refer to the plugin's documentation for more details.

## Encryption

If you want to encrypt the data used by TeamCity, it is recommended to use generic, non-TeamCity-specific tools for this as TeamCity does not provide dedicated functionality.

TeamCity stores the data in the SQL database and on the file system.

You can configure the database to store the data in encrypted form and use secure JDBC-backed connection to the database (configured in the [database.properties](#)).

Also, you can configure encryption on the disk storage on the OS level.

## Logs and Debugging Data

If you want to ensure that you do not store the internal TeamCity logs for more than a limited amount of days, you can configure [internal logging](#) to rotate the log files each day and limit the number of files to keep. TeamCity agents generally do not operate with any user-related data in a structural way, but if you need to ensure the logs are regularly rotated on the agent, you will need to configure [agent logging](#) the same way.

Per-day rotation can be configured by adding `<param name="rotateOnDayChange" value="true"/>` line within all `<appender name="ROLL" class="jetbrains.buildServer.util.TCRollingFileAppender">` appenders. This change should be done to the default `conf\teamcity-server-log4j.xml` and also logging presets stored under <TeamCity Data Directory>\config\\_logging. TeamCity can also store diagnostics data like thread dumps which can record user-related data in unstructured way. It is recommended to review the content of the <TeamCity Home>\logs directory regularly and ensure that no old files are preserved in there. Also, the extra logs should be deleted after logging customization sessions like collecting debug logs, etc. There is a [known issue](#) that `logs\catalina.out` file is not rotated automatically at all. It is recommended to establish an automatic procedure to rotate the file regularly.

## Customizations

These notes only address bundled TeamCity functionality with the most common documented settings. You should assess your specific TeamCity installation considering customizations like the configured build scripts, installed plugins, external systems communicating with TeamCity via API, etc.

## TeamCity Release Cycle

The information below can be used for reference purposes only.

"major" release below means any release with a change in first or second version number (e.g. X in X.X.Z)  
"bugfix" release means releases with a change in the third version number (e.g. Z in X.X.Z)

Release stages that we generally have are:

Available under EAP (Early Access Program) - usually available only for major releases, starts several months after previous major release and usually months before the next major release. Typically new EAP releases are [published](#) on monthly or bi-monthly basis.

General Availability - as a rule, there is a major release each 8 months. There are multiple bugfix releases following the major release. Bugfix releases and support patches for critical issues (if applicable) are provided until "End of Sale" of the release.

End of Sale - occurs with the release of a new major version. After this time no bugfix updates or patches are usually provided (Exceptions are critical issues without a workaround which at the same time allow for relatively simple fix and inability for the customer to upgrade for an important reason). Only limited support is provided for these versions.

End of Support - occurs with the release of two newer major versions. At this point we stop providing regular technical support for the release.

The dates of previous releases and the sequence of TeamCity versions are listed on the [Previous Releases Downloads](#) page.

## Troubleshooting

When a problem occurs, you have a number of places to look for information after you've found out the problem isn't in setup:

- Check the [Known Issues](#) and [Common Problems](#) sections, collect relevant information using the [Reporting Issues](#) guidelines.
- [The TeamCity Forum](#) - Search the forum to see if anyone else has experienced your problem. Our forum's user base is quite active and is a good place to find support. If you cannot find any relevant information either in the forums or in tracker and you are not sure whether you faced a bug or it's just a result of misconfiguration, the right way to start is to create a new thread in the forums.
- [TeamCity's Issue Tracker](#) - Browse the issue tracker to see if somebody has already reported on your problem. If the same issue exists, please vote for the issue. If you are sure you have faced a bug, please [collect](#) the relevant data about the problem and post a new issue into the tracker. Be sure to include the TeamCity build number, describe where exactly you see the problem, what your previous actions were if relevant and also please describe your environment (OS, Web Server, TeamCity distribution used, how TeamCity is set up, etc.)
- [Contact us](#) to report an issue or ask a question using the general guidelines described.
- If you own Enterprise TeamCity license and need to submit information that is not meant to be public, you can also contact the development team via [Online Form](#) or [Feedback email](#).



When contacting us make sure to:

- include the affected TeamCity version;
- include detailed exact error messages, logs, screenshots;
- mention all related postings on the topic.

See also:

[Troubleshooting: Known Issues | Reporting Issues](#)

## Common Problems

- Most frequently used documentation sections
- Build works locally but fails or misbehaves in TeamCity
- Build is slow under TeamCity
- Started Build Agent is not available on the server to run builds
- Artifacts of a build are not cleaned
- Database-related issues
  - "out of memory" error with internal (HSQLDB) database
  - The transaction... log is full
  - The table 'table\_name' is full
  - Unable to extend ... segment ... in tablespace ...
  - NOCOUNT is enabled on MS SQL Server
  - MySQL JDBC driver error: PacketTooBigException
  - Database character set/collation-related problems
    - Character set/collation mismatch
    - TeamCity displays ???? instead of national symbols
    - "Unique key violations" or "Duplicates found" error on restore from backup
    - Resolve character set/collation-related problems
    - MySQL exception: Specified key was too long; max key length is 767 bytes
    - Index column size too large. The maximum column size is 767 bytes
  - 'This driver is not configured for integrated authentication' error with MS SQL database
  - Protocol violation error (Oracle only)
- Common Maven issues
- "Critical error in configuration file" errors
- TeamCity installation problems
- Problems with TeamCity NuGet Feed
- Problems with .Net-related TeamCity Tools
  - Startup performance issues
- Using tools requiring manual input, in particular Extended Validation code signing

## Most frequently used documentation sections

[Configuring server memory settings](#)

[Reporting server slowness issues](#)

[Back to top](#)

## Build works locally but fails or misbehaves in TeamCity

If a build fails or otherwise misbehaves in TeamCity but you believe it should not, the first thing to do is to check whether the issue is related to TeamCity or not.

To do that, follow this procedure:

Find a way to run the task from a command prompt. Make sure it works on the TeamCity agent machine, under the same user as the TeamCity agent runs under, with the same environment the agent receives. If necessary, run the TeamCity agent under a different user or tweak its environment. When the command runs OK, configure the same command in a TeamCity build using the command-line runner with the custom script setting. If that works, try other runner if that feels applicable.

Here are details on the approach:

Check that the build runs fine from the command prompt when run on the same machine as the TeamCity agent and under the same user that the agent is running, with the same environment variables and the same working directory, same architecture (32/64 bit) command line.

If the TeamCity build agent is run as a service (e.g. it is installed as a Windows service), try running the TeamCity agent under a regular user with administrative permissions [from the command line](#). See also [Windows Service limitations](#).

If this fixes the issue, you can try to figure out why running under the service is a problem for the build. Most often this is service-specific and is not related to TeamCity directly. Also, you can set up the TeamCity agent to be run from the console all the time (e.g. [configure](#) an automatic user logon and run the agent on the user logon).

Here are the detailed steps you can use to run a build from the command line:

Assuming you have a configured build in TeamCity which is failing, do the following:

- run the build in TeamCity and see it misbehaving
- disable the agent so that no other builds run on it. This can be done while the build is still in progress
- log in to the agent machine using the same user as the one running the TeamCity agent (check the right user in the machine processes list)
- stop the agent
- in a command line console, "cd" to the checkout directory of the build in question (the directory can be looked up in the beginning of the build log in TeamCity)
- run the build with a command line as you would do on a developer machine. This is runner-dependent. (For some

- runners you can look up the command line used by TeamCity in the build log, see also the `logs\teamcity-agent.log` agent log file for the command line used by TeamCity)
- if the build fails - investigate the reason as the issue is probably not TeamCity-related and should be investigated on the machine.
- if it runs OK, continue
- in the same console window "cd" to <TeamCity agent home>\bin and start TeamCity agent from there with the `agent start` command
- ensure the runner settings in TeamCity are appropriate and should generate the same command line as you used manually. e.g. use "Command Line" build step with "Custom script" option and the same command which can be saved in a .sh or .bat file and run from the command prompt
- run the build in TeamCity selecting the agent in the Run custom build dialog
- when finished, enable the agent

If the build succeeds from the console but still fails in TeamCity, please use a command line runner in TeamCity to launch the same command as in the console. If it still behaves differently in TeamCity, most probably this is an environment or a tool issue.

If the command line runner works but the dedicated runner does not while the options are all the same, please create a new issue in our [tracker](#) detailing the case. Please attach all the build step settings, the build log, all agent logs covering the build, the command you used in the console to run the build and the full console output of the build.

[Back to top](#)

## Build is slow under TeamCity

If you experience slow builds, the first thing to do is to check the build log to see if there are some long operations or the time is just spread over the entire process.

You can compare build logs of slower and faster builds to figure out what the difference is.

You can also run the build from the console on the same machine as detailed [above](#) to see if there is any difference between the build run from the console and the build in TeamCity.

If the slowness is spread over all the operations, the agent machine resources (CPU, disk, memory, network) are to be analyzed during the build to see if there is a bottleneck in any of those. If there is, the process loading the resource is to be found and investigated (e.g. with the help of the thread dump taken via "View thread dump" link on the running build results).

If there is some long operation and it is a TeamCity-related one (before start or after end of the actual build process), the TeamCity agent and server are to be analyzed (logs and thread dumps).

If you want to [turn to us](#) with the issue, please describe the visible effects, detail the process of investigation and attach the build log, full agent logs and other data collected.

## Started Build Agent is not available on the server to run builds

First start of agent after installation or TeamCity server upgrade/plugin installation can take time as agent downloads updates from the server and auto-upgrades.

Regularly, agent should become connected in 1 to 10 minutes, depending on the agent/server network connection speed.

If the agent is not connected within that time, check the name of the agent (as configured in `conf\buildAgent.properties` file) and check the tabs under the Agents server UI section:

- the agent is under Connected - the agent is ready to run builds
- the agent is under Disconnected - the agent was connected to the server, but became disconnected. Check the "Inactivity reason" in the table. If the reason is "Agent has unregistered (will upgrade)", then wait for several more minutes
- the agent is under Unauthorized - all the agents connected to the server for the first time should be authorized by a server administrator

If the agent stays in the state for more than 10 minutes and you have a fast network connection between the agent and the server, please:

- check the agent process is running and the `serverURL` in `conf\buildAgent.properties` is correct;
- check that all the [requirements](#) are met;
- check [agent logs](#) (`teamcity-agent.log`, `launcher.log`, `upgrade.log`) for any related messages/errors;
- check [server logs](#) (`teamcity-server.log`) for any messages/errors mentioning agent name or IP.

If you cannot find the cause of the delayed agent upgrade in the logs, [contact us](#) and provide the full agent and server logs. Please also check/include the state of the agent processes (java ones) on the agent machine.

[Back to top](#)

## Artifacts of a build are not cleaned

If you encounter a case when artifacts are preserved while they should have been removed by the server cleanup process, please check:

- the cleanup rules of the build configuration in question, artifacts cleanup section
  - presence of the icon "This build is used by other builds" in the build history line (prior to Pin action/icon on Build History)
  - build's Dependencies tab, "Delivered Artifacts" section. For every build configuration, check whether "Prevent dependency artifacts clean-up" is turned ON (this is default value). If it is, then the build's artifacts are not cleaned because of the setting.
- Read more on [cleanup settings](#).

[Back to top](#)

## Database-related issues

"out of memory" error with internal (HSQLDB) database

If during the TeamCity server start-up you encounter errors like:

- "error in script file line: ... out of memory"
- "java.sql.SQLException: out of memory",  
perform the following:
- try [increasing server memory](#). If this does not help, most probably this means that you have encountered internal database corruption. You can try to deal with this corruption using the [notes](#) based on the HSQLDB documentation.

A way to attempt a manual database restore:

- stop the TeamCity server
- backup the [<TeamCity Data Directory>/system/buildserver.data](#) file
- remove the [<TeamCity Data Directory>/system/buildserver.data](#) file and replace it with zero-size file of the same name
- start the TeamCity server

However, if the database does not recover automatically, chances that it can be fixed manually are minimal.

The internal (HSQL) database is not stable enough for production use and we highly recommend using an [external database](#) for TeamCity non-evaluation usage.

If you encountered database corruption, you can restore the last good backup or drop builds history and users, but preserve the settings, see [Migrating to an External Database#Switching to Another Database](#).

The transaction... log is full

This error can occur with an MS SQL or Sybase database. In this case we recommend increasing the transaction log for the TeamCity database. The log size can be 1 - 16 GB depending on the number of build agents in the system and the number of tests all agents report daily.

The table 'table\_name' is full

This error can occur with a MySQL database. The error indicates that the database has run out of free space either on the disk where the database files are located or in the temp directory.

Unable to extend ... segment ... in tablespace ...

This error can occur with an Oracle database. The error indicates that Oracle could not obtain more space for a table or an index as the database has run out of space on the disk or the specified quotas are insufficient.

NOCOUNT is enabled on MS SQL Server

`teamcity-server.log` reports that the unsupported "NOCOUNT" option is enabled on the MS SQL database server. Contact your DBA to disable the setting as described [here](#).

MySQL JDBC driver error: PacketTooBigException

The [ER\\_NET\\_PACKET\\_TOO\\_LARGE](#) error (PacketTooBigException / Packet for query is too large) is caused by the server-side `max_allowed_packet` configuration variable set to a low value or left at the default one.

 The variable controls the maximum size of MySQL communication buffer with 4MB being the default for Windows builds of MySQL 5.6, whereas in popular Linux distributions (e. g. Debian and Fedora Core), this variable defaults to 16MB, for both i686 and amd64 architectures.

1. Check the value of `max_allowed_packet` configuration variable by either examining `my.cnf` / `my.ini`,

or via

```
mysql> show variables like 'max_allowed_packet';
```

2. Increase (or explicitly set) the value of `max_allowed_packet` configuration variable in `my.cnf` or `my.ini`:

```
[mysqld]
max_allowed_packet=16M
```

 Setting a client-side value (via `maxAllowedPacket` connection property ) in the JDBC URL will have no effect, as this value cannot exceed the server-side limit.

## Database character set/collation-related problems

### Character set/collation mismatch

TeamCity reports character set/collation mismatch error: database tables/columns have a character set or collation that is not the same as the default character set or collation in your database schema. You may see this message if you are using a non-unicode character set as default for your database as TeamCity enforces unicode charset for some of the `varchar` fields. Make sure you configured the database according to our [recommendations](#).

### TeamCity displays ??? instead of national symbols

If you want to allow your local characters in texts in TeamCity (e.g. VCS messages, test names, user names, etc.), you need to migrate to a database with the appropriate character set.

### "Unique key violations" or "Duplicates found" error on restore from backup

TeamCity server restoration from a backup may fail with errors like "Unique key violation" or "Duplicates found" if the character sets (or collations) of the source and destination databases are not the same, and the cardinality of the destination character set is less than that of the source database.

To resolve the problem, select and set up the proper character set (and collation) for the destination database.

As for case sensitivity, the possible transitions are:

CS > CS

CI > CS

CI > CI

However, it is recommended to always use CS.

If the source character set is Unicode or UTF, the destination one must also be Unicode or UTF.

If the source character set is 8-bit non-UTF, the destination one can be the same or Unicode=UTF.

This applies to TeamCity 6.0 and above.

## Resolve character set/collation-related problems

To fix a problem, perform the following steps:

1. Create a new database with the appropriate character set and collation. We recommend using a unicode case-sensitive collation: see instructions for [PostgreSQL](#) and [MySQL](#). For MySQL, `utf8_bin` or `utf8mb4_bin` is preferred.



See also [PostgreSQL](#), [MySQL](#), [MS SQL](#) documentation for details on character set.

2. Copy the current `<TeamCity Data Directory>/config/database.properties` file, and change the database references in the copy to the newly created database.
3. Stop the TeamCity server.

4. Use the `maintainDB` tool to migrate to the new database:

```
maintainDB migrate [-A <path-to-data-dir>] -T <new-database-properties-file>
```

Depending on the size of your database, the migration may take from several minutes to several hours. For more information on the `maintainDB` tool, see [this section](#).

5. Upon the successful completion of the database migration, the `maintainDB` tool should update the `< TeamCity Data Directory >/config/database.properties` file with references to the new database. Ensure that the file has been updated. Edit the file manually if the tool fails to do it automatically.
6. Start the TeamCity server.

[Back to top](#)

MySQL exception: Specified key was too long; max key length is 767 bytes

Index column size too large. The maximum column size is 767 bytes

Check if the character set of your MySQL database. It is recommended to use the `utf8` character set.

If your database uses the `utf8mb4` character set (available since MySQL 5.5), set the following InnoDB configuration options (under `[mysqld]` section in `my.cnf` or `my.ini`) for TeamCity 2017.2.1+ to run:

- `innodb_large_prefix=1` for index key prefixes longer than 767 bytes (up to 3072 bytes) to be allowed for InnoDB tables that use DYNAMIC row format (deprecated in MySQL 5.7.7).
- `innodb_file_format=Barracuda` to enable the DYNAMIC row format.
- `innodb_file_per_table=1` to enable the Barracuda file format

The parameters above have the following default values:

	MySQL 5.5	MySQL 5.6	MySQL 5.7	MariaDB 5.5	MariaDB 10.0	MariaDB 10.1	MariaDB 10.2
<code>innodb_large_prefix</code>	0	0	1	0	0	0	1
<code>innodb_file_format</code>	Antelope	Antelope	Barracuda	Antelope	Antelope	Antelope	Barracuda
<code>innodb_file_per_table</code>	0	1	1	1	1	1	1

Depending on the database server version, the following configuration changes need to be made:

MySQL 5.5:

- `innodb_large_prefix=1`
- `innodb_file_format=Barracuda`
- `innodb_file_per_table=1`

MySQL 5.6 and MariaDB from 5.5 to 10.1:

- `innodb_file_format=Barracuda`
- `innodb_file_per_table=1`

For MySQL 5.7+ and MariaDB 10.2+ use the defaults, no changes are required.

[Back to top](#)

'This driver is not configured for integrated authentication' error with MS SQL database

During TeamCity installation, the following error might occur when connecting and creating the MS SQL database with Windows integrated security: "SQL error when doing: Taking a connection from the data source: This driver is not configured for integrated authentication."

The most common reason for the problem is the different bitness of the `sqljdbc_auth.dll` MS SQL shared library and the JRE used by TeamCity.

To solve the problem, do the following:

- a. Make sure you use the MS SQL native driver (downloadable from [the Microsoft Download Center](#)).
- b. Use the right JRE bitness — ensure that you are running TeamCity using Java with the same bitness as your `sqljdbc_auth.dll` MS SQL shared library.

By default, TeamCity uses the 32-bit Java. However, both 32-bit and 64-bit Java versions [can be used](#).

To run TeamCity with the required JRE, do one of the following:

- either set the `TEAMCITY_JRE` environment variable
- or remove the JRE bundled with TeamCity from `<TeamCity home>\jre` and set `JAVA_HOME`.



Note that on upgrade, TeamCity will overwrite the existing JRE with the default 32-bit version, so you'll have to update to the 64-bit JRE again after upgrade.

See also this related [external posting](#).

[Back to top](#)

## Protocol violation error (Oracle only)

This error can occur when the Oracle JDBC driver is not compatible with the Oracle server. For example, Oracle JDBC driver version 11.1 is not compatible with Oracle server version 10.2.

In order to resolve the problem, use the Oracle JDBC driver from your Oracle server installation, or [download the driver](#) of the same version as the Oracle server.

## Common Maven issues

There are two kinds of Maven-related issues commonly seen in the TeamCity build configurations:

- Error message on "Maven" tab of build configuration: "An error occurred during collecting Maven project information ..."
- Error message in build configuration with Maven dependencies trigger activated: "Unable to check for Maven dependency Update ..."

If the build configuration produces successful builds despite displaying such error messages, these errors are likely to be caused by the server-side Maven misconfiguration.

To collect information for the Maven tab, or to perform Maven dependencies check (for the trigger), TeamCity runs the embedded Maven. The execution is performed on the server machine, and any agent-side maven settings are not accessible. TeamCity resolves the `settings.xml` files on the server-side separately, as described [on this documentation page](#).

It makes sense to check if the server-side `settings.xml` files contain correct information about remote repositories, proxies, mirrors, profiles, credentials etc.

[Back to top](#)

## "Critical error in configuration file" errors

If you encounter the error, it means the settings stored in the TeamCity Data Directory are in inconsistent state. This can occur after manual change of the files or if newer version of TeamCity starts to report the inconsistencies.

To resolve the issue, you can edit the file noted in the message on the server file system. (make sure to create backup copy of the file before any manual edits). Usually server restart is not necessary for the changes to take effect.

VCS root with id "XXX" does not exist

The build configuration or template reference a VCS root which is not defined in the system.

Remedy actions: Restore the VCS root or create a new VCS root with the id noted or edit the file noted in the message to remove the reference to the VCS root.

[Back to top](#)

## TeamCity installation problems

If the TeamCity Web UI cannot be accessed after installation, you might be running TeamCity on a port that is already in use

by another program. Check and configure your TeamCity installation.

[Back to top](#)

## Problems with TeamCity NuGet Feed

If you are experiencing issues with partial TeamCity NuGet Feed, i.e. missing NuGet packages etc., you might have to reindex the TeamCity NuGet Feed.

To force TeamCity to reindex all available packages and reset the NuGet package list, navigate to the server Administration|Diagnostics | Caches and use the [buildsMetadata Reset](#) link.

For earlier versions, refer to [this section](#).

[Back to top](#)

## Problems with .Net-related TeamCity Tools

### Startup performance issues

After upgrade to TeamCity 9.0 or later, .NET Framework below version 4.0 installed on TeamCity agents may cause performance issues of .Net-related TeamCity tools due to Code access security (CAS) policy imposed by Microsoft.

To solve the issue, use one of the options:

- a. Add the following setting described in the [Microsoft documentation](#) to the `machine.config` file on all agents:

```
<configuration>
<runtime>
  <generatePublisherEvidence enabled="false"/>
</runtime>
</configuration>
```

You can modify the `machine.config` file as described in this [external blog post](#) and pass this config file to all agents, e.g. using a custom script.

- b. Alternatively, upgrade .Net Framework on the TeamCity agents to version 4.0 and above. Details are available in [the Microsoft documentation](#).

[Back to top](#)

## Using tools requiring manual input, in particular Extended Validation code signing

You might want to use tools which require some manual interaction during the build procedure executed on the TeamCity agent. This is not a TeamCity-specific problem, so it should be approached using generic means.

Under WIndows, you might want to configure TeamCity agent to run not as a Service, but with access to the desktop by configuring automatic user logon, [related details](#).

There is no simple solution for Extended Validation (EV) code signing as the feature is built in for a reason. There is some discussion on the issue on [stack overflow](#). The appropriate solutoin seems to implement a dedicated service with own authorization approach and sign the binariies through it.

[Back to top](#)

## Known Issues

This page contains a list of workarounds for known issues in TeamCity.

- Agent running as Windows Service Limitations
  - Security-related issues
  - Windows service limitations
    - Issues with automated GUI and browser testing
      - Running automated GUI tests and using RDP
      - Issues with . Net Selenium
    - Early start of the service before other resources are initialized

- [java.lang.OutOfMemoryError: unable to create new native thread error](#)
- [Clearing Browser Caches](#)
- [Logging with Log4j in Your Tests](#)
- [Agent Service Can Exit on User Logout under Windows x64](#)
- [Failed Build Can be Reported as a Successful One With Maven 2.0.7](#)
- [Conflicting Software](#)
- [Subversion issues](#)
  - [svn: E175002: Received fatal alert: bad\\_record\\_mac](#)
  - [Subversion-related JVM Crashes](#)
- [NUnit 2.4.6 Performance](#)
- [StarTeam Performance](#)
- [Perforce 2009.2 Performance on Windows](#)
- [Wrong times for build scheduled triggering \(Timezone issues\)](#)
- [Upgrading IntelliJ IDEA May Affect Active Pre-Tested Commits](#)
- [Other Java Applications Running on the Same Server](#)
- [The Server Does Not Start Claiming the Database is in Use](#)
- [Slow download from TeamCity server](#)
- [Failure to publish artifacts to server behind IIS reverse proxy](#)
- [Prerelease packages are not visible in the TeamCity NuGet feed](#)
- [Packages indexing is slow in TeamCity NuGet feed](#)
- [SSL problems when connecting to HTTPS from TeamCity \(handshake alert: unrecognized\\_name\)](#)
- [SSL problems connecting MS SQL Server and TeamCity](#)
- [Windows Docker Containers](#)
  - [Windows Docker containers](#)
  - [Information about installed Docker server OS on Windows missing on Agent](#)
  - [Linux Docker Containers under Windows](#)

## Agent running as Windows Service Limitations

When a TeamCity build agent is installed as a Windows service, there may appear various "Permission denied" or "Access denied" errors during the build process, see details below.

### Security-related issues

The user account used by the service is required to have sufficient permissions to perform the build and [manage the service](#). If you run the TeamCity agent service under the SYSTEM account, do the following:

1. [Change SYSTEM for a usual user account with necessary permissions granted](#).
2. [Restart the service](#).

### Windows service limitations

As a Windows service, the TeamCity agent and the build processes are not able to access network shares and mapped drives.

To overcome these restrictions, run TeamCity agent [via console](#).

### Issues with automated GUI and browser testing

These problems include errors running tests headless, issues with the interaction of the TeamCity agent with the Windows desktop, etc.

To resolve/ avoid these:

1. [Run TeamCity agent via console](#).
2. [Configure the build agent machine not to launch a screensaver locking the desktop](#).



Note that there is a Windows limitation to accessing a remote computer via mstsc: the desktop of the remote machine will be locked on RDP disconnect, which will cause issues running tests. The VNC protocol allows you to remote control another machine without locking it.  
To run GUI tests and be able to use RDP, see the [workaround](#) below.

3. [Configure the TeamCity agent to start automatically](#) (e.g. configure an automatic user logon on Windows start and then configure the TeamCity agent start (via agent.bat start) on the user logon).  
For graphical tests the build agent cannot be started as a service and it is recommended to configure the build agent launch with a 1 minute delay after the user auto-logon, e.g. using the "bin\agent.bat start" command in the task scheduler and configuring the delay there.

### Running automated GUI tests and using RDP

RDP uses its own video driver overriding the one from the machine's video card for the session. Redirecting the session to

console will unload the Windows graphical drivers. This can be done by adding the following step to your build configuration prior to your tests (the example below is for PowerShell, but other languages (DOS, Python) can be used too):

```
$sessionInfo=((quser $env:USERNAME | select -Skip 1) -split '\s+')
if ($sessionInfo[1] -like "rdp-tcp*") { tscon $sessionInfo[2] /dest:console }
```

where "quser [current username]" lists all the connections to that machine for the user, either console or graphical. The one listed as rdp-tcp#\* is the remote desktop connection which can be redirected to the console using "tscon [connection id] /dest:console".



An unsupervised computer with a running desktop permanently logged into a user session might be considered a network security threat, as access to it can be difficult to trace. Therefore, it is recommended to run automated GUI and browser tests on a virtual machine isolated from sensitive corporate network resources, e.g. on a machine not included in a Windows domain.

## Issues with .Net Selenium

When a TeamCity agent is started as a Windows service and automated tests for .Net applications use Selenium WebDriver, the tests may fail due to browser drivers limitations. As a solution, consider starting the agent [manually](#).

Early start of the service before other resources are initialized

To handle this, consider using the Automatic (Delayed Start) option of the service settings or configure service dependencies.

For more investigation steps, see the [Common Problems](#) page.

[Back to top](#)

`java.lang.OutOfMemoryError: unable to create new native thread error`

If your TeamCity server is running on SUSE® Linux Enterprise (or using systemd Daemon), the `java.lang.OutOfMemoryError: unable to create new native thread` error may be caused by the [cgroup process number controller](#) feature limiting the number of processes and the amount of threads in a cgroup to 512 by default.

Increasing the limit (e.g. to 4096) on the TeamCity server should solve the issue.

See also this [external posting](#).

[Back to top](#)

## Clearing Browser Caches

There is a web UI-related issue which some our users have encountered (and it cannot be reproduced on other computers) which is tied to the cached versions of content. If you have come across such problem, make sure your browser does not use cached versions of content by [clearing browser caches](#).

[Back to top](#)

## Logging with Log4j in Your Tests

If you use Log4j logging in your tests, in some cases you may miss the Log4j output from your logs. In such cases please do the following:

- Use Log4j 1.2.12
- For Log4j 1.2.13+, add the "Follow=true" parameter for the console appender used in Log4j configuration:

```
<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">
  <param name="Follow" value="true"/>
</appender>
```

[Back to top](#)

## Agent Service Can Exit on User Logout under Windows x64

The used version of [Java Service Wrapper](#) does not fully support Windows 64 and this causes agent launcher process to be killed on user logout. The agent itself will be function until the next restart (server upgrade or agent properties change).

[Back to top](#)

## Failed Build Can be Reported as a Successful One With Maven 2.0.7

This is a known bug in this version of Maven. Consider using any later version.

In case it's not possible you can patch mvn.bat yourself by replacing the fragment at line 148 of mvn.bat:

```
:error
set ERROR_CODE=1
```

with the following one:

```
:error
if "%OS%"=="Windows_NT" @endlocal
set ERROR_CODE=1
```

[Back to top](#)

## Conflicting Software

Most common indicators of conflicting software are errors like "Access is denied", "Permission denied" or java.io.FileNotFoundException mentioning the file that is present and is writable by the user the agent/build runs under. Also, certain software running in background (like antivirus) can significantly slow down build agent operations like sources checkout, artifact publishing or even build running.

Certain antivirus software like Kaspersky Internet Security can result in Java process crashes or other misbehavior like inability to access files. e.g. see [the issue](#).

ESET antivirus can also slow down Ant/IntelliJ IDEA project builds a great deal (slowing down TCP connections to localhost on agent).

If you run antivirus on the TeamCity server or agent machines and get disk access errors or experience degraded performance, please disable or better completely uninstall the antivirus software before investigating the issue and reporting the issue to JetBrains.

It is recommended to exclude entire TeamCity server home and [TeamCity Data Directory](#) from the background checks and perform periodical checks there in the well-known maintenance window so that those do not affect server performance much. On TeamCity agent, it is recommended to exclude TeamCity agent home from the background checks.

Please disable various indexing services. e.g. there might be problems with Windows Indexing Service. See [issue](#) for more details. Windows System Restore Feature might also need disabling.

Please also do not install software with background indexing like WinCVS, TortoiseCVS, TortoiseSVN and other Tortoise\* products. This applies to server and also to agents if you use agent-side checkout.

Skype software is known to:

- use port 80 on the system so you might not be able to use TeamCity server using default 80 port.
- corrupt layout of pages displayed in Internet Explorer. Internet Explorer Skype plugin is to blame. ([TW-13052](#)).

[Back to top](#)

## Subversion issues

svn: E175002: Received fatal alert: bad\_record\_mac

Please add system property `-Dsvnkit.http.sslProtocols=SSLv3,TLS` on the build server (see [Configuring TeamCity Server Startup Properties](#)).

If you use checkout on agent, add this property [on build agent](#) as well.

## Subversion-related JVM Crashes

If JVM crashes while executing SVN-related code (e.g. under `org.tmatesoft.svn` package), you can try to disable it by either:

- Passing `-Dsvnkit.useJNA=false` JVM option to the crashing process (server or agent), or
- Making NTLM support less prioritative by passing `-Dsvnkit.http.methods=Basic,Digest,NTLM` JVM option.

Anyway, upgrading the JVM used to the latest available version is recommended.

[Back to top](#)

## NUnit 2.4.6 Performance

Due to an issue in NUnit 2.4.6, its performance may be slower than NUnit 2.4.1. For additional information, please refer to the corresponding issue in our issue tracker: [TW-4709](#)

[Back to top](#)

## StarTeam Performance

Using StarTeam SDK 9.0 instead of StarTeam SDK 9.3 on the TeamCity server can significantly improve VCS performance when there is a slow connection between TeamCity and StarTeam servers.

[Back to top](#)

## Perforce 2009.2 Performance on Windows

If you run Perforce 2009.2 on Windows you may experience significant slow down. This is an issue with P4 server running on Windows. Please refer to corresponding section in Perforce documentation.

## Wrong times for build scheduled triggering (Timezone issues)

Please make sure you use the latest update for JDK 1.8 available for your platform (e.g. Oracle JDK [download](#)).

[Back to top](#)

## Upgrading IntelliJ IDEA May Affect Active Pre-Tested Commits

Before you upgrade to IntelliJ IDEA X (or other IntelliJ X platform products) please make sure you do not have active pre-tested commits, otherwise they will not be able to be committed after upgrade.

This is only relevant if you use directory-based IDEA project (project files are stored under `.idea` directory).

## Other Java Applications Running on the Same Server

If other web applications are available via the same hostname, a session cookie conflict can occur. This usually is visible via random user logouts or losing session-level data. (e.g. [TW-12654](#)). To resolve this, you can use different host names when accessing the applications.

[Back to top](#)

## The Server Does Not Start Claiming the Database is in Use

Only a single TeamCity server can work with one database, which is checked on the TeamCity server start. "The Database is in Use" error on the start-up is reported in either of the following cases:

- An attempt to start more than one TeamCity server connected to the same database
- A second TeamCity instance detected

- The internal HSQL database is being used by another application

The error is most probably caused by the fact that there is another running TeamCity installation which is connected to the same database. Check that the [database properties](#) are correct and there is no other TeamCity server using the same database.

In TeamCity 8.0 and earlier, if all the settings are correct, the error can occur when the TeamCity server or the database server has been shut down incorrectly.

The resolution depends on the database type:

- MySQL: restart the MySQL server and then start TeamCity again.
- PostgreSQL, Oracle, MS SQL: kill the connections from the incorrectly shut down TeamCity, and then start TeamCity again.
- Internal database (HSQL): remove the `buildserver.1ck` file from the [TeamCity Data Directory\system](#) directory, and then start TeamCity again.

[Back to top](#)

## Slow download from TeamCity server

If you experience slow speed when downloading artifacts from TeamCity, try checking the speed on the server machine, downloading from localhost.

If the speed is OK for the localhost, the issue can be in the network configuration or OS/hardware settings when combined with TeamCity(Tomcat) settings.

Please also make sure [<TeamCity Home>\conf\server.xml](#) file corresponds to the file included in TeamCity distribution (can be checked in .tar.gz distribution).

If you have the following "Connector" node (ports numbers can be different):

```
<Connector port="8111" protocol="HTTP/1.1"
           connectionTimeout="60000"
           redirectPort="8543"
           useBodyEncodingForURI="true"
        />
```

change it to:

```
<Connector port="8111" protocol="org.apache.coyote.http11.Http11NioProtocol"
           connectionTimeout="60000"
           redirectPort="8543"
           useBodyEncodingForURI="true"
           socket.txBufSize="64000"
           socket.rxBufSize="64000"
           tcpNoDelay="1"
        />
```

and restart TeamCity server.

If this does not help with the download speed, to investigate the case you might need to find an administrator with appropriate network-related issues investigation skills to look into the case.

## Failure to publish artifacts to server behind IIS reverse proxy

This problem is only relevant to configurations that involve IIS reverse proxy between build server and agents.

Sometimes a build agent can be found in infinite loop trying to publish build artifacts to server. Build log looks like this:

```
[11:15:05]Publishing artifacts
[11:15:05][Publishing artifacts] Collecting files to publish: [toZip/** => artifact.zip]
[11:15:06][Publishing artifacts] Creating archive artifact.zip (9s)
[11:15:06][Creating archive artifact.zip] Creating
C:\BuildAgent\temp\buildTmp\ZipPreprocessor2847146024236637664\artifact.zip
[11:15:15][Creating archive artifact.zip] Archive was created, file size 32142324 bytes
[11:15:15][Publishing artifacts] Sending toZip/**
[11:15:25][Publishing artifacts] Sending toZip/**
[11:15:39][Publishing artifacts] Sending toZip/**
[11:15:48][Publishing artifacts] Sending toZip/**
[11:16:01][Publishing artifacts] Sending toZip/**
[11:16:16][Publishing artifacts] Sending toZip/**
```

meanwhile teamcity-agent.log is filled with 404 responses from IIS:

```
[2012-08-01 12:04:55,514]  WARN - jetbrains.buildServer.AGENT - <!DOCTYPE html PUBLIC
"-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
<title>404 - File or directory not found.</title>
<style type="text/css">
<!--
body{margin:0;font-size:.7em;font-family:Verdana, Arial, Helvetica,
sans-serif;background:#EEEEEE;}
fieldset{padding:0 15px 10px 15px;}</style>
```

The most common cause for this is maxAllowedContentLength setting (in IIS) either

- is set to too small value
- is left unconfigured and so defaults to 30000000 bytes (<30 Mb)

So any artifact larger than maxAllowedContentLength is discarded by IIS

Check the settings value and try to rerun your build

Prerelease packages are not visible in the TeamCity NuGet feed

Problem. Prerelease packages are not visible in the TeamCity NuGet feed.

Cause. NuGet clients prior to version 3 fail to list prerelease packages if the package version violates [the required format](#).

Solution: Delete build artifacts whose versions violate [the required format](#).

Packages indexing is slow in TeamCity NuGet feed

Problem. After TeamCity server host machine move or upgrade to the TeamCity 2017.1 build metadata can be reset.

Cause. The TeamCity NuGet feed relies on build metadata, and packages re-indexing can take a lot of time depending on the number of packages and the idle time of the TeamCity server.

Solution. To speed up build metadata re-indexing, specify following [internal properties](#):

```
teamcity.buildIndexer.indexPackSize=1000
teamcity.buildIndexer.packSleepDurationMs=10
```

To check the metadata indexing progress, look for lines similar to the ones below in the [teamcity-server.log](#) file:

INFO - .index.BuildIndexer (metadata) - Enqueued next 100 builds for indexing, builds left: 7064, last build id: 8142

After re-indexing is complete, remove these internal properties.

## SSL problems when connecting to HTTPS from TeamCity (handshake alert: unrecognized\_name)

This problem may happen when changing JVM from 1.6 to 1.7 and connecting some incorrectly configured HTTPS servers. The problem and workaround for it are described in this issue: <http://youtrack.jetbrains.com/issue/TW-30210>

## SSL problems connecting MS SQL Server and TeamCity

Since MS SQL Server always establishes an SSL connection between the jdbc client (the TeamCity application) and the server, connection problems may occur (SQL exception: Connection reset).

Affected MS SQL versions: Any version prior to the ones listed below:

- SQL Server 2012 Production version and later
- SQL Server 2008 Service Pack 3 Cumulative Update 4
- SQL Server 2008R2 Service Pack 1 Cumulative Update 6
- SQL Server 2008R2 Service Pack 2

Cause: The problem is caused by the Java 1.6 update addressing [this security vulnerability](#).

Solution: Microsoft SQL Server upgrade is recommended.

Workarounds: If Microsoft SQL Server upgrade is not possible for some reason, TeamCity can be set up to use older Microsoft SQL Server versions with the database connection still SSL- or TLS-encrypted.

 Any of the two workarounds listed below will make the connection between TeamCity and the database server vulnerable

- Continue using a block cipher such as AES\_128\_CBC or 3DES\_EDE\_CBC, but disable CBC protection via `-Djsse.enableCBCProtection=false` Java command-line option (that can be added to TEAMCITY\_SERVER\_OPTS environment variable, as described [here](#)). The `jsse.enableCBCProtection` Java system property is also available in all OpenJDK 8 versions and IBM J9 8.0.0 SR1 and later. Secure connection between TeamCity and Microsoft SQL Server would be stable but still vulnerable to [CVE-2011-3389](#) also known as BEAST.
- Fall back to a stream cipher (which is not susceptible to BEAST) such as RC4\_128. This will render the connection vulnerable to [CVE-2015-2808](#).

Please try running with antivirus software uninstalled before reporting the issue to JetBrains. e.g. see [the issue](#).

## Windows Docker Containers

### Windows Docker containers

- Currently, they [do not support port mapping from containers to localhost](#), so the `-p` option does not have any effect for localhost. However, it works for the non-localhost IP address associated with this machine and you can access a running application via the machine's hostname or determine the IP address via the `ipconfig` command. Note that the `netstat -an` command may not show that the port is open on any IP address, while in fact it can work. This is also a known problem of Docker on Windows.
- On Windows 10, the memory allocated per container is 1GB by default. To increase this value, use the following memory options:

```
docker run ... -m 2GB -e TEAMCITY_SERVER_MEM_OPTS="-Xmx2g  
-XX:ReservedCodeCacheSize=350m"
```

- On Windows 10 containers work via Hyper-V and may experience problems with network and other subsystems. To diagnose these problems, execute the following PowerShell script:

```
Invoke-WebRequest https://aka.ms/Debug-ContainerHost.ps1 -UseBasicParsing |  
Invoke-Expression
```

- When starting a TeamCity server from a Windows Docker image, make sure to grant Authenticated Users Full control over the directories used as volumes. [See the related issue](#).
- When starting a Windows Docker container with the directory C:/BuildAgent/work mapped as a volume to the container host, Git for Windows fails with a following error:  
"Invalid path '/ContainerMappedDirectories': No such file or directory". The workaround is not to add "C:/BuildAgent/work" as a volume.

To analyze the script output, please refer to the [following documents](#). If it shows that there are problems with the container network subsystem, try resetting it using the [cleanup scripts](#).

More details on troubleshooting Docker for Windows are available in the [Docker](#) and [Microsoft](#) documentation.

Information about installed Docker server OS on Windows missing on Agent

On Windows 10, the Docker server depends on Hyper-V service and its start may take a significant amount of time. To resolve the issue, configure the TCBuildAgent Windows service to depend on the Docker for Windows Service, "com.docker.service" by default.

Linux Docker Containers under Windows

Since TeamCity 2017.2, the Docker Wrapper works on Windows when Windows-based containers are started.

If a Linux container is started on a Windows machine, TeamCity displays the error message "Starting Linux Docker containers under Windows is not supported. To avoid this problem, add the '[teamcity.agent.jvm.os.name](#) does not contain Windows' agent requirement.

If you need to support a use case when the Docker wrapper runs Linux containers under Windows platform, please vote for /comment on [TW-51820](#).

## Reporting Issues

If you experience problems running TeamCity and believe they are related to the software, please [contact us](#) with a detailed description of the issue.

To fix a problem, we may need a wide range of information about your system as well as various logs. The section below explains how to collect such information for different issues.

In this section:

- [Best Practices When Reporting Issues](#)
- [Slowness, Hangings and Low Performance](#)
  - [Server Thread Dump](#)

- Collecting CPU Profiling Data on Server
- Agent Thread Dump
- Taking Thread Dump
- Database-related Slowdowns
- OutOfMemory Problems
- "Too many open files" Error
- Agent does not connect to the server
- Logging events
  - Version Control debug logging
- Patch Application Problems
- Logging for .NET Runners
- Remote Run Problems
- Logging in IntelliJ IDEA/Platform-based IDEs
  - Open in IDE Functionality Logging
  - No Suitable Build Configurations Found for Remote Run
- Logging in TeamCity Eclipse plugin
- TeamCity Visual Studio Addin issues
  - TeamCity Addin logging
  - Visual Studio logging
- dotCover Issues
- JVM Crashes
- Build Log Issues
- IntelliJ IDEA Inspections
- Uploading Large Data Archives

## Best Practices When Reporting Issues

Following these guidelines will ensure timely response and effective issue resolution. Check [Feedback](#) for appropriate ways to contact us.



- note the TeamCity version in use, including the build number (can be found in the footer and the `teamcity-server.log`). Consider checking if the issue is still relevant in the most recent TeamCity version;
- do not create duplicate postings, if you still do, make sure to note all previous postings on the same topic you have made or found;
- do not combine several issues into one posting, post the most important issue first, or post several issues noting others if they are related;
- note the pattern of issue occurrence (first time, recurring), how it was mitigated before, whether there were any recent environment changes;
- be specific: note exact times, error messages, etc.;
- describe the expected and actual behavior;
- detail the related settings configured in TeamCity (include screenshots, settings files, actual values, REST API entity representations);
- include related screenshots of the TeamCity UI (always include the entire page and the browser URL in the capture);
- include related text messages as text (not as image), include the messages with all the details;
- when reporting build procedure issues, try to [reproduce](#) them without TeamCity on the agent machine in the same environment and let us know the results;
- do not include large portions (above 10Kb) of the textual data into the email text, rather attach it in a file;
- attach/[upload](#) archive with TeamCity server logs (see [details](#), ideally, the entire `<server home>\logs` directory with all the directories inside. If impractical, all the files updated around or later than the issue time); if related to the build-time or agent behavior, attach the entire `<agent home>\logs` directory and the build logs for the related build (downloaded via dedicated link from the build results);
- for performance/slowness/delays issues, take a set of (10+ spread across the issue time) server or agent [thread dumps](#) during the issue occurrence and make sure to send us the dumps;
- when sending files greater than 500Kb in size or more than three files, package them into a single archive;
- when replacing/masking data in logs, note the replacements patterns used;
- note if there is an anti-virus installed and if there is a network proxy or reverse proxy in front of the TeamCity server;
- when relevant, note the OS, versions of any manually installed components like Java, the TeamCity distribution used (`.exe`, `.tar.gz`)
- note any customizations/not standard environment settings;
- list any non-bundled plugins installed;
- if any, note the previous manual modifications of the TeamCity Data Directory or the database;
- when suggesting an improvement or feature or asking for settings advice, detail why you need the feature and what the original goal you want to achieve is. Suggestions as to how you would like to see the feature are welcome too;
- check the sections below for common cases and specific information to collect and send to us.

## Slowness, Hangings and Low Performance

If TeamCity is running slower than you would expect, please use the notes below to locate the slow process and send us all the relevant details if the process is a TeamCity one.

### Determine Which Process Is Slow

If you experience a slow TeamCity web UI response, checking for changes process, server-side sources checkout, long cleanup times or other slow server activity, your target should be the machine where the TeamCity server is installed.

If the issue is related only to a single build, you will also need to investigate the TeamCity agent machine which is running the build.

Investigate the system resources (CPU, memory, IO) load. If there is a high load, determine the process causing it. If it is not a TeamCity-related process, that might need addressing outside of the TeamCity scope. Also check for generic slow-down reasons like anti-virus software, etc.

If it is the TeamCity server that is loading the CPU/IO or there is no substantial CPU/IO load and everything runs just fine except for TeamCity, then this is to be investigated further.

Please check if you have any [Conflicting Software](#) like anti-virus running on the machine and disable/uninstall it.

If you have a substantial TeamCity installation, please check you have appropriate [memory settings](#) as the first step.

### Collect Data

During the slow operation, take several thread dumps of the slow process (see below for thread dump taking approaches) with 5-10 seconds interval. If the slowness continues, please take several more thread dumps (e.g. 3-5 within several minutes) and then repeat after some time (e.g. 10 minutes) while the process is still being slow.

Then [send](#) us a detailed description of the issue accompanied with the thread dumps and full server (or agent) [logs](#) covering the issue. Unless it is undesirable for some reason, the preferred way is to file an issue into our [issue tracker](#) and let us know via feedback email. Please include all the relevant details of investigation, including the CPU/IO load information, what specifically is slow and what is not, note URLs, visible effects, etc.

For large amounts of data, please use [our FTP](#).

### Server Thread Dump

When an operation on the server is slow, please take a set of the server thread dumps (10+) spread over the time of the slowness. TeamCity automatically saves thread dumps on super slow operations. so there might already be some saved in logs/threadDumps-<date> directories.

It is recommended to send us an archive of the entire content of server's <[TeamCity Home](#)>/logs/threadDumps-<date> directories for all the recent dates.

It is recommended that you take a thread dump of the TeamCity server from the Web UI (if the hanging is local and you can still open the TeamCity Administration pages): go to the Administration | Server Administration | Diagnostics page and click the Save Thread Dump button to save a dump under the <[TeamCity Home](#)>/logs/threadDumps-<date> directory (where you can later download the files from "Server Logs").

If the server is fully started but web UI is not responsive, try the [direct URL](#) using the actual URL of your TeamCity server.

If the UI is not accessible (or the server is not yet fully started), you can take a server thread dump manually using the approaches described [below](#).

You can also adjust the `teamcity.diagnostics.requestTime.threshold.ms=30000` [internal property](#) to change the timeout after which a thread dump is automatically created in the `threadDumps-<date>` directory under TeamCity logs whenever there is a user-originated web request taking longer than timeout.

### Collecting CPU Profiling Data on Server

If you experience degraded server performance and the TeamCity server process is producing a large CPU load, take a CPU profiling snapshot and send it to us accompanied with the detailed description of what you were doing and what your system setup is.

You can take CPU profiling and memory snapshots by installing the [server profiling plugin](#) and following the instructions on the plugin page.

Here are some hints to get the best results from CPU profiling:

- after starting the server, wait for some time to allow it to "warm up". This can take from 5 to 20 minutes depending on the data volume that TeamCity stores.
- when a CPU usage increase is found on the server, please try to indicate what actions cause the load.
- start CPU profiling and repeat the action several times (5 - 10).
- capture a snapshot.
- archive the snapshot and send it to us including the description of the actions that cause the CPU load.

## Agent Thread Dump

It is recommended that you take an agent thread dump from the Web UI: go to the Agent page, Agent Summary tab, and use the Dump threads on agent action.

If the UI is not accessible, you can take the dump thread manually using the approaches described [below](#). Note that the TeamCity agent consists of two java processes: the launcher and agent itself. The agent is triggered by the launcher. You will usually be interested in the agent (nested) process and not the launcher one.

## Taking Thread Dump

These can help if you are unable to take a thread dump from the TeamCity web UI.

To take a thread dump:

Under Windows

You have several options:

- To take a server thread dump if the server is run from the console, press Ctrl+Break (Ctrl+Pause on some keyboards) in the console window (this will not work for an agent, since its console belongs to the launcher process). If the server is run as a service, try running it from console by logging under the same user as configured in the service and executing "<TeamCity server home>\bin\teamcity-server.bat run" command.

Another approach is to figure out the process id of the TeamCity server process (it's the top-most "java" process with "org.apache.catalina.startup.Bootstrap start" at the end of the command line and use one of the following approaches:

- run jstack <pid\_of\_java\_process> in the bin directory of the Java installation used to be by the process (the Java home can be looked up in the process command line. If the installation does not have jstack utility, you might need to get the java version via java -version command, download full JDK of the same version and use "jstack" utility from there). You might also need to supply "-F" flag to the command.
- use TeamCity-bundled agent thread dump tool (can be found in the agent's plugins). Run the command:

```
<TeamCity agent>\plugins\stacktracesPlugin\bin\x86\JetBrains.TeamCity.Injector.exe  
<pid_of_java_process>
```

Note that if the hanging process is run as a service, the thread dumping tool must be run from a console with elevated permissions (using Run as Administrator). If the service is run under System account, you might also need to launch the thread dumping tools via "[PsExec.exe -s <path to the tool>\<tool> <options>](#)". When the service is run under a regular user, wrapping the tool invocation in [PsExec.exe -u <user> -p <password> <path to the tool>\<tool> <options>](#) might also help.

If neither of these work for the server running as a service, try [running the server from console](#) and not as a service. This way the first (Ctrl+Break) option can be used.

Under Linux

- run jstack <pid\_of\_java\_process> (using jstack from the Java installation used by the process) or kill -3 <pid\_of\_java\_process>. In the latter case output will appear in <TeamCity Home>/logs/catalina.out or <TeamCity agent home>/logs/error.log.

See also [Server Performance](#) section above.

## Database-related Slowdowns

When the server is slow, check if the problem is caused by database operations.

It is recommended to use database-specific tools.

You can also use the `debug-sql` server [logging preset](#). Upon enabling, all the queries which take longer 1 second will be logged into the `teamcity-sql.log` file. The time can be changed by setting the `teamcity.sqlLog.slowQuery.threshold` [internal property](#). The value should be set in milliseconds and is 1000 by default.

## MySQL

Along with the server thread dump, please attach the output of the "show processlist;" SQL command executed in MySQL console. Like with thread dumps, it makes sense to execute the command several times if slowness occurred and send us the

output.

Also, MySQL can be set up to keep a log of long queries executed with the changes in `my.ini`:

```
[mysqld]
...
log-slow-queries
long_query_time=15
```

The log can also be sent to us for analysis.

[Back to top](#)

## OutOfMemory Problems

If you experience problems with TeamCity "eating" too much memory or OutOfMemoryError/"Java heap space" errors in the log, please do the following:

- Determine what process encounters the error (the actual building process, the TeamCity server, or the TeamCity agent). You can track memory and CPU usage by TeamCity with the charts on the Administration | Server Administration | Diagnostics page of your TeamCity web UI.
- If the server is to blame, please check you have increased memory settings from the default ones for using the server in production (see [the section](#)).
- If the build process is to blame, set "JVM Command Line Parameters" settings in the build runner. Increase value for '`-Xmx`' JVM option, like `-Xmx1200m`, e.g. Java Inspections builds may specifically need to increase `-Xmx` value.
- If the TeamCity server is to blame and increasing the memory size does not help, please report the case for us to investigate. For this, while the server is high on memory consumption, take several server thread dumps as described [a above](#), get the memory dump (see below) and all the server logs including `threadDumps-*` sub-directories, archive the results and [send them to us](#) for further analysis. Make sure that `Xmx` setting is less than 8Gb before getting the dump:
  - if a memory dump (`hprof` file) is created automatically the `java_xxx.hprof` file is be created in the process startup directory (`<TeamCity Home>/bin` or `<TeamCity Agent home>/bin`);
  - for the server, you can also take memory dump manually when the memory usage is at its peak. Go to the Administration | Server Administration | Diagnostics page of your TeamCity web UI and click Dump Memory Snapshot .
  - another approach to take a memory dump manually is to use the `jmap` standard JVM command line utility of the full JVM installation of the same version as the Java used by the process. Example command line is:  
`jmap -dump:file=<file_on_disk_to_save_dump_into>.hprof <pid_of_the_process>`

See how to change JVM options for the [server](#) and for [agents](#).

[Back to top](#)

## "Too many open files" Error

1. Determine what computer it occurs on
2. Determine the process which has opened a lot of files and the files list (on Linux use `lsof`, on Windows you can use `handle` or [TCPView](#) for listing sockets)
3. If the number is under thousands, check the OS and the process limits on the file handles (on Linux use `ulimit -n`) and increase them if necessary. Please note that default Linux 1024 handles per process is way too small for a server application like TeamCity. Please increase the number to at least 16000. Please check the actual process limits after the change as there are different settings in the OS for settings global and per-session limits (e.g. see [the post](#))

If the number of files is large and looks suspicious and the locking process is a TeamCity one (the TeamCity agent or server with no other web applications running), then, while the issue is still occurring, grab the list of open handles several times with several minutes interval and send the result to us for investigation together with the relevant details.

Please note that you will most probably need to reboot the machine with the error after the investigation to restore normal functioning of the applications.

## Agent does not connect to the server

Please refer to [Common Problems](#)

## Logging events

The TeamCity server and agent create logs that can be used to investigate issues.



### How to enable DEBUG logging on server?

Before reproducing the problem it makes sense to enable 'DEBUG' log level for TeamCity classes.

On the server side, go to the Administration | Server Administration | Diagnostics page and select logging preset ('debug-all', 'debug-vcs', etc).

After that, DEBUG messages will go to `teamcity-*.log` files ([read more](#)).

For detailed information, please refer to the corresponding sections:

[TeamCity Server Logs](#)

[Viewing Build Agent Logs](#)

[Back to top](#)

### Version Control debug logging



To enable VCS logging on the server side, switch logging preset to "debug-vcs" in administration web UI and then retrieve `logs/teamcity-vcs.log` log file.

Most VCS operations occur on the TeamCity server, but if you're using the [agent-side checkout](#), VCS checkout occurs on the build agents.

For the agent and the server, you can change the Log4j configuration manually in `<TeamCity Home>/conf/teamcity-server-log4j.xml` or `<BuildAgent home>/conf/teamcity-agent-log4j.xml` files to include the following fragment:

```
<category name="jetbrains.buildServer.VCS" additivity="false">
    <appender-ref ref="ROLL.VCS"/>
    <appender-ref ref="CONSOLE-ERROR"/>
    <priority value="DEBUG"/>
</category>

<category name="jetbrains.buildServer.buildTriggers.vcs" additivity="false">
    <appender-ref ref="ROLL.VCS"/>
    <appender-ref ref="CONSOLE-ERROR"/>
    <priority value="DEBUG"/>
</category>
```

Please also update the `<appender name="ROLL.VCS"` node to increase the number of the files to store:

```
<param name="maxBackupIndex" value="30"/>
```

If there are separate logging options for specific version controls, they are described below.

### Subversion debug logging



To enable SVN logging on the server side, switch the logging preset to "debug-SVN" in the administration web UI and then retrieve the `logs/teamcity-vcs.log` and `logs/teamcity-svn.log` files.

An alternative manual approach is also necessary for agent-side logging.

First, please enable the generic VCS debug logging, as described [above](#).

Uncomment the SVN-related parts (the `SVN.LOG` appender and `javasvn.output` category) of the Log4j configuration file on the server and on the agent (if the [agent-side checkout](#) is used). The log will be saved to the `logs/teamcity-svn.log` file. Generic VCS log should be also taken from `logs/teamcity-vcs.log`.

### ClearCase

Uncomment the Clearcase-related lines in the `<TeamCity Home>/conf/teamcity-server-log4j.xml` file. The log will be saved to `logs/teamcity-clearcase.log` directory.

## Patch Application Problems

In case the [server-side checkout](#) is used, the "patch" that is passed from the server to the agent can be retrieved by:

- add property `system.agent.save.patch=true` to the build configuration.
- trigger the build.

the build log and the agent log will contain the line "Patch is saved to file \${file.name}"

Get the file and provide it with the problem description.

[Back to top](#)

## Logging for .NET Runners

To investigate process launch issues for .Net-related runners, enable debugging as described below. The detailed information will then be printed into the build log. It is recommended not to have the debug logging for a long time and revert the settings after investigation.

Add the `teamcity.agent.dotnet.debug=true` configuration parameter in the build configuration or on the agent and run the build.

▼ [Alternative way to enable the logging](#)

1. Open the `<agent home>/plugins/dotnetPlugin/bin` folder.
2. Make a backup copy of `teamcity-log4net.xml`
3. Replace `teamcity-log4net.xml` with the content of `teamcity-log4net-debug.xml`



After a debug log is created, it is recommended to roll back the change.

The change in the `teamcity-log4net.xml` will be removed on the build agent autoupgrade.

[Back to top](#)

## Remote Run Problems

The changes that are sent from the IDE to the server on a [remote run](#) can be retrieved from the server `.BuildServer/system/changes` directory. Locate the `<change_number>.changes` file that corresponds to your change (you can pick the latest number available or deduce the URL of the change from the web UI).

The file contains the patch in the binary form. Please provide it with the problem description.

[Back to top](#)

## Logging in IntelliJ IDEA/Platform-based IDEs

To enable debug logging for the [IntelliJ Platform-based IDE plugin](#), include the following fragment into the Log4j configuration of the `<IDE home>/bin/log.xml` file:

```

<appender name="TC-FILE" class="org.apache.log4j.RollingFileAppender">
    <param name="MaxFileSize" value="10Mb"/>
    <param name="MaxBackupIndex" value="10"/>
    <param name="file" value="$LOG_DIR$/idea-teamcity.log"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d [%r] %6p - %30.30c - %m \n"/>
    </layout>
</appender>

<appender name="TC-XMLRPC-FILE" class="org.apache.log4j.RollingFileAppender">
    <param name="MaxFileSize" value="10Mb"/>
    <param name="MaxBackupIndex" value="10"/>
    <param name="file" value="$LOG_DIR$/idea-teamcity-xmlrpc.log"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d [%r] %6p - %30.30c - %m \n"/>
    </layout>
</appender>

<category name="jetbrains.buildServer.XMLRPC" additivity="false">
    <priority value="DEBUG"/>
    <appender-ref ref="TC-XMLRPC-FILE"/>
</category>

<category name="jetbrains.buildServer" additivity="false">
    <priority value="DEBUG"/>
    <appender-ref ref="TC-FILE"/>
</category>

```

After changing this file, restart the IDE. The TeamCity plugin debug logs are saved into `idea-teamcity*` files and will appear in the logs directory of the [IDE settings](#) (`<IDE settings/data directory>/system/log` directory).

#### [Open in IDE Functionality Logging](#)

(Applicable to IntelliJ IDEA and Eclipse)

Add the following JVM option before starting IDE:

`-Dteamcity.activation.debug=true`

the logging related to the open in IDE functionality will appear in the IDE console.

#### No Suitable Build Configurations Found for Remote Run

First of all, check that your [VCS settings in IDEA](#) correspond to the [VCS settings](#) in TeamCity.

If they do not, change them and it should fix the problem.

Secondly, check that the build configurations you expect to be suitable with your IDEA project has either [server-side VCS checkout mode](#) or [agent-side checkout](#) and NOT manual VCS checkout mode (it is not possible to apply a personal patch for a build with the manual checkout mode because TeamCity must apply that patch after the VCS checkout is done, but it does not know or manage the time when it is performed).

If the settings are the same and you do not use the manual checkout mode but the problem is there, do the following:

- Provide us your IDEA VCS settings and TeamCity VCS settings (for the build configurations you expect to be suitable with your IDEA project)
- Enable debug logs for the TeamCity IntelliJ plugin (see [above](#))
- Enable the TeamCity server debug logs (see [above](#))
- In the [TeamCity IntelliJ plugin](#), try to start a remote run build
- Provide us debug logs from the TeamCity IntelliJ plugin and from the TeamCity server.

[Back to top](#)

#### Logging in TeamCity Eclipse plugin

To enable tracing for the [plugin](#), run Eclipse IDE with the `-debug <filename>` command line parameter. The `<filename>` portio

n of the argument should be a properties file containing key-value pairs. The name of each property corresponds to the plugin module and the value is either 'true' (to enable debug) or 'false'. Here is an example of enabling most common tracing options:

```
jetbrains.teamcity.core/debug = true
jetbrains.teamcity.core/debug/communications = false
jetbrains.teamcity.core/debug/ui = true
jetbrains.teamcity.core/debug/vcs = true
jetbrains.teamcity.core/debug/vcs/detail = true
jetbrains.teamcity.core/debug/parser = true
jetbrains.teamcity.core/debug/platform = true
jetbrains.teamcity.core/debug/teamcity = true
jetbrains.teamcity.core/performance/vcs = true
jetbrains.teamcity.core/performace/teamcity = true
```

Read more about Eclipse Debug mode [Gathering Information About Your Plug-in](#) and built-in Eclipse help.

[Back to top](#)

## TeamCity Visual Studio Addin issues

### TeamCity Addin logging

To capture logs from the TeamCity [Visual Studio Addin](#), please run Microsoft Visual Studio executable (devenv.exe) with additional command line arguments:

- For TeamCity VS Add-in as a part of [ReSharper Ultimate](#) use /ReSharper.LogFile <PATH\_TO\_FILE> and /ReSharper.LogLevel <Normal|Verbose|Trace> switches
- For Legacy version of TeamCity VS Add-in, use /TeamCity.LogFile <PATH\_TO\_FILE> and /TeamCity.LogLevel <Normal|Verbose|Trace> switches

### Visual Studio logging

To troubleshoot common Visual Studio problems please run Visual Studio executable file with [/Log command Line switch](#) and send us resulting log file.

[Back to top](#)

## dotCover Issues

To collect additional logs generated by [JetBrains dotCover](#), add the `teamcity.agent.dotCover.log` configuration parameter to the build configuration with a path to an empty directory on the agent.

All dotCover log files will be placed there and TeamCity will publish zipped logs as hidden build artifact `.teamcity/.NETCoverage/dotCoverLogs.zip`.

## JVM Crashes

On a rare occasion of the TeamCity server or agent process terminating unexpectedly with no apparent reason, it can happen that this is caused by a Java runtime crash.

If this happens, the Oracle JVM creates a file named `hs_err_pid*.log` in the working directory of the process. The working directory is usually <[TeamCity server or agent home](#)>/bin Under Windows when running as a service it can be other like `c:\Windows\SysWOW64`. You can also search the disk for the recent files with "hs\_err\_pid" in the name.

See also [Oracle documentation](#), [the Fatal Error Log section](#).

Please send this file to us for investigation and consider updating the JVM for [the server](#) (or for [agents](#)) to the latest version available.

If you get the "There is insufficient memory for the Java Runtime Environment to continue. Native memory allocation (malloc) failed to allocate ..." message with the crash or in the crash report file, make sure to [switch to 64 bits JVM](#) or reduce `-Xmx` setting not to increase 1024m, see details in the [memory configuration section](#).

## Build Log Issues

While investigating issues related to a build log, we might need the raw binary build log as stored by TeamCity. It can be downloaded via the Web UI from the Build Log build's tab: select "Verbose" log detail and use the "raw messages file" link at the top-right.

## IntelliJ IDEA Inspections

The inspections result from a TeamCity build may not match inspections from a local run in IntelliJ IDEA.

To help us investigate issues with inspections, do the following:

1. Add "system.teamcity.dont.delete.temp.result.dir=true" to the [configuration parameters](#)
2. Add "%system.teamcity.build.tempDir%/inspection\*result/\*\* => inspections-reports-data-%build.number%.zip" rule to [Artifact paths](#)
3. Add "%system.teamcity.build.tempDir%/idea-logs/\*\* => inspections-reports-idea-logs-%build.number%.zip" rule to [Artifact paths](#)
4. Add "-Didea.log.path=%system.teamcity.build.tempDir%/idea-logs/" to the runner's [JVM command line parameters](#)' field.
5. Run a new build.
6. Send us 'inspections-reports-\* .zip' files.

## Uploading Large Data Archives

Files under 10 MB in size can be attached right into the [tracker issue](#) (if you do not want the attachments to be publicly accessible, limit the attachment visibility to "teamcity-developers" user group only).

You can also send small files (up to 2 MB) via email: [teamcity-support@jetbrains.com](mailto:teamcity-support@jetbrains.com) or via [online form](#) (up to 20 MB). Please do not forget to mention your TeamCity version and environment and archive the files before attaching.

### FTP

If the file is over 10 MB, you can upload the archived files to <ftp://ftp.intellij.net/.uploads> and let us know the exact file name. If you receive the permission denied error on an upload attempt, please rename the file. It's OK that you do not see the file listing on the FTP.

The FTP accepts standard anonymous credentials: username: "anonymous", password: "<your e-mail>". In addition to usual, unencrypted connections, TLS ones are also supported.

In case of access issues, time-out errors, etc. please try using passive FTP mode.

### HTTP

You can upload a file via <https://uploads.jetbrains.com/> form and let us know the exact file name.

[Back to top](#)

## Applying Patches

### Microsoft Visual Source Safe Integration

To apply a patch for `vss-native.exe`:

1. Shut down the TeamCity server.

2. Open the <TeamCity Home>/webapps/root/WEB-INF/plugins/vss/ or <TeamCity Home>/webapps/root/WEB-INF/lib/ folder.
3. Back up the vss-support.jar file.
4. Inside the vss-support.jar file, replace the /bin/vss-native.exe with the new one.
5. Start the server.

To apply a full VSS plugin patch:

1. Shut down the TeamCity server.
2. Open <TeamCity Home>/webapps/root/WEB-INF/plugins/vss/ or <TeamCity Home>/webapps/root/WEB-INF/lib/ .
3. Back up vss-support.jar .
4. Replace vss-support.jar with the new one.
5. Start the server.

### Capturing Logs From VSS-native

Each time TeamCity starts, it creates a new instance of the vss-native.exe file and places it into the <TeamCity Home>/temp folder. The name of the copy is generated automatically and uses the following template: TC-VSS-NATIVE-<some digits>.exe

To manually enable detailed logging (for debugging purposes) for VSS Native:

1. Copy the <TeamCity Home>/temp/TC-VSS-NATIVE-<some digits>.exe file to any folder.
2. Run the program with the /log switch.

To get the commandline syntax and options reference, run the program without any switch.

## Microsoft Team Foundation Server Integration

To apply a patch for **tfs-native.exe**:

1. Shutdown the TeamCity server
2. Open <TeamCity Server>/webapps/root/WEB-INF/plugins/tfs/ or <TeamCity Server>/webapps/root/WEB-INF/lib/ .
3. Backup tfs-support.jar .
4. Inside the tfs-support.jar file, replace /bin/tfs-native.exe with a new one.
5. Start the server.

To apply a full TFS plugin patch:

1. Shutdown the TeamCity server.
2. Open <TeamCity Home>/webapps/root/WEB-INF/plugins/tfs/ or <TeamCity Home>/webapps/root/WEB-INF/lib/ .
3. Back up tfs-support.jar .
4. Replace tfs-support.jar with a new one.
5. Start the server.

### Capturing logs from TFS-native

To enable creating logs from TFS-native:

1. Locate tfs-native.exe under the TeamCity temp folder. The file name format is TC-TFS-NATIVE-<digits>.exe .
2. Create a copy of the file in any other folder.
3. Run this program with the /log switch.

To get the command-line switches help, run the process with no parameters.

Log files will be created in the <TeamCity agent>/temp/buildTmp/TeamCity.NET folder. For each process a new log file will be created.

## .NET runners

To patch .NET part of .NET runners:

1. Open <TeamCity Server>/webapps/ROOT/WEB-INF/plugins/dotNetRunners/agent .
2. Copy dotNetPlugin.zip to a temporary folder.
3. Back up dotNetPlugin.zip .
4. Extract dotNetPlugin.zip .
5. Replace the contents of the /bin folder with new files.
6. Pack the files again. Make sure there are no files in the root of the archive.
7. Create the <TeamCity Server>/webapps/ROOT/update/plugins directory.
8. Put dotNetPlugin.zip file into <TeamCity Server>/webapps/ROOT/update/plugins. All build agents will upgrade automatically

## 9. Run builds.

To enable logging from .NET runners:

1. Open <TeamCity Server>/webapps/ROOT/WEB-INF/plugins/dotNetRunners/agent.
2. Copy dotNetPlugin.zip to a temporary folder.
3. Back up dotNetPlugin.zip.
4. Extract dotNetPlugin.zip.
5. Copy /bin/teamcity-log4net-debug.xml to /bin/teamcity-log4net.xml.
6. You may patch the Log4NET config file if you need.
7. Pack the files again. Make sure there are no files in the root of the plugin archive.
8. Create the <TeamCity Server>/webapps/ROOT/update/plugins directory.
9. Put dotNetPlugin.zip file into <TeamCity Server>/webapps/ROOT/update/plugins. All build agents will upgrade automatically.
10. Run builds.

By default, all the log files will be stored in the <TeamCity agent>/temp/buildTmp/TeamCity.NET folder. Log files are created for each process separately.

## Visual C Build Issues

If you experience any problems running Visual C++ build on a build agent, you can try to workaround these issues with the following steps, sequentially:



Any of these steps may solve your issue. Please feel free to leave feedback of you experience.

- Make sure you do not use mapped network drives.
- Make sure build user have enough right to access necessary network paths
- Log on to the build agent machine under the same user as for build and try running the following command:

```
msbuild.exe <path to solution.sln> /p:Configuration:Release /t:Rebuild
```

- Build Agent service runs under the user with local administrative privileges
- Make sure Microsoft Visual Studio is installed on the build agent
- You have to start Visual Studio 2005 or Visual Studio 2008 under build user once <http://www.jetbrains.net/devnet/message/5233781#5233781>
- If Error spawning cmd.exe appears, you should put the following lines exactly into the list in Tools -> Options -> Projects and Solutions -> VC++ Directories:

```
--$(SystemRoot)\System32  
--$(SystemRoot)  
--$(SystemRoot)\System32\wbem
```

<http://www.jetbrains.net/devnet/message/5217957#5217957>

- You need to add all environment variables from ...\\Microsoft Visual Studio 9.0\\VC\\vcvarsall.bat to environment or to `buildAgent.properties` file
- Try using devenv.exe with Command Line Runner instead of Visual Studio(sln) build runner
- Ensure all paths to sources do not contain spaces
- Set VCBUILDUserEnvironment=true in runner properties
- Specify 'VCBuildAdditionalOptions' property with value '/useenv' in the build configuration settings to instruct msbuild to add '/useenv' commandline argument for spawned vcbuild processes.

See also:

[Administrator's Guide: .NET Testing Frameworks Support | NUnit support](#)

## Getting Started with NUnit

This tutorial aims at describing the basic practices of using [NUnit 3](#) in TeamCity. The test project and script samples can be found [here](#). The order of use cases is based on the number of the TeamCity features involved: the first case is the most basic,

more complex cases that follow utilize a larger number of features. We recommend that you familiarize yourself with all features, finding their advantages and disadvantages, and then decide in favor of one or another.

On this page:

- [Installing NUnit](#)
- [Case 1. Command Line](#)
- [Case 2. MSBuild](#)
- [Case 3. NUnit Build Step](#)
- [Case 4. NUnit Build Step, options](#)
- [Debugging NUnit tests](#)

## Installing NUnit

To use the NUnit with TeamCity, you need to install [NUnit NuGet package](#) on TeamCity agents first.

To do that, use one of the following options:

- you can add the NuGet install build step as the first step of your build configuration.  
For example, you can add a command line build step before the NUnit build step which will install the `NUnit.Console` NuGet package as follows:

```
%teamcity.tool.NuGet.CommandLine.DEFAULT%\tools\nuget.exe install NUnit.Console -version 3.6.0 -o packages
```

Note that `%teamcity.tool.NuGet.CommandLine.DEFAULT%` is a reference to NuGet installed under the TeamCity agent. You can install NuGet on agents from the Administration | Tools page, where you can also mark one of the installed NuGet versions as default.

After that the `%teamcity.tool.NuGet.CommandLine.DEFAULT%` parameter reference should properly resolve to the NuGet installation path on the agent.

Then `nunit3-console` should appear under the packages directory.

To run tests, in the next NUnit build step, specify the NUnit path in the NUnit settings as `packages\NUnit.ConsoleRunner.3.6.0\tools\nunit3-console.exe`.

- Another approach is to install NUnit manually on all of the agents in some standard place, and configure the path to `nunit-console.exe` in your NUnit build step.

## Case 1. Command Line

What can be simpler than the command line? Starting from version 3, NUnit supports TeamCity out-of-the-box, which allows running tests from the command line, preserving the basic features TeamCity-NUnit integration. NUnit automatically detects if it is run by TeamCity, and if so, it switches to the integration mode.

In addition to auto detection, the NUnit 3 console handles the special `--teamcity` argument, which also includes the integration with TeamCity. This argument can be used for the purposes of debugging, when you need to see which information NUnit sends to TeamCity: detailed test information, the order of tests, etc.

As mentioned earlier, using the NUnit console from the command line is the simplest way to run tests. And even in this case TeamCity will provide the following basic features of integration:

- reporting the test run information in Build Log while the tests are executing
- TeamCity will report the test results upon the tests finish
- all the [investigation](#) features will be available in TeamCity.

This is what the TeamCity build step to run tests from the command line looks like:

## Build Step (2 of 2): Tests ▾

<b>Runner type:</b>	Command Line Simple command execution
<b>Step name:</b>	Tests Optional, specify to distinguish this build step from other steps.
<b>Run:</b>	Executable with parameters
<b>Command executable:</b> *	packages\NUnit.Console.3.0.0\tools\nunit3-console.exe
<b>Command parameters:</b>	Tests.dll --where cat=Sample1

To sum up, with this simple use case you can avail yourself to the basic features of the TeamCity-NUnit 3 integration.

## Case 2. MSBuild

This use case is very similar to the previous one, but it should be mentioned that MSBuild is the standard build platform in the .Net world.

TeamCity supports collecting code coverage statistics out of the box; additional plugins can be [installed](#) to use the JetBrains [dotTrace](#) and [JetBrains dotMemory Unit](#) in TeamCity.

The [project file](#) is trivial: the [Exec](#) task, which launches tests, is run:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="14.0" DefaultTargets="RunTests"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<Target Name="RunTests">
<Exec IgnoreExitCode="True" Command="nunit3-console.exe Tests.dll">
<Output TaskParameter="ExitCode" ItemName="exitCode" />
</Exec>
<Error Text="Error while running tests" Condition="@{exitCode} < 0" />
</Target>
</Project>
```

The NUnit console returns the number of failed tests as the positive exit code and, in case of the NUnit test infrastructure failure, as the negative exit code.

TeamCity controls the test execution progress, but the NUnit infrastructure exceptions may not allow TeamCity to collect the required information. That is why the `IgnoreExitCode="True"` attribute needs to be set, which will ignore the positive exit codes and will not interrupt the build due to several failed tests. The [Error](#) task will stop the build in case of the test infrastructure errors for the negative exit codes.

## Build Step (3 of 3): Tests ▾

<b>Runner type:</b>	MSBuild
Runner for MSBuild files	
<b>Step name:</b>	Tests
Optional, specify to distinguish this build step from other steps.	
<b>Build file path:</b> *	sample2.proj
The specified path should be relative to the checkout directory.	
<b>MSBuild version:</b>	Microsoft Build Tools 2015
<b>MSBuild ToolsVersion:</b>	14.0
<b>Run platform:</b>	x86
<b>Targets:</b>	
Enter targets separated by space or semicolon.	
<b>Command line parameters:</b>	
Enter additional command line parameters to MSBuild.exe.	
<b>.NET Coverage</b>	
<b>.NET Coverage tool:</b> <small>?</small>	<No .NET Coverage>
Choose a .NET coverage tool.	
<small>⚠ Test code coverage is supported only for NUnit tests run using TeamCity facilities. <small>?</small></small>	
<b>JetBrains dotTrace</b> ↗	
<b>Run build step under JetBrains dotTrace profiler:</b>	<input type="checkbox"/>
<b>JetBrains dotMemory Unit</b> ↗	
<b>Run build step under JetBrains dotMemory Unit:</b>	<input type="checkbox"/>

Besides the project file, you can define the MSBuild version and platform, the target, you can use profiles and other settings.

To make the build more stable, you can introduce a few changes to the project file. For example, you can define the path to the NUnit console dynamically. This path may change with the updates of the NUnit console NuGet package, but our build configuration will not require any changes. The code below gets the path to the NUnit console to the pathToNUnitConsole variable:

```
<GetNUnitConsolePath  
BaseDir="$(MSBuildProjectDirectory)\packages">  
  <Output TaskParameter="PathToNUnitConsole" ItemName="pathToNUnitConsole"/>  
</GetNUnitConsolePath>
```

You can dynamically list the assemblies to be tested: for example, search for them in specific directories and filter them to match a regular expression pattern:

```

<ItemGroup>
  <Assemblies Include="Tests\bin\**\*.dll"/>
</ItemGroup>

<CreateNUnitListOfAssemblies
  Assemblies="@{Assemblies}" RegexpAssemblyFilter=".*Tests.dll$">
  <Output TaskParameter="ListOfAssemblies" ItemName="listOfAssemblies"/>
</CreateNUnitListOfAssemblies>

```

The following project files contain some examples: [sample2adv.proj](#), [unit-utils.targets](#)

## Case 3. NUnit Build Step

The [NUnit](#) build step is probably the simplest and yet the most powerful way to launch NUnit tests in TeamCity. In most cases it's sufficient to set only the 2 parameters: the path to the NUnit console runner and the list of assemblies to be tested:

### Build Step (3 of 3): Tests ▾

<b>Runner type:</b>	<input type="text" value="NUnit"/> <small>NUnit tests runner</small>
<b>Step name:</b>	<input type="text" value="Tests"/> <small>Optional, specify to distinguish this build step from other steps.</small>
<b>NUnit runner:</b> <small>②</small>	<input type="text" value="NUnit 3.0.0"/>
<b>Path to NUnit console runner:</b> <small>* ②</small>	<input type="text" value="packages\NUnit.Console.3.0.0\tools\nunit3-console.exe"/> <small>Specify the path to NUnit console runner including the file name</small>
<b>.NET Runtime:</b>	Platform: <input type="text" value="auto (MSIL)"/> Version: <input type="text" value="auto"/>
<b>Run tests from:</b>	Edit assembly files include list: <input type="text" value="Tests/bin/Debug/Tests.dll"/> <small>Enter comma- or newline-separated paths to assembly files relative to checkout directory. <small>↑</small></small>
<b>.NET Coverage</b>	
<b>.NET Coverage tool:</b> <small>②</small>	<input type="text" value="&lt;No .NET Coverage&gt;"/> <small>Choose a .NET coverage tool.</small>
JetBrains dotTrace <input checked="" type="checkbox"/>	
Run build step under dotTrace profiler: <input type="checkbox"/>	
JetBrains dotMemory Unit <input checked="" type="checkbox"/>	
Run build step under JetBrains dotMemory Unit: <input type="checkbox"/>	

Look at the NUnit runner field which defines the NUnit version used to run tests. When configuring your build step for NUnit 3, the Path to NUnit console runner field is required to contain the path to the NUnit console: prior to TeamCity 9.1.4 specify the directory containing the console executable file, in the later TeamCity versions specify the path to the console executable file including the file name.

In all the examples the NuGet package manager provides the NUnit infrastructure. Using NuGet enables the user to conveniently manage the test environment, update NUnit, run test locally as they will be run by TeamCity.

The NUnit build step is a is straightforward and user-friendly way to run NUnit tests in TeamCity. It provides the maximum range of options simultaneously concealing the specifics of running tests on different OS's by using [Mono](#).

## Case 4. NUnit Build Step, options

The NUnit build step requires the Path to NUnit console runner to be set when configuring the step for NUnit 3 - this is a mandatory field.

The other fields provide a lot of useful options, and this section discusses some of them.

### Build Step (4 of 4): Tests |

<b>Runner type:</b>	<input type="text" value="NUnit"/>
NUnit tests runner	
<b>Step name:</b>	<input type="text" value="Tests"/>
Optional, specify to distinguish this build step from other steps.	
<b>Execute step:</b>	<input type="text" value="If all previous steps finished successfully"/>
Specify the step execution policy.	
<b>NUnit runner:</b>	<input type="text" value="NUnit 3.0.0"/>
<b>Path to NUnit console runner:</b> *	<input type="text" value="%NUnit3ConsolePath%"/>
Specify the path to NUnit console runner including the file name	
<b>Path to application configuration file:</b>	<input type="text" value="Tests\bin\Debug\Tests.dll.config"/>
Specify the path to the application configuration file to be used by NUnit tests	
<b>Additional command line parameters:</b>	<input type="text" value="--agents=1"/>
Enter additional command line parameters to the NUnit console runner	
<b>.NET Runtime:</b>	<b>Platform:</b> <input type="text" value="auto (MSIL)"/>
	<b>Version:</b> <input type="text" value="auto"/>
<b>Run tests from:</b>	<b>Edit assembly files include list:</b> <input type="text" value="Tests/bin/Debug/Tests.dll"/> Enter comma- or newline-separated paths to assembly files relative to checkout directory.

One of the options is defining the application configuration file. Sometimes tests obtain data from a configuration file and to facilitate this, you need to define the path to the application configuration file to be used when running tests in the Path to application configuration file field. The path can be absolute or relative to the [build checkout directory](#). Unfortunately, NUnit is limited by allowing only one configuration file per build step. Due to this limitation, if you need to test several assemblies with different configurations in one build step, you'll have to aggregate several application configuration files into a common configuration file. If it is not possible, the test launch can be split into several steps and define a configuration file in each of the steps.

The NUnit 3 console has a great number of settings defined by command line arguments. The [Additional command line parameters](#) field allows setting many of the command line parameters for the NUnit console, but there are some limitations:

- The `--where` command line argument is used as the priority test filtering setting. When it is used simultaneously with the NUnit categories include or/ and NUnit categories exclude fields to filter tests by category, TeamCity will ignore these options: a warning will be displayed and only the `--where` argument will be used.

- Using the parameters below in the Additional command line parameters field may cause your build to fail:
  - the list of assemblies to be tested, as the NUnit build step determines the list of assemblies from the NUnit project file due to the command line size limitations
  - NUnit project file, as TeamCity creates its own temporary NUnit project file (see [below](#))
  - work: the NUnit build step uses the [build checkout directory](#) as the base directory
  - noheader is used by default
  - x86 if the .NET Runtime - Platform field is set to "x86"
  - framework if the .NET Runtime - Version is set to something other than "auto"
  - explore if the algorithm below is used

The NUnit build step can be set to first run the tests which failed in the previous builds. The idea is that it saves time to conclude about a probable state of the finished build. If the Reduce test failure feedback time flag is set, the test are run in three steps provided that the previous build has failed tests:

- As the first step, TeamCity obtains all the list of tests for assemblies taking into account filters by category (the [--explore](#) console argument is used), the statistics of the previous step run is analysed and 2 test lists are created: the priority list and the rest. The priority list includes the tests which failed in the previous build.



When the "Reduce test failure feedback time option" is set, the [Explicit](#) attribute will not work.

- During this step the priority tests are run.
- The remaining tests follow.

Another great feature of the NUnit build step is the fact that it executes tests on different OS's the same way, without changing the configuration. This is true if the agents, where the full-blown .Net is unavailable, have [Mono](#). It allows the build configurations to be OS-independent.

## Debugging NUnit tests

One of the priority tasks for NUnit support in TeamCity was the possibility for the user to easily reproduce in the command line all the actions performed by TeamCity to run tests. This allows for quick and effective resolution of the potential issues with configuring a build.

TeamCity records all the data related to running tests into the [Build Log](#). Thus the commands which TeamCity uses to run tests can be copied from the build log and can be run from the command line locally or on agents.

When running tests, besides commands, TeamCity creates temporary files: NUnit project files; JetBrains dotCover, JetBrains dotTrace or JetBrains dotMemory Unit configuration files. These are contained in the [hidden build artifacts](#). For example, the command line launching tests for case \$ will look as follows in the build log:

```
...\\nunit3-console.exe ...\\50qkbf9J2qJNkUK4KEtKvxs8TFFnlrno.nunit
```

In this case [50qkbf9J2qJNkUK4KEtKvxs8TFFnlrno.nunit](#) is the NUnit project file, which can be downloaded from [hidden build artifacts](#):

The screenshot shows the TeamCity interface with the 'Artifacts' tab selected. The 'teamcity' folder contains several sub-folders: 'logs', 'nuget', 'properties', and 'settings'. Below these is a file named '50qkbf9J2qJNkUK4KEtKvxs8TFFnlrno.nunit' with a size of '(230 B)'. This file is highlighted with a red border. At the bottom of the artifacts list, there is a message: 'Total size: 30.32 KB' and 'Hidden artifacts from the .teamcity directory are displayed. [Hide](#)'.

See also:

[Administrator's Guide: NUnit Support](#)

# Getting started with PHP

In this tutorial:

- Introduction
- Setting up the project
- Adding build steps
  - Unit tests
  - Running Phing
  - Code coverage
- Exploring build results

## Introduction

TeamCity supports your Continuous Integration (CI) process in many technologies. In this tutorial, we'll configure a Continuous Integration (CI) process for a PHP project. We will be using the open-source PHP project [PHPExcel](#) as a sample project we want to provide CI for. This project features a large amount of code, PHPUnit tests and uses Phing to create build artifacts. Using TeamCity, we will automate the build process and make it ready for immediate feedback once the source code on GitHub changes.

This tutorial assumes you already have a PHP environment with PEAR, PHPUnit and Phing installed. If not, now is the time. You can find more info on configuring your PHP environment [through this blog post](#).

## Setting up the project

We'll start by creating a new project and build configuration in TeamCity. The build number for this new build configuration will be "1.7.6.{0}" since PHPExcel is currently in the 1.7.6.x version range.

PHPExcel comes with a build script that creates build artifacts under the build/release/\* folder, which means we can already add that path as the artifact path TeamCity monitors.

 <http://localhost:8080/admin/createBuildConfiguration>    Create Build Configuration ...

 [Log in to TeamCity -- Tea...](#)

 [Projects](#) |  [My Changes](#) |  [Agents](#) 1 |  [Build Queue](#) 0

Administration > PHPExcel - CI Project > Create Build Configuration

## General Settings

**Name:** \*

**Description:**

**Build number format:**   

Format may include '{0}' as a placeholder for build counter value, for example 1.{0}. It may reference to any available parameter, for example, VCS revision number: %build.vcs.number%. Note: maximum length of a build number after all substitutions is 256 characters.

**Build counter:** \*  [Reset counter](#)

**Artifact paths:**  [Edit artifact paths:](#)

New line or comma separated paths to build artifacts. Ant-style wildcards like dir/\*\*/\*, directories like \*.zip => winFiles, unix/distro.tgz => linuxFiles, where linuxFiles are target directories are supported.

**Build options:**

enable hanging builds detection  
 enable status widget 

Limit the number of simultaneously running builds (0 — unlimited)

[VCS settings >](#)

The PHPExcel project has a GitHub repository at <https://github.com/maartenba/PHPExcel.git>, a URL which we can configure in the Version Control System (VCS) settings.

← → TC http://localhost:8080/admin/editVcsf ⚙ ⚡ TC New VCS Root -- TeamCity ×

Log in to TeamCity -- Tea...

TC Projects My Changes Agents 1 Build Queue 0

Administration > Create Build Configuration > New VCS Root

Type of VCS

Type of VCS: Git

VCS Root Name

VCS Root Name: maartenba/PHPExcel - develop

Enter a unique name to distinguish this VCS root from other roots. If not specified, the name will be generated automatically.

General Settings

Fetch URL: \* https://github.com/maartenba/PHPExcel.git

It is used for fetching data from repository.

Push URL:

It is used for pushing tags to the remote repository. If blank, the fetch url is used.

Default Branch: \* develop

Branch to be used if no branch from Branch Specification is set

Branch Specification:  Edit branch specification:  
[Empty text area with scroll bars]

Branches to monitor in addition to default one. Newline-delimited set of rules in the form of  
+|-branch name (with optional \* placeholder) 

## Adding build steps

A build configuration consists of several build steps which perform the actions we desire during build.

### Unit tests

Since PHP is an interpreted language, we don't need a compilation step and can immediately start with unit tests: we want to make sure all tests are green every time source code is changed. Whenever there is a failing unit test, we want to fail the entire build and not provide any build artifacts. TeamCity comes with a number of predefined build steps for Java and .NET, but

since we're on PHP we'll have to select the Command Line build runner.

The screenshot shows the TeamCity interface for creating a build configuration. The top navigation bar includes links for 'Log in to TeamCity', 'Projects', 'My Changes', 'Agents', and 'Build Queue'. The current page is 'Administration > PHPExcel - CI Project > Create Build Configuration'. The main section is titled 'New Build Step'.

**Runner type:** Command Line (Simple command execution)

**Step name:** Run tests with coverage (You can specify a build step name to distinguish it from other steps.)

**Execute step:** Only if all previous steps were successful (You can specify step execution policy)

**Working directory:** (Optional, set if differs from the checkout directory)

**Run:** Custom script

**Custom script:** `php c:\teamcity-php\PHPUnit-TC.php -c %teamcity.build.checkoutDir%\unitTests\phpunit.xml`

Platform-specific script which will be executed as a .cmd file on Windows or as a shell script in Unix-like environments.

Buttons at the bottom right include '<< VCS settings' and 'Save'.

We want to run the PHPUnit configuration that's provided with PHPExcel source code. Invoking this can be done using the following command line script:

```
phpunit \c phpunit.xml
```

By default, TeamCity will import the test results provided by PHPUnit. However we can also report real-time test results to

TeamCity, so that we can already see results during a build run before it's even finished. Using a wrapper around PHPUnit which uses [service messages](#) to report build results. Locate the wrapper somewhere on the build agent or have the build agent download it from the above GitHub repository directly using a second VCS root.

The screenshot shows the TeamCity interface for the PHPExcel - CI project. The main navigation bar includes 'Projects' (selected), 'My Changes', 'Agents 1', 'Build Queue 0', 'Overview', 'Changes 0' (selected), 'Tests', 'Build Log', 'Build Parameters', and 'Artifacts'. The build configuration is 'Main' with build number '#1.7.6.2 (07 Jan 13 08:59)'. The build status is 'Stop' (indicated by a red asterisk). The 'Changes' tab shows 66 failed tests, grouped by package/suite. The 'Tests' section lists the failed tests under 'PHPExcel Unit Test Suite: DateTimeTest: DateTimeTest::testDATEDIF' (9 failures) and 'PHPExcel Unit Test Suite: DateTimeTest: DateTimeTest::testDATEVALUE' (7 failures). Each failure is associated with a specific data set number.

Started: 07 Jan 13 08:59 Stop

Time left: N/A (passed: 14s)

Investigation: Start investigation... of current problems in this build configuration (Main)

Thread dump: View thread dump

Running step: PHPExcel Unit Test Suite: EngineeringTest: EngineeringTest::testBESSELK: testBESSELK \pear\PHPUnit\Framework\Assert.php(2134): PHPUnit\_Framework\_Co...

66 tests failed (66 new)

Group by: package/suite

- All tests
- PHPExcel Unit Test Suite: DateTimeTest: DateTimeTest::testDATEDIF (9)
  - testDATEDIF with data set #2
  - testDATEDIF with data set #52
  - testDATEDIF with data set #89
  - testDATEDIF with data set #90
  - testDATEDIF with data set #91
  - testDATEDIF with data set #92
  - testDATEDIF with data set #93
  - testDATEDIF with data set #94
  - testDATEDIF with data set #95
- PHPExcel Unit Test Suite: DateTimeTest: DateTimeTest::testDATEVALUE (7)
  - testDATEVALUE with data set #24
  - testDATEVALUE with data set #30

We can already invoke our build configuration and should be getting unit test results displayed. But we're not finished yet, we want to add some more build steps.

Running Phing

Our next build step will be invoking [Phing](#), a PHP project build system or build tool based on Apache Ant. PHPExcel comes with

a Phing build script which we'll invoke after all unit tests have passed. Let's add a new Command Line build step which uses Phing's command-line tool and pass some parameters to it:

```
phing \-f build\build.xml \-DpackageVersion=%system.build.number% \-DreleaseDate=CIbuild  
\-DdocumentFormat=doc release-standard
```

PHPExcel has four build targets defined (release-standard, release-documentation, release-pear and release-phar), all providing different build artifacts. The release-standard target which we've now specified at the command line is the default build for PHPExcel which generates a ZIP file containing all source code and phpDocumentor output.

We are also passing Phing the current build number from TeamCity using Phing's -D command line switches. The build script can use these to create the correct file names.

If you haven't configured the artifact paths while creating our build project, it's best to do so now (see [Setting up the project](#)). We want to make sure that the ZIP file generated in this build script is available from TeamCity's web interface.

When we run another build, we'll now see that unit tests are being run and afterwards the Phing build script is being run. Once the entire build is completed, we can find the ZIP file generated by the Phing build script as a downloadable build artifact.

The screenshot shows the TeamCity interface for a build of the PHPExcel project. The top navigation bar includes links for 'Projects' (selected), 'My Changes', 'Agents 1', 'Build Queue 0', and 'Log in to TeamCity'. The main title is 'PHPExcel - CI :: Main > #1.7...'. Below the title, the build number '#1.7.6.4 (07 Jan 13 09:07)' is displayed with a red error icon. A horizontal menu bar offers tabs for 'Overview', 'Changes 0', 'Tests', 'Build Log', 'Build Parameters', 'Artifacts' (selected), and 'Code Coverage'. Under the 'Artifacts' tab, two files are listed: 'coverage.zip (2.41Mb)' and 'PHPExcel\_1.7.6.4\_doc.zip (5.79Mb)'. A note indicates a total size of '8.21Mb'. There is also a link to 'Show hidden artifacts'. At the bottom, there are links for 'Help' and 'Feedback', and the text 'TeamCity Professional 7.1.3 (build 24266)'.

## Code coverage

The first build step we created was running unit tests using PHPUnit. The nice thing about PHPUnit is that it can provide code coverage information as well, nicely formatted as an HTML report. TeamCity can display HTML reports on a custom tab in the build results.

Let's first make sure code coverage is enabled. Edit the first build step (running PHPUnit) and make sure it uses PHPExcel's phpunit-cc.xml configuration file for configuring PHPUnit. This configuration file which is specific to PHPExcel outputs its code coverage report in the unitTests/codeCoverage folder. While it is possible to add all generated files to TeamCity as a build artifact, it's cleaner to ZIP that entire folder and make it available as one single file. We can have TeamCity create this ZIP file

for us by using a special artifact path pattern! Edit the build artifacts again and make sure the following two artifact paths are specified:

```
build/release/*%system.build.number%*  
unitTests/codeCoverage => coverage.zip
```

We can let TeamCity create a ZIP archive from the unitTests/codeCoverage path by simply using => and specifying a target artifact name.

Run the build again. Once it completes, the Artifacts tab should contain the coverage.zip file we've just created. Next to that, there should now be an additional tab Code Coverage available which displays code coverage results. Since we're using a TeamCity convention for reporting code coverage, namely creating a coverage.zip build artifact, TeamCity will automatically display the coverage results in a new report tab.

← → TC http://localhost:8080/viewLog.html?l ↗ ⌂ TC PHPExcel - CI :: Main > #1.7... ×

Log in to TeamCity -- Tea...

TC Projects My Changes Agents 1 Build Queue 0

PHPExcel - CI > Main > ! #1.7.6.4 (07 Jan 13 09:07)

Overview Changes 0 Tests Build Log Build Parameters Artifacts Code Coverage

C:\TeamCity\buildAgent\work\271472a647ed9c6c\Classes (Dashboard)

## Class Coverage Distribution

Coverage Range (%)	Count
0%	~105
0-10%	~25
10-20%	~5
20-30%	~5
30-40%	~5
40-50%	~5
50-60%	~5
60-70%	~5
70-80%	~5
80-90%	~5
90-100%	~5
100%	~5

## Class Cor

X	Y
1400	1400
750	750
500	500
100	100
50	50
20	20
10	10
5	5
2	2
1	1
0	0

## Top Project Risks

- PHPExcel\_Reader\_Excel5 (1614170)
- PHPExcel\_Calculation\_Statistical (486506)

## Least Tes

- PHPExcel\_Worksh
- PHPExcel\_Worksh

Help Feedback TeamCity Professional 7.1.3 (build 24266)

The screenshot shows the TeamCity build log interface for the PHPEXCEL-CI project. The build number is #1.7.6.4 (07 Jan 13 09:07). The code editor displays two functions:

```
54     public static function _isLeapYear($year) {  
55         return (((($year % 4) == 0) && ((($year % 100) != 0) || ((($year  
56     } //     function _isLeapYear()  
57  
58  
59     private static function _dateDiff360($startDay, $startMonth, $sta  
60         if ($startDay == 31) {  
61             --$startDay;  
62         } elseif ($methodUS && ($startMonth == 2 && ($startDay == 29  
63             $startDay = 30;  
64         }  
65         if ($endDay == 31) {  
66             if ($methodUS && $startDay != 30) {  
67                 $endDay = 1;  
68                 if ($endMonth == 12) {
```

It's possible to add additional build reporting and display results from PHP mess detector, [PHPLint](#) or even have a tab available which displays [phpDocumentor](#) contents by creating custom report tabs based on information from build artifacts.

## Exploring build results

Using TeamCity, we can view build history, VCS history, commits and so on. We have general build statistics such as success rate, build duration and test count in a graphical format.

When working with an environment based on the IntelliJ Platform, like PhpStorm or WebStorm, it's easy to link TeamCity with the IDE. After [installing the TeamCity plugin into your IDE](#) you'll notice that there are some useful little things like opening a unit test in the IDE from within TeamCity:

The screenshot shows the TeamCity 'Tests' tab with a table of test results. A context menu is open over a row for a test named 'testCurrency with data'. The menu items are: 'Test Details', 'Investigate / Mute...', 'Open in IDE' (which is highlighted with a red box), and 'Show in Build Log'. The table columns are 'Status' and 'Test'.

Status	Test
Failure	testXIRR with data set #2 (PHPExcel Unit Test Suite)
Failure	testXIRR with data s (PHPExcel Unit Test Suite)
OK	testToString (PHPExcel Unit Test Suite)
OK	testCurrency with data (AdvancedValueBind)
...	

As we've seen in this tutorial, it's very straightforward to run builds for PHP and provide Continuous Integration for your PHP projects!

Happy building!

## Continuous Delivery to Windows Azure Web Sites (or IIS)

In this tutorial, we'll go over the basics of these and see how we can deploy an ASP.NET MVC project to IIS or Windows Azure Web Sites from our TeamCity server using WebDeploy.

Deploying ASP.NET applications can be done in a multitude of ways. Some build the application on a workstation and then xcopy it over to the target server. Some use a build server, download the artifacts, change the configuration files and xcopy those over to the server. The issue with that arises when something bad creeps in: deployments become unpredictable.

What if there are leftovers of unnecessary or old assemblies on that workstation we're xcopying from? What if we forget to change the database connection string in Web.config and mess up that release? How do we quickly roll back if that happens? The .NET stack has a solution to this: Configuration Transforms and WebDeploy.

- Configuration Transforms
- WebDeploy
  - Manually creating a deployment package
- Step 1: Configuring deployment packages / WebDeploy with Visual Studio
- Step 2: Setting up the continuous integration build on TeamCity
- Step 3: Setting up the deployment on TeamCity
- Step 4: Promoting CI builds
- Conclusion

### Configuration Transforms

One of the things that typically have to happen during deployment is making changes to the configuration. Changing the database connection string, changing ASP.NET settings to no longer show us YSOD's and so on. Don't hard-code these things or write a big if-else statement based on the server's hostname to figure out the configuration. Instead, use something like configuration transforms.



Configuration transforms are files that describe "transformations" to Web.config, based on the build configuration being used. Building the Release configuration? Then Web.config will be updated with the rules described in Web.Release.config. Let's remove the debug attribute from our configuration when doing a Release build:

```

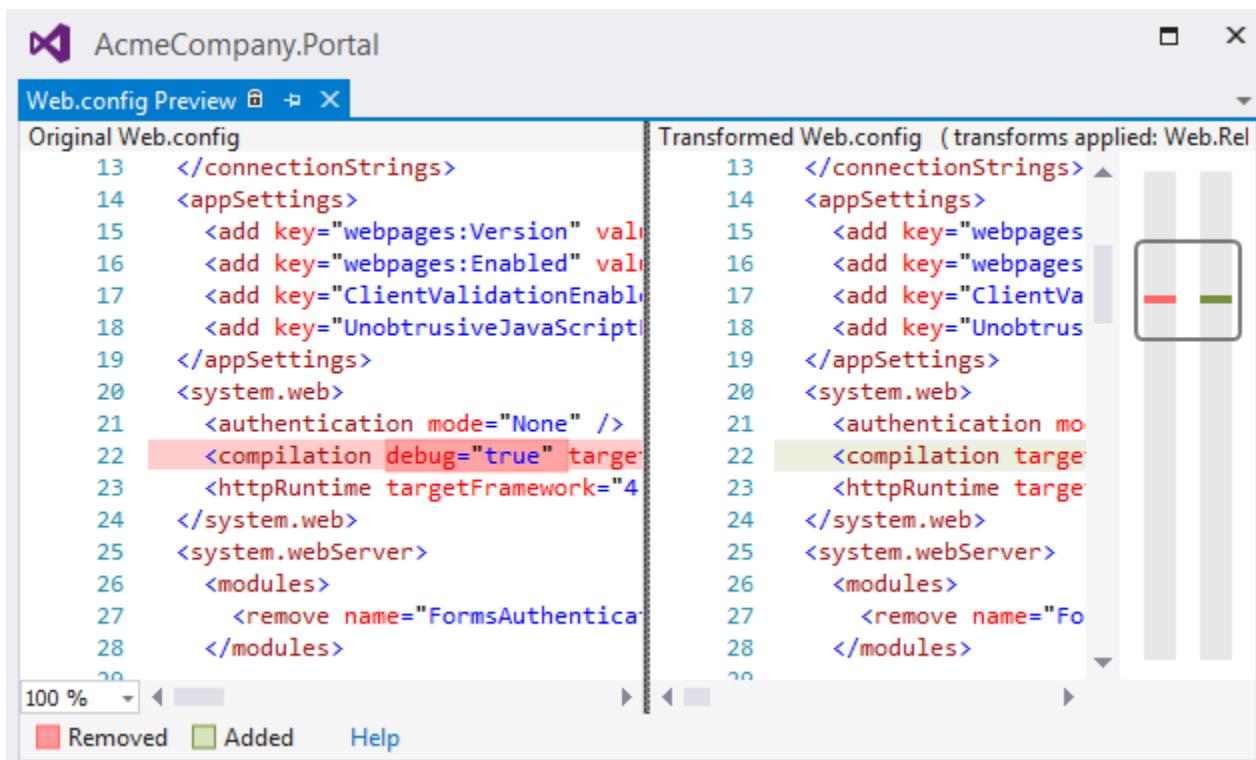
<?xml version="1.0"?>
<configuration xmlns:xdt="http://schemas.microsoft.com/XML-Document-Transform">
  <system.web>
    <compilation xdt:Transform="RemoveAttributes(debug)" />
  </system.web>
</configuration>

```

A typical ASP.NET application created in Visual Studio will contain transforms for Debug and Release builds, but they can be added by creating a new build configuration (through the Build | Configuration Manager... menu) and then using the context menu Add Config Transform.

For this tutorial, I've created 2 new configurations: Development and Production, and generated 2 new configuration transforms as well (Web.Development.config and Web.Production.config).

To test the config transform, we can make use of the context menu Preview Transform, which will show us exactly what the resulting configuration file is going to look like. The following is the result of running the Web.Release.config transform:



We can use this to virtually change or add any setting we'd like to change. Connection strings, file paths, app settings, diagnostics configuration and so on. Here's some more [documentation on what you can do with config transforms](#).

## WebDeploy

For several versions, Visual Studio has had the option to create so-called "web packages" for any ASP.NET application, containing all files required to run the app. Pages, images, CSS, JavaScript and the application binaries can be exported in such package. It's even possible to include databases and IIS settings!

These deployment packages can be used together with WebDeploy, a tool which can upload the package to a server using various protocols and can apply the config transforms we've talked about earlier.

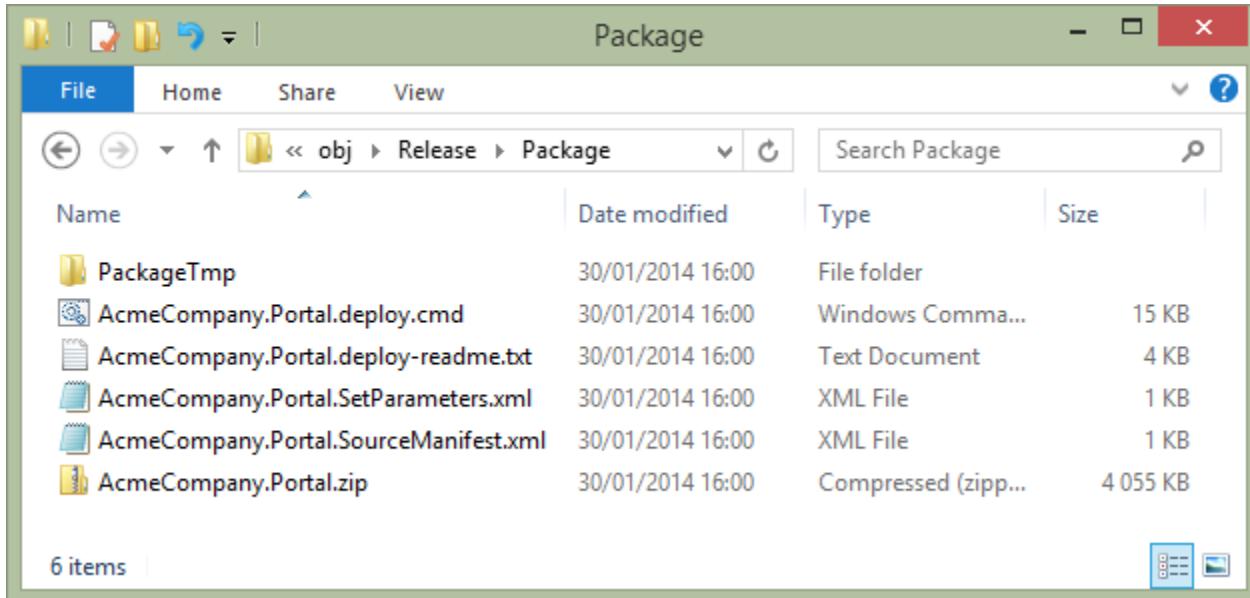
But before we deploy, let's first see how we can create a deployment package. And just so we learn about the package format, let's first do this manually by invoking msbuild.

### Manually creating a deployment package

Deployment packages can be created by running the Package build target on the project, which can easily be done using msbuild:

```
msbuild AcmeCompany.Portal.csproj /T:Package /P:Configuration=Release
```

The project will be compiled and a new folder created, containing our deployment package. And more!



The ZIP file contains our application, the other files are supporting files for deploying to a target machine. An interesting file is `AcmeCompany.Portal.SetParameters.xml`. It contains the result of our config transforms, but allows for overriding these values. Why? Well, the person building the deployment package may not know the connection string. Imagine only an administrator knows? That person can override the setting with the correct, final connection string for production through this file.

The `AcmeCompany.Portal.deploy.cmd` batch file can be run to deploy to a target environment, but... how does that work?

WebDeploy can make use of several methods to transfer the deployment package to a remote server and update configuration. It can be done using WebDeploy (an HTTPS based protocol), FTP or using a File Share. For the first option, some [additional tools should be enabled on the target IIS server](#). With good reason: the WebDeploy server-side tool will do real synchronization between sites and delete redundant content from the server. For FTP or a file share, no additional tools are required.

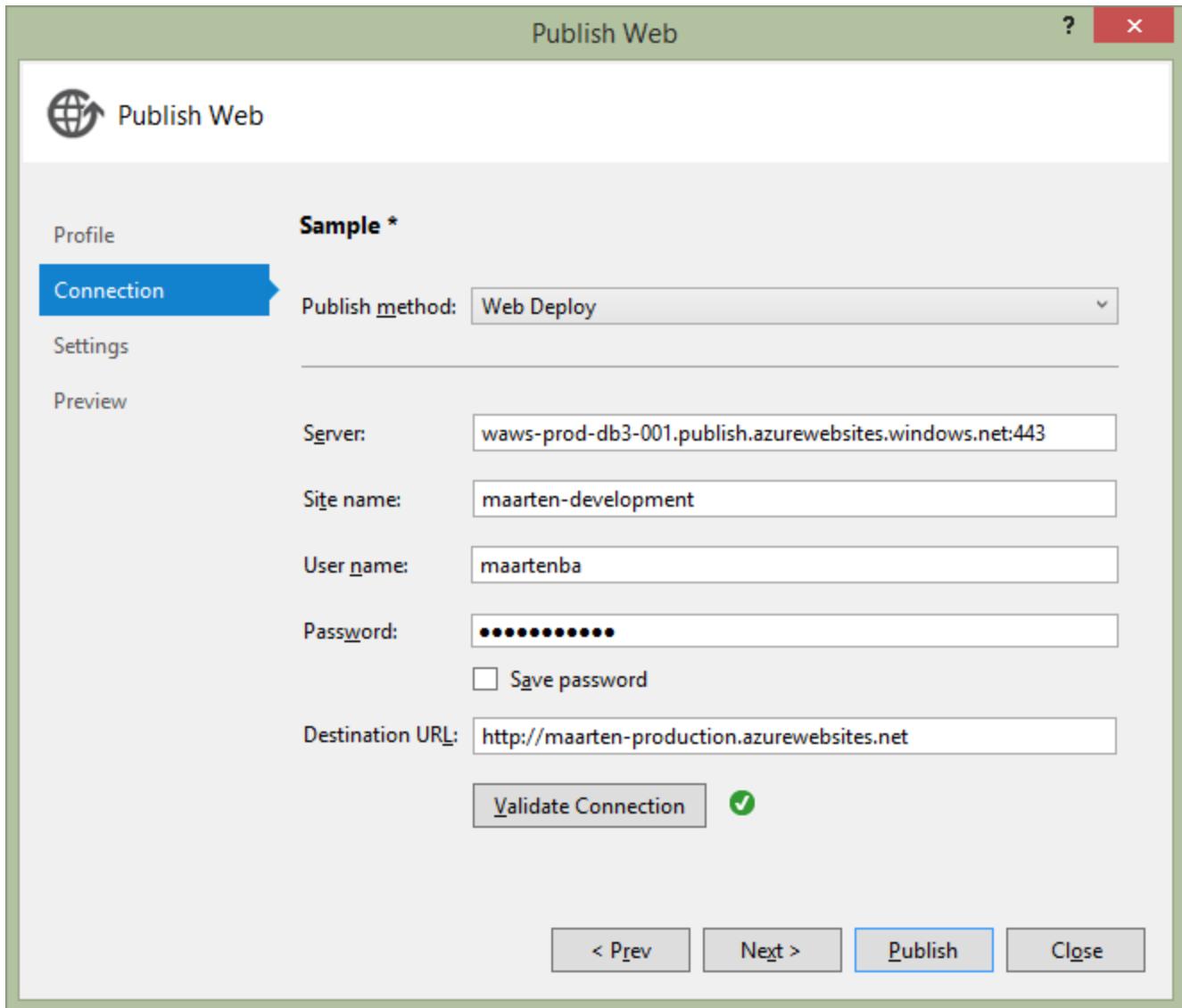
For the remainder of this tutorial, we will be covering deployment to Windows Azure Web Sites using WebDeploy, which is identical to how it works on IIS.

## Step 1: Configuring deployment packages / WebDeploy with Visual Studio

In the previous step, we've created a deployment package manually and we would also have to invoke WebDeploy manually. There is an easier way though: configuring deployment packages and WebDeploy in one go, from Visual Studio.

From the web application that should be deployed, use the context menu on the project node and click Publish. This will open up a dialog where we can do some configuration related to our deployment. We can even create multiple deployment profiles, for example one for staging and one for production.

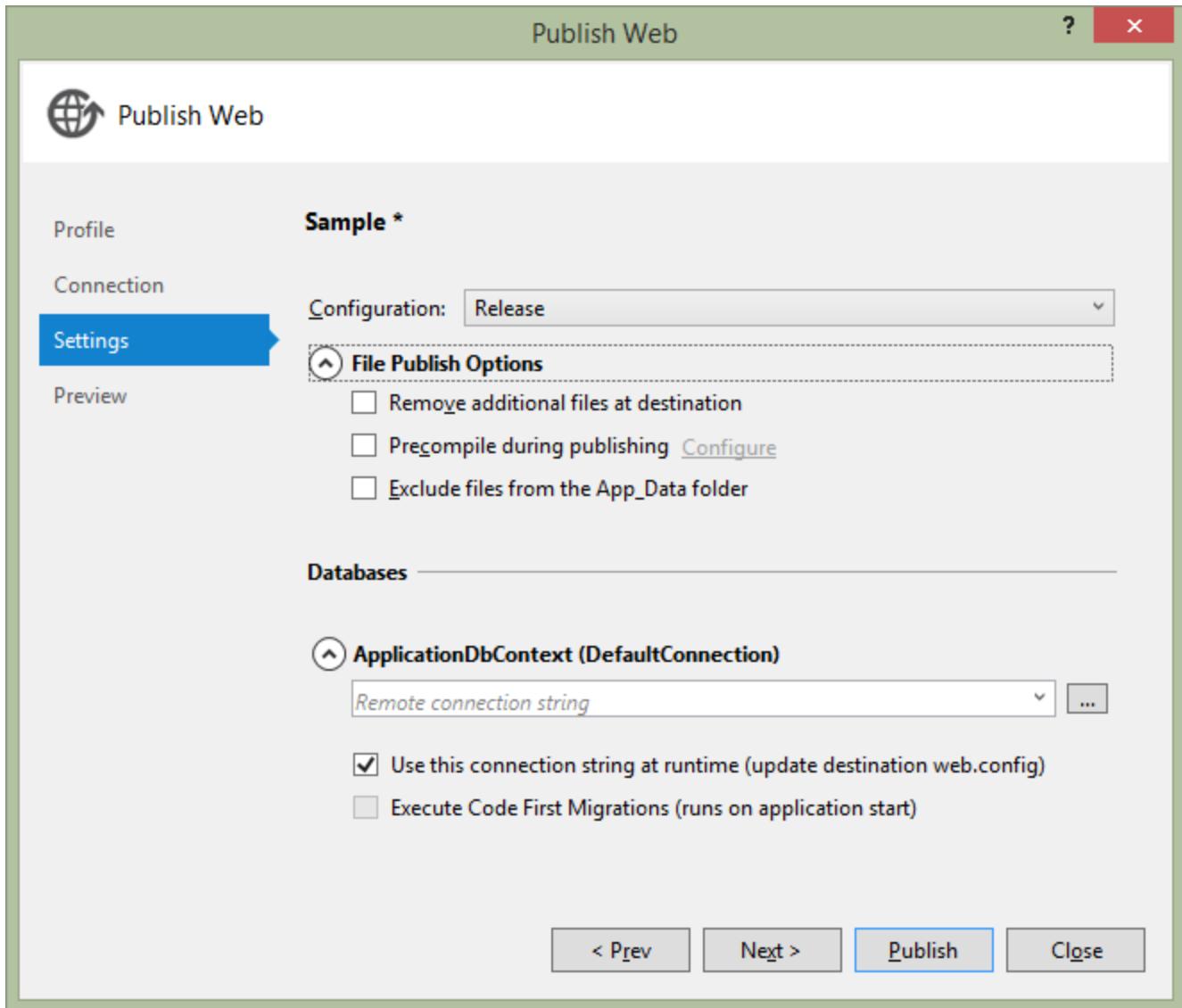
In the first step, we have to specify destination server details. This would typically be the HTTPS endpoint to the WebDeploy host (or FTP or file share details if that option was selected). After providing all details, we can validate the connection to see if it works.



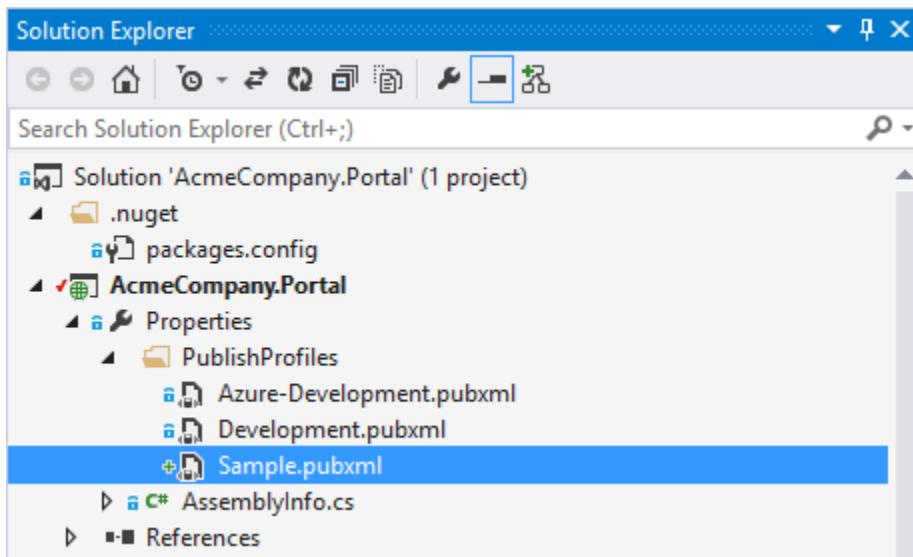
Note that instead of going through the entire wizard, Windows Azure Web Sites tooling allows importing the publish profile from the Windows Azure Management Portal. I'm showing the entire process here for when deploying to IIS.

Does a password have to be specified? No! In case the developer doesn't know it, credentials can be left blank; we'll provide the username and password later on when deploying from TeamCity.

In the next step, we can specify some deployment specifics: should files that are not in the deployment package be deleted from the target server? Should the application be precompiled? Should the database connection string be overridden? And when using Entity Framework Code First: should migrations be executed?



We can close the wizard after this step, and save the publish settings just created into a file in our project:



This is just an XML file and we can edit it if needed. And actually we should, to make our life easier later on. Open the XML file and find the <DesktopBuildPackageLocation> element. When running the WebDeploy packaging step from the command line (which TeamCity will effectively do), this location will not be found. To resolve this, change the element value and prefix the

path with `$(SolutionDir)`. Here's an example of what this element could look like:

```
<DesktopBuildPackageLocation>$(SolutionDir)\artifacts\webdeploy\Development\AcmeCompany.Portal.zip</DesktopBuildPackageLocation>
```

Save the file and make sure it is added to source control so we can make use of it when running the deployment on TeamCity.

## Step 2: Setting up the continuous integration build on TeamCity

We want to have a continuous integration (CI) build for our project, which we can trigger on every VCS check-in. This CI build will provide us with immediate feedback on the project's build status and health.

TeamCity 8.1 allows us to create a project based on a VCS URL. We can simply enter the URL to a git, Mercurial, Subversion, ... repository:

Administration >  <Root project> > Create Project From URL

**Parent Project:** \*

**Repository URL:** \*   
AVCS repository URL. Supported formats: `http(s)://`, `svn://`, `ssh://git@`, `git://`, etc. as well as URLs in Maven format. [?](#)

**Username:**   
Optional. Provide username if access to repository requires authentication.

**Password:**   
Optional. Provide password if access to repository requires authentication.

**Proceed** **Cancel**

This repository will be analyzed and scanned for build steps. In our case, TeamCity discovered a Visual Studio 2013 build step which we can immediately add to our build configuration:

	Build Step	Parameters Description
<a href="#">Use this</a>	Visual Studio (sln)	<code>Build file path: AcmeCompany.Portal.sln</code> <code>Targets: Rebuild</code> <code>Configuration: Release</code> <code>Platform: &lt;default&gt;</code>

Adding the suggested build step will result in a working build if we run it. We can specify artifact paths, version number and so on. One thing is missing though! The WebDeploy deployment package is nowhere to be seen. The reason for this is we are building the Rebuild target, which simply rebuilds our project without packaging. To solve this, we can add some additional command line parameters to our build step:

### Command line parameters:

```
/p:DeployOnBuild=True  
/p:PublishProfile="Development"  
/p:ProfileTransformWebConfigEnabled=False
```

Enter additional command line parameters to MSBuild.exe.

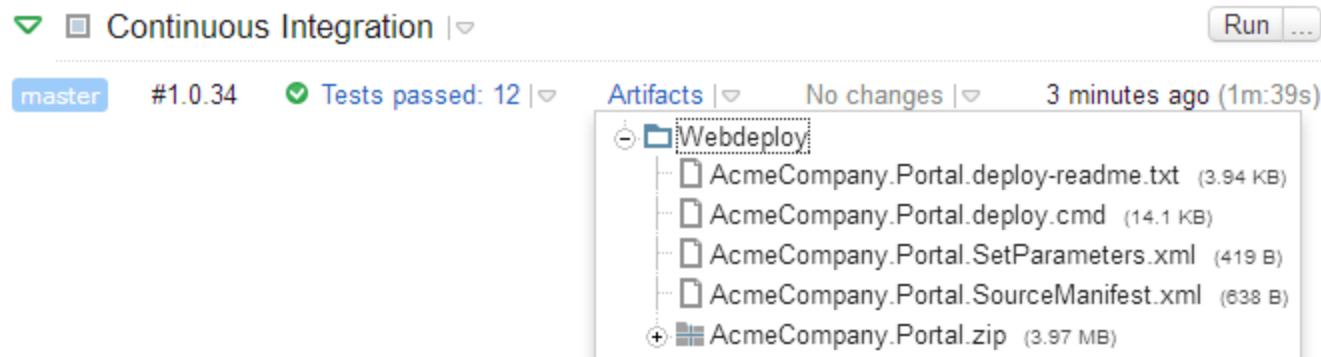
 Hide advanced options

Here's what these parameters do:

- /p:DeployOnBuild=True - triggers WebDeploy packaging
- /p:PublishProfile="Development" - specifies the deployment profile to use when packaging
- /p:ProfileTransformWebConfigEnabled=False - let's discuss this one in detail!

 Standard configuration transformations are run in an early stage, but WebDeploy runs another transformation using the <LastUsedBuildConfiguration> setting from our publish profile. This causes earlier configuration transformations to be overwritten, which we don't want to happen. Disabling the ProfileTransformWebConfigEnabled parameter avoids running this additional configuration transformation.

If we now run the build again (having specified artifacts\webdeploy\Development => Webdeploy as the artifact path, which is the path we configured in the publish profile earlier on), we will see a familiar set of files published as artifacts:



The screenshot shows the TeamCity interface for a build named "Continuous Integration". The build status is "master" and the build number is "#1.0.34". It shows "Tests passed: 12". The "Artifacts" section is expanded, showing a folder named "Webdeploy" containing several files: "AcmeCompany.Portal.deploy-readme.txt" (3.94 KB), "AcmeCompany.Portal.deploy.cmd" (14.1 KB), "AcmeCompany.Portal.SetParameters.xml" (419 B), "AcmeCompany.Portal.SourceManifest.xml" (638 B), and "AcmeCompany.Portal.zip" (3.97 MB). A "Run ..." button is visible in the top right corner of the artifacts panel.

Now let's see if we can set up the actual deployment as well!

### Step 3: Setting up the deployment on TeamCity

The strategy we'll be using for our deployments is described in the [How To....](#) We will be creating a new build configuration for every target environment we want to deploy to. These new build configurations will:

- Run the build
- Perform the deployment

What we want to achieve is this nice waterfall, where we can promote our build from CI to development to staging to production, or whichever environments we have in between CI and production.



From the TeamCity Administration, copy the CI build configuration and name it differently, for example "Deploy to Windows Azure Web Sites - Development". Next, we will make some changes to the build configuration.

Let's start by specifying build dependencies. Under the build configuration's Dependencies, add a new snapshot dependency on our CI build. This will ensure that deployment will only be possible if a matching CI build has passed completely, and that the deployment will be based on the exact same VCS revision as we built during CI.

**Snapshot Dependencies**

Build configurations linked by a snapshot dependency will use the same snapshot of the sources. The build dependencies are built. If necessary, the dependencies will be triggered automatically. ?

+ Add new snapshot dependency

<input type="checkbox"/>	Edit Snapshot Dependency	<input type="button" value="X"/>
<input type="checkbox"/>	Depend on:	AcmeCorp.Portal :: Continuous Integration
Artifact	Options:	<input checked="" type="checkbox"/> Do not run new build if there is a suitable one <input checked="" type="checkbox"/> Only use successful builds from suitable ones <input type="checkbox"/> Run build even if dependency has failed <input type="checkbox"/> Run build on the same agent
<input type="button" value="Save"/>	<input type="button" value="Cancel"/>	

We want to be able to identify the build numbers throughout the entire chain of deployments. For example, if CI build 1.0.0 is deployed to staging, we want to be sure that this is actually version 1.0.0 and not some intermediate version. Under General Settings, change the build number format to use the same version number as the originating CI build. The build number format will have to be similar to %dep.WebAcmeCorpPortal\_ContinuousIntegration.build.number%, duplicating the version number from the CI build.

Our CI build was building the default configuration for our solution. Since we are now deploying to a different environment and we've created deployment configurations (and configuration transforms) for Development and Production, let's change the build configuration through the Visual Studio build step.

Configuration: Development

Enter solution configuration to build. Debug or Release are supported in default solution file. Leave blank to use default

Now comes the actual deployment step! Up until now, we have built our project but we haven't really done anything to ship it to an actual server. Let's change that by adding a new build step based on a Command Line runner. As the build script, enter the following:

```
"C:\Program Files\IIS\Microsoft Web Deploy V3\msdeploy.exe"
  -source:package='artifacts\WebDeploy\<target environment>\AcmeCompany.Portal.zip'
  -dest:auto,
    computerName="https://<windows azure web site web publish
URL>:443/msdeploy.axd?site=<windows azure web site name>",
      userName="<deployment user name>",
      password="<deployment password>",
      authType="Basic",
      includeAcls="False"
-verb:sync
-disableLink:AppPoolExtension -disableLink:ContentExtension -disableLink:CertificateExtension
-setParamFile:"msdeploy\parameters\<target
environment>\AcmeCompany.Portal.SetParameters.xml"
```

That's quite a bit, right? Let's go through this command:

- "C:\Program Files\IIS\Microsoft Web Deploy V3\msdeploy.exe" is the path to the msdeploy.exe which has to be available on the build agent.
- -source:package='artifacts\WebDeploy\<target environment>\AcmeCompany.Portal.zip' specifies the deployment package we want to upload
- -dest:auto,computerName="https://<windows azure web site web publishURL>:443/msdeploy.axd?site=<windows azure web site name>",userName="<deployment user name>",password="<deployment password>",authType="Basic",includeAcls="False" specifies the URL to the deployment service. For Windows Azure Web Sites, this will be in the aforementioned format. For IIS, this may be different (see Sayed Ibrahim Ashimi's excellent post on [WebDeploy parameters](#))
- -verb:sync tells WebDeploy to synchronize only changed files (this will drastically reduce deployment time as not all files will be uploaded for every deployment)
- -disableLink:AppPoolExtension -disableLink:ContentExtension -disableLink:CertificateExtension are used to disable certain configuration steps on the remote machine. These may be different for your environment, see [MSDN for a complete list](#).
- -setParamFile:"msdeploy\parameters\<target environment>\AcmeCompany.Portal.SetParameters.xml" is an important one. It specifies the WebDeploy parameters that will be replaced in the deployed Web.config file on the remote server, for example the connection string. More on this in a second.

The parameters file passed to the msdeploy.exe has to be created somehow. We've seen the build artifacts for our CI build contained a copy of this file and that one can be used if deployment secrets (such as the production database connection string) are available in source control. We probably don't want this, at least not in the same source control root our developers are all using.



Instead of storing passwords in a separate VCS root, they can also be added as a [configuration parameter](#) of type pass word in TeamCity. This will require creating the configuration file during the deployment, based on these configuration parameters.

For my setup, I've customized the AcmeCompany.Portal.SetParameters.xml file and put the configurations for the different target environments in a second VCS root, only available to the TeamCity server. This keeps the database connections strings a secret to everyone but TeamCity.

## VCS Roots

In this section you can configure how project source code is retrieved from VCS. [?](#)

[+ Attach VCS root](#)

Name

(jetbrains.git) AcmeCorp.Portal belongs to AcmeCorp.Portal

(jetbrains.git) AcmeCorp.Portal (msdeploy) belongs to AcmeCorp.Portal

We can repeat these steps to create a build configuration for staging, for QA, for production and so on. Since we want to promote builds over this entire chain, these configurations should all have a snapshot dependency on the previous environment.

Here's what this could look like: 3 different build configurations, denoting different versions that are deployed to each target environment:

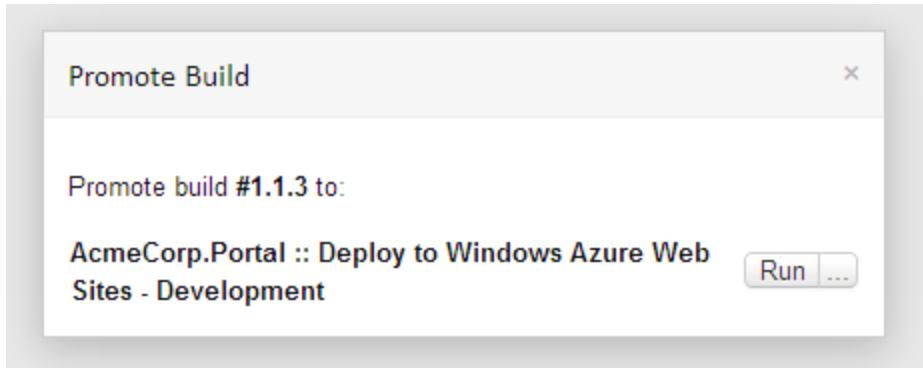
The screenshot shows the TeamCity interface with a tree view of build configurations. At the top level is 'AcmeCorp.Portal' with a 'Continuous Integration' branch. Under CI are two builds: 'master' (version #1.1.3) and '<default>' (version #1.1.0). Both builds show green status indicators for tests passed (12), artifacts, and no changes, with the latest being 'one minute ago' (2m:55s) and the other 'moments ago' (3m:56s). Below CI are two 'Deploy to Windows Azure Web Sites' branches: 'Development' and 'Production'. Each has a single build entry: 'Development' has version #1.1.3 and 'Production' has version #1.0.33. All builds show green status indicators for tests passed (12), artifacts, and no changes, with the latest being 'moments ago' (3m:56s) and '4 hours ago' (5m:22s) respectively. Each build row includes a 'Run ...' button and a close button ('x').

## Step 4: Promoting CI builds

Now that we have everything in place, let's see how we can promote builds from one environment to another. When we navigate to the build results of a CI build, we can use the Actions dropdown to promote our build to the next environment.

The screenshot shows a dropdown menu titled 'Actions' with the following options: Comment..., Pin..., Tag..., Promote..., Mark as failed..., Label this build sources..., Merge this build sources..., and Remove...'. The 'Promote...' option is highlighted with a blue background. A progress bar at the bottom indicates '100%' completion and '18/18' items.

Having configured the snapshot dependencies for our build configurations, TeamCity knows what the next environment should be: development.



This will trigger a new build that will deploy version 1.1.3 to the development environment. Once validated, we can navigate to that build's results and promote the build to the next environment.

Because of the snapshot dependencies we created, we can now also go to any build's Dependencies tab and see the environments where it has been deployed to. Here's build 1.1.3 as seen from development. We can see a CI build has been made, deployment to development has been done and deployment to production is still running:

Overview Changes Tests Build Log Parameters Dependencies Artifacts

### Snapshot dependencies

This build is part of 1 build chain. [?](#)

Page 1 of 1 (1 build chain [?](#))

[↑](#) [↓](#)

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Production

AcmeCorp.Portal :: Continuous Integration | [▼](#) [▶](#)  
master ✓ #1.1.3 Tests passed: 12 | [▼](#)

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Development | [▼](#) [▶](#)  
<default> ✓ #1.1.3 Tests passed: 12 | [▼](#)

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Production | [▼](#) [▶](#)  
<default> ✨ #1.1.3 Tests passed: 12 | [▼](#)

- For a build configuration with snapshot dependencies, we can enable showing of changes from these dependencies using the Show changes from snapshot dependencies version control setting. This enables us to see exactly which changes are deployed. See [Build Dependencies Setup - Changes from Dependencies](#) for more information.

## Conclusion

By thinking of a deployment as a chain of builds, doing deployments from TeamCity is not too hard. In this tutorial, we've used WebDeploy as an example means of transferring build artifacts to a target environment, but this could also have been another solution (like xcopy).

Using VCS labeling, it's also possible to label sources when a specific deployment happens. By pinning builds (optionally through the TeamCity API), we can make sure that build cleanup does not remove certain builds and artifacts.