

1. TeamCity Documentation	4
1.1 What's New in TeamCity 8.1	5
1.2 What's New in TeamCity 8.0	11
1.3 Concepts	21
1.3.1 Agent Home Directory	21
1.3.2 Agent Requirements	22
1.3.3 Agent Work Directory	23
1.3.4 Already Fixed In	23
1.3.5 Authentication Modules	23
1.3.6 Build Agent	24
1.3.7 Build Artifact	25
1.3.8 Build Chain	26
1.3.9 Build Checkout Directory	27
1.3.10 Build Configuration	28
1.3.11 Build Configuration Template	29
1.3.12 Build Grid	31
1.3.13 Build History	31
1.3.14 Build Log	32
1.3.15 Build Number	32
1.3.16 Build Queue	32
1.3.17 Build Runner	33
1.3.18 Build State	34
1.3.19 Build Tag	35
1.3.20 Build Working Directory	36
1.3.21 Change	36
1.3.22 Change State	36
1.3.23 Clean Checkout	37
1.3.24 Clean-Up	38
1.3.25 Code Coverage	40
1.3.26 Code Duplicates	41
1.3.27 Code Inspection	41
1.3.28 Continuous Integration	41
1.3.29 Dependent Build	42
1.3.30 Difference Viewer	43
1.3.31 First Failure	44
1.3.32 Guest User	44
1.3.33 History Build	45
1.3.34 Identifier	45
1.3.35 Notifier	46
1.3.36 Personal Build	46
1.3.37 Pinned Build	47
1.3.38 Pre-Tested (Delayed) Commit	47
1.3.39 Project	48
1.3.40 Remote Debug	49
1.3.41 Remote Run	50
1.3.42 Role and Permission	50
1.3.43 Run Configuration Policy	51
1.3.44 Super User	52
1.3.45 TeamCity Data Directory	52
1.3.46 TeamCity Specific Directories	55
1.3.47 User Account	55
1.3.48 User Group	56
1.3.49 VCS root	56
1.3.50 Wildcards	57
1.4 Supported Platforms and Environments	58
1.5 Installation and Upgrade	63
1.5.1 Installation	63
1.5.1.1 Installing and Configuring the TeamCity Server	64
1.5.1.2 Setting up and Running Additional Build Agents	69
1.5.1.2.1 Build Agent Configuration	75
1.5.1.3 Setting Up TeamCity for Amazon EC2	76
1.5.1.4 Installing Additional Plugins	78
1.5.1.5 Installing Agent Tools	79
1.5.2 Upgrade Notes	79
1.5.3 Upgrade	94
1.5.4 TeamCity Maintenance Mode	97
1.5.5 Setting up an External Database	98
1.5.6 Migrating to an External Database	103
1.6 User's Guide	105
1.6.1 Managing your User Account	105
1.6.2 Subscribing to Notifications	106
1.6.3 Viewing Your Changes	109
1.6.4 Working with Build Results	110
1.6.5 Investigating Build Problems	115
1.6.6 Viewing Tests and Configuration Problems	116

1.6.7 Viewing Build Configuration Details	117
1.6.8 Statistic Charts	117
1.6.9 Search	119
1.6.10 Maven-related Data	121
1.7 Administrator's Guide	121
1.7.1 Testing Frameworks	122
1.7.2 Code Quality Tools	122
1.7.3 TeamCity Configuration and Maintenance	124
1.7.3.1 TeamCity Data Backup	124
1.7.3.1.1 Creating Backup from TeamCity Web UI	125
1.7.3.1.2 Creating Backup via maintainDB command-line tool	126
1.7.3.1.3 Manual Backup and Restore	127
1.7.3.1.4 Backing up Build Agent's Data	128
1.7.3.1.5 Restoring TeamCity Data from Backup	129
1.7.3.2 Configuring Authentication Settings	130
1.7.3.2.1 LDAP Integration	134
1.7.3.2.2 NTLM HTTP Authentication	142
1.7.3.2.3 Enabling Guest Login	144
1.7.3.2.4 Changing user password with default authentication scheme	145
1.7.3.3 TeamCity Startup Properties	145
1.7.3.4 Configuring Server URL	145
1.7.3.5 Configuring TeamCity Server Startup Properties	146
1.7.3.6 Configuring UTF8 Character Set for MySQL	147
1.7.3.7 Setting up Google Mail and Google Talk as Notification Servers	148
1.7.3.8 Using HTTPS to access TeamCity server	148
1.7.3.9 TeamCity Disk Space Watcher	150
1.7.3.10 TeamCity Server Logs	151
1.7.3.11 Build Agents Configuration and Maintenance	153
1.7.3.11.1 Agent Pools	153
1.7.3.11.2 Configuring Build Agent Startup Properties	154
1.7.3.11.3 Viewing Agents Workload	155
1.7.3.11.4 Viewing Build Agent Details	156
1.7.3.11.5 Viewing Build Agent Logs	157
1.7.3.12 TeamCity Memory Monitor	158
1.7.3.13 Disk Usage	158
1.7.3.14 Server Health	159
1.7.4 Managing Projects and Build Configurations	162
1.7.4.1 Creating and Editing Projects	162
1.7.4.2 Creating and Editing Build Configurations	163
1.7.4.2.1 Configuring General Settings	165
1.7.4.2.2 Configuring VCS Settings	168
1.7.4.2.3 Configuring Build Steps	188
1.7.4.2.4 Adding Build Features	230
1.7.4.2.5 Configuring Unit Testing and Code Coverage	238
1.7.4.2.6 Build Failure Conditions	256
1.7.4.2.7 Configuring Build Triggers	258
1.7.4.2.8 Configuring Dependencies	265
1.7.4.2.9 Configuring Build Parameters	275
1.7.4.2.10 Configuring Agent Requirements	283
1.7.4.3 Working with Meta-Runner	283
1.7.4.4 Copy, Move, Delete Build Configuration	285
1.7.4.5 Working with Feature Branches	286
1.7.4.6 Triggering a Custom Build	289
1.7.4.7 Ordering Build Queue	291
1.7.4.8 Muting Test Failures	292
1.7.4.9 Changing Build Status Manually	293
1.7.4.10 Customizing Statistics Charts	293
1.7.4.11 Archiving Projects	294
1.7.5 Managing Licenses	295
1.7.5.1 Licensing Policy	295
1.7.5.2 Third-Party License Agreements	297
1.7.6 Integrating TeamCity with Other Tools	300
1.7.6.1 Mapping External Links in Comments	300
1.7.6.2 External Changes Viewer	301
1.7.6.3 Integrating TeamCity with Issue Tracker	301
1.7.6.3.1 Bugzilla	303
1.7.6.3.2 JIRA	303
1.7.6.3.3 YouTrack	304
1.7.7 Managing User Accounts, Groups and Permissions	304
1.7.7.1 Managing Users and User Groups	304
1.7.7.2 Viewing Users and User Groups	306
1.7.7.3 Managing Roles	307
1.7.8 Customizing Notifications	307
1.7.9 Assigning Build Configurations to Specific Build Agents	309
1.7.10 Patterns For Accessing Build Artifacts	310

1.7.11 Mono Support	312
1.7.12 Maven Server-Side Settings	313
1.7.13 Tracking User Actions	314
1.8 Installing Tools	314
1.8.1 IntelliJ Platform Plugin	314
1.8.2 Eclipse Plugin	315
1.8.3 Visual Studio Addin	317
1.8.4 Windows Tray Notifier	317
1.8.4.1 Working with Windows Tray Notifier	317
1.8.5 Syndication Feed	320
1.9 Extending TeamCity	320
1.9.1 Build Script Interaction with TeamCity	321
1.9.2 Accessing Server by HTTP	329
1.9.3 Including Third-Party Reports in the Build Results	331
1.9.4 Custom Chart	332
1.9.5 Developing TeamCity Plugins	336
1.9.5.1 Plugin Types in TeamCity	336
1.9.5.2 Bundled Development Package	337
1.9.5.3 Developing Plugins Using Maven	338
1.9.5.4 Open API Changes	339
1.9.5.5 Plugins Packaging	344
1.9.5.6 Server-side Object Model	347
1.9.5.7 Agent-side Object Model	349
1.9.5.8 Extensions	349
1.9.5.9 Web UI Extensions	350
1.9.5.10 Plugin Settings	353
1.9.5.11 Development Environment	354
1.9.5.12 Typical Plugins	354
1.9.5.12.1 Build Runner Plugin	355
1.9.5.12.2 Risk Tests Reordering in Custom Test Runner	356
1.9.5.12.3 Custom Build Trigger	357
1.9.5.12.4 Extending Notification Templates Model	357
1.9.5.12.5 Issue Tracker Integration Plugin	358
1.9.5.12.6 Version Control System Plugin	360
1.9.5.12.7 Version Control System Plugin (old style - prior to 4.5)	362
1.9.5.12.8 Custom Authentication Module	366
1.9.5.12.9 Custom Notifier	370
1.9.5.12.10 Custom Statistics	371
1.9.5.12.11 Custom Server Health Report	372
1.9.5.12.12 Extending Highlighting for Web diff view	373
1.9.5.12 REST API	379
1.10 How To...	390
1.11 Troubleshooting	402
1.11.1 Common Problems	402
1.11.2 Known Issues	406
1.11.3 Reporting Issues	412
1.11.4 Applying Patches	419
1.11.5 Enabling Detailed Logging for .NET Runners	420
1.11.6 Visual C Build Issues	421
1.12 Getting Started	421
1.12.1 Continuous Delivery to Windows Azure Web Sites (or IIS)	425
1.12.2 Getting started with PHP	435
2. Build Configuration General Settings	442

TeamCity Documentation

TeamCity 8.x

Welcome to Documentation space for TeamCity 8.x - distributed build management and continuous integration server. Here you will find description, instructions and notes about installing, configuring and using TeamCity, its features, options, and plugins.

Documentation Space Structure

For your convenience, the documentation space is divided into following sections:

Installation and Upgrade

Refer to the [Installation](#) section if you are installing TeamCity for the first time.

Review [Upgrade Notes](#) and see the [Upgrade](#) section if you are upgrading your existing TeamCity instance.

User's Guide

TeamCity User's Guide is for everyone who uses TeamCity. From this section you will learn how to subscribe to TeamCity notifications, view how your changes have affected different projects, view current problems, use TeamCity search.

Administrator's Guide

The Administrator's Guide contains essential instructions on configuring and maintaining the TeamCity server and build agents, working with projects and build configurations, managing users and their permissions, integrating TeamCity with other tools, and more.

Concepts

This section is intended to give you basic understanding of TeamCity-specific terms, like Build Configuration, Project, Build Queue, etc. If you're new to TeamCity it is recommended to familiarize yourself with these concepts.

Misc

If you have a question on TeamCity it's worth checking out [How To...](#) section and pages under [Troubleshooting](#).
Also check out our [video user guide](#).

Where to Find More Information and Send Feedback

To get general information about TeamCity and its features, please visit [TeamCity Official Site](#).

On the [Official TeamCity Blog](#) and [TeamCity Developers Blog](#) you will find the latest news from our team, handy feature's descriptions, usage examples, tips and tricks.

You can also follow us on [Twitter](#).

Feedback

You are welcome to contact us with questions or suggestions about TeamCity. See [Feedback](#) for appropriate contact method. The link is also available in the footer of any web page of TeamCity server.

Documentation for previous TeamCity versions

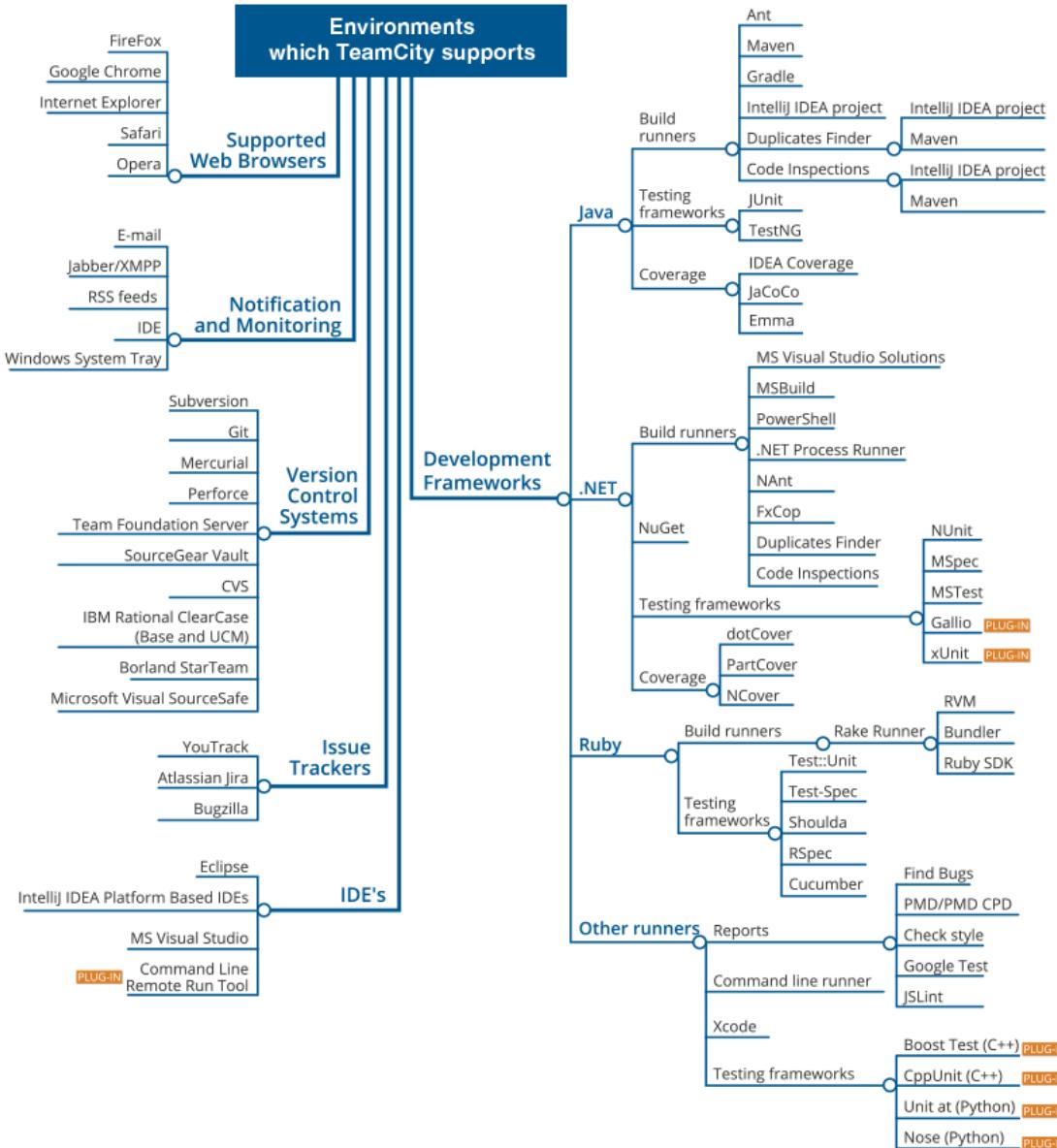
If you're using TeamCity earlier than 8.0, please refer to the corresponding [documentation](#).

TeamCity Plugins

There is a number of plugins available for TeamCity developed both by JetBrains and third parties. See the [list](#).

TeamCity Supported Platforms and Environments

Have a "10,000-foot look" at TeamCity, and the supported IDE's, frameworks, version control systems and means of monitoring. Point to a component on the diagram below and jump to its description:



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* and *Mac OS* 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 8.1

- Simplified setup
 - Create project from URL
 - Suggested settings
 - Advanced and default settings

- Easier settings override in inherited build configurations
- Easier build artifacts configuration
- Listing build targets
- Project and build configuration navigation
- External database set-up on the first start
- Additional build features
 - Feature branches auto-merge
 - VCS labeling build feature
- Project-level settings
 - Per-project Maven and Build report tabs
 - Per-project SSH Keys
- More Health Reports
- Changes page aka "My Changes"
- Statistics improvements
- Build failure condition on a metric change
- Improved build run time estimation
- REST API Improvements
- Support for JaCoCo
- Support for Visual Studio 2013, Subversion 1.8, Xcode 5 and more
- Backup / Restore improvements
- Other improvements
- Previous Releases



Starting with TeamCity 8.0 each Build Agent license adds 10 more build configurations to Professional edition (as well as one agent)

Simplified setup

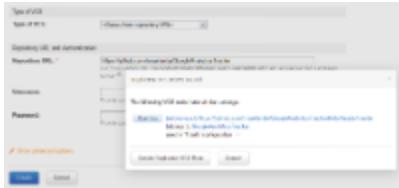
Create project from URL

If you have a URL to a project in a version control system, you can easily [create a project in TeamCity using this URL only](#). TeamCity will detect the type of your version control system, create a [VCS root](#) for you in the new project, verify connection to the repository, create a [build configuration](#), and propose pre-configured [build steps](#) to run your build. It was easy to create a new project before, but now it's really a no-brainer.

TeamCity auto-detection mechanism supports [most popular version control systems](#) and a wide range of build tools and technologies, including: Maven, Gradle, Ant, NAnt, MSBuild, Visual Studio solution, Powershell, Xcode, Rake, IntelliJ IDEA, as well as various command line scripts.

Similarly to creating a project from URL, you can also create a [VCS root](#) from a repository URL only:

Besides that, if while creating a new VCS root TeamCity detects a duplicated VCS root, it will show a warning and suggest using the existing VCS root:



Suggested settings

If there are some configuration changes that could improve your build configuration setup, TeamCity will suggest them via a new suggested settings icon:



Advanced and default settings

TeamCity has a lot of options and configuration tweaks. As usual, most of them are useful in a minority of cases, but they often distract new users and make the product look more complicated than it actually is. To address this problem, we introduced advanced settings. All complicated options and the settings used not so often are now marked as advanced and are hidden by default (unless their values were changed, in which case the modified advanced settings are automatically shown).

The settings of a build configuration whose values have been changed from defaults are now marked with an orange border. If a build configuration is inherited from a template, the template settings play the role of the defaults; thus, all settings redefined in a build configuration can be easily located.



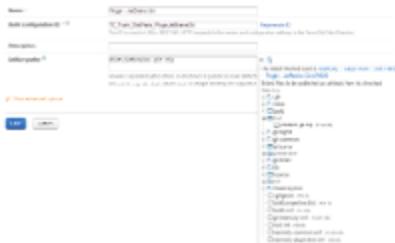
Easier settings override in inherited build configurations

For a build configuration inherited from a [template](#), the settings like the maximum number of running builds, execution timeout, build failure conditions and some others could not be overridden even with the help of parameter references. In TeamCity 8.1 we addressed this problem. The following settings can now be overridden and this can be done in place, without introducing new parameters:

- build number format
- artifact paths
- build options (hanging builds detection, status widget, number of simultaneously running builds)
- VCS checkout mode
- checkout directory
- clean all files before build
- show changes from snapshot dependencies
- execution timeout
- all common build failure conditions, including execution timeout

Easier build artifacts configuration

If your build produces [artifacts](#), you can configure TeamCity to publish these artifacts to the server. Previously the process was a bit tedious, because you had to know [the exact paths](#) to the artifacts. In 8.1 we added the ability to browse the checkout directory of a build. Now, once a build is finished on some agent, you can easily browse the checkout directory and pick artifacts:



Listing build targets

The available Ant, NAnt and MSBuild targets can be viewed right in TeamCity web interface:



The same is supported for IntelliJ IDEA run configurations, artifacts and inspection profiles.

Project and build configuration navigation

Project and build configuration editing pages have been reworked for better consistency:

- the project tab navigation changed to the vertical layout
- the navigation sidebar inside build configuration pages has been moved to the left
- the actions like copy, move, extract template, have been moved from the sidebar to the upper right **Actions** menu

External database set-up on the first start

When setting up TeamCity for the first time, you can configure it to use an external database right on the server start-up: just select the type of the database and specify the database connection settings. However, you may still need to download the JDBC driver for your database.



Additional build features

Feature branches auto-merge

If you are using feature branches with Git and Mercurial, with the help of the **Automatic Merge Build Feature** you can now configure TeamCity to merge a feature branch automatically to the master if a build in the feature branch completes successfully. There is also an option to merge manually from the build's **Changes** tab.



The whole user experience is similar to the **Pre-tested commit** feature. Developers push commits in their own branches. TeamCity monitors these branches and runs a build on the pushed changes. If a build satisfies the merge criteria (successful, or there were no new failures in it), the branch is merged by TeamCity into default/master or any other branch configured in the auto-merge build feature. Unlike pre-tested commit, no IDE is required to use this feature.

VCS labeling build feature

The [VCS labeling](#) settings were reworked as a [Build Feature](#) instead of options in the VCS Settings section. The existing settings are converted automatically to the new format. The build feature allows for more flexible configuration: you can now have different labeling settings for different VCS roots, or disable and redefine the labeling settings inherited from a template.



Project-level settings

Per-project Maven and Build report tabs

Previously Maven settings and build report tabs could be configured on a global level only and only system administrators were able to manage them. Since version 8.1 these settings are moved to the project level and are now available for editing by project administrators.

As with any other project-level settings, Maven settings and Build report tabs defined in some project are available not only to this project but to all sub-projects as well. Existing settings will be moved to the Root project during server upgrade.

Per-project SSH Keys

If your [Git](#) repository requires SSH access, to use it in TeamCity in earlier versions you had to put the SSH private key in some location on the server, and in case of the agent side checkout - to each agent where your build was to be executed.

Now, you can [upload](#) an SSH private key right into the project using the TeamCity web interface. You can then configure the Git VCS root to use this uploaded key.

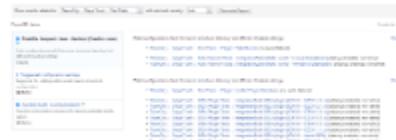
More Health Reports

We added several new health reports:

- detect problems with reusing builds for configurations with snapshot dependencies
- detect queued builds sitting in the queue without compatible agents
- report cloud agent issues (e.g. the agent machine cannot be stopped, or an agent is in the idle state longer than the timeout)
- detect exhaustion of the server disk space and pause the build queue in such a case
- detect missing Maven settings
- detect database collation mismatch in TeamCity tables

All suggested settings detected by TeamCity are shown under the special **Suggested settings** category in the server health report. Some of the health reports that do not require administrator's permissions are now available in the user space, on the build configuration home page.

Finally, server health report has been redesigned a bit:



Changes page aka "My Changes"

The page previously known as [My Changes](#) now has a user selector allowing you to see modifications made by any other TeamCity user the same way you see your own changes. As a result the page was renamed to **Changes**. Besides, instead of the carpet showing the status of builds, 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.



Statistics improvements

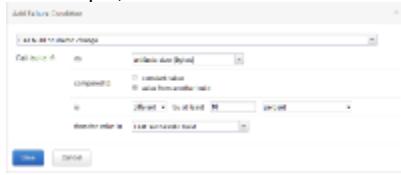
We expanded [statistics charts](#) to occupy the whole page width. Statistics charts now support zooming. In addition, all statistic values published by

a specific build are now available on the Build Parameters tab with the ability to see history of changes for reported values:



Build failure condition on a metric change

A build failure condition on a metric change has been extended with the support for percentage. You can now fail a build if the number of tests is, for example, 10% lower than in some other build. We have also redesigned the web interface to make it more straightforward:



Improved build run time estimation

TeamCity always estimated build duration based on the average run time of the latest builds in the history of a build configuration. Starting with 8.1 we switched to a new algorithm taking into account build stages reached by a running build. For example, if a build is still executing a build step and TeamCity knows that there are two more build steps, the system will calculate the time left for the current step execution and add the time required to complete the next two steps.

REST API Improvements

Since TeamCity 8.1, new features are available in TeamCity REST API. It can now be used to:

- trigger a build; while triggering a build, a change to the build can be specified as well as all the rest of the settings from the custom run build dialog.
- list builds in the build queue
- stop a running build
- access the canceled build details
- query for investigations for a build problems and tests in addition to build configuration investigation
- list investigations assigned for a user
- list the build's tests
- get test run history
- delete a build agent
- assign project to an agent pool.

As a security measure, users without the VIEW_USER_PROFILE (View user profile) or CHANGE_USER (Modify user profile and roles) permissions can no longer list users and user groups.

Support for JaCoCo

JaCoCo coverage engine is now available for Maven, Ant, Gradle and IntelliJ IDEA builds.

For now JaCoCo coverage results are shown in TeamCity web interface only and cannot be loaded from TeamCity in IDE.

Support for Visual Studio 2013, Subversion 1.8, Xcode 5 and more

TeamCity is now fully compatible with Microsoft Visual Studio 2013, Team Foundation Server 2013, Subversion 1.8 and Xcode 5.

In addition, we bundled the fresh version of Maven - 3.1, and upgraded Ant to version 1.8.4. The bundled dotCover has been upgraded to version 2.6.

Backup / Restore improvements

The duration of the process of [data restoring into an external database](#) has been decreased. Since there were many problems with failed restore process due to the lack of disk space, we added a new "--continue" command to continue process from the point where it failed.

Other improvements

- several security issues were fixed related to web UI XSS vulnerability and partial information exposure for users without sufficient permissions
- the packages restore command introduced in NuGet 2.7 is now supported
- the **Copy** action is now available for templates
- an artifact dependency can be configured on a previous build of the same build configuration ([TW-12984](#))
- you can browse the [TeamCity Data Directory](#) right from the web interface using the [Administration|Global Settings|Browse Data Directory](#) tab; it is also possible to upload new files and modify the existing ones
- changing IDs of a build configuration, project or VCS root does not invalidate bookmarked links with previous IDs
- project names in the UI got drop-down menus with all the sub-tabs and the **Edit Settings** link for quick navigation. Build configurations now also list all the sub-tabs in their drop-downs
- Save** and **Cancel** buttons are moved to the left
- a project can be selected in the **Investigation** dialog when an investigation is assigned for a test. This is useful when the test fails in a sibling projects
- a new option in YouTrack integration allows for monitoring all projects from the YouTrack server automatically
- a new build parameter is introduced to pass [the user who triggered the build](#) into the build script ([TW-4502](#))
- the [Shared Resources](#) build feature has been improved to pass all the taken locks into the build. See the related [upgrade notes](#) entry
- the schedule trigger has got the timezone setting for the cron-based trigger
- the TeamCity JIRA integration now uses JIRA REST API instead of RPC
- Ruby Environment Configurator now supports RVM with .ruby-version files
- [sources checkout](#) has been reworked to reduce the number of unnecessary [clean checkouts](#)
- all artifacts published by a build are now cached in the local artifact dependencies cache on the agent, this should speedup artifact dependencies in some cases
- fixed issues

Previous Releases

[What's New in TeamCity 8.0](#)

What's New in TeamCity 8.0

- Projects Hierarchy
- Human Friendly Identifiers
- Project Directory Structure
- Mixed Mode Authentication
- Faster Build History Clean-up
- Disk Usage Report
- Server Health Report
- Meta-Runner
 - Preparing Build Configuration
 - Verifying How Build Configuration Works
 - Extracting and Using Meta-Runner
- Shared Resources
- Build Problems
- Queued Build Page
- Feature Branches Improvements
- Changes from Sub-repositories in Mercurial
- IntelliJ IDEA Project Runner
 - IntelliJ IDEA Based Make
 - Remote Debug from IDE
- Test Status in IntelliJ IDEA
- .NET Inspections Improvements
 - Full MSBuild Model Support
 - Analysis Results Caches
- Other Noteworthy Changes
 - Updated Look and Feel
 - Change Log Page
 - Statistics
 - Builds Schedule Tab
 - Remove Build Action
 - Plugin Development
 - REST API
- Fixed Issues
- Previous Releases



Starting with TeamCity 8.0 each Build Agent license adds 10 more build configurations to Professional edition (as well as one agent)

Projects Hierarchy

TeamCity 8.0 allows creating subprojects inside other projects. Now all projects form a tree (each project can have one parent project only) with the **<Root project>** at the top of the hierarchy. The **<Root project>** is created by default and cannot be removed.

Project hierarchy affects the server behavior in many places:

- VCS roots created in a project are available in this project and in all subprojects down the hierarchy. As such, the concept of shared VCS root has been eliminated, and all shared roots are moved to the **<Root project>** during upgrade.
- Similarly to VCS roots, build configuration templates created in a project are available in this project and in all subprojects. 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.
- A project parameter defined for a project will be available in build configurations of this project and all subprojects. A parameter defined in the **<Root project>** effectively becomes global, i.e. appears in all build configurations.
- Roles assigned to a user or group in a project propagate to its subprojects. For example, if a user is assigned the Project administrator role in a project, this user will have the Project administrator role in all subprojects of this project. On the other hand, if the **View project** permission is assigned to someone in a project, this user will have the **View project** permission for all the parent projects too.
- A notification rule assigned to a project applies to all its subprojects too.
- If an investigation is assigned to a test or problem in some project, it propagates to its subprojects too. The same applies to mutes.
- When a project is archived, all its subprojects are archived too.

Different pages in the user interface now include results from subprojects:

- The project **Change Log** page now shows changes from the current project and all subprojects.
- The project **Current problems/ Investigations/ Muted Problems** tabs show data from the current project and all its subprojects.
- **Test History** shows results from the project and its subprojects.
- If project is selected in **Administration -> Audit**, the information on all its subprojects will be displayed too.

Subprojects are also shown in breadcrumbs, all projects popup, choosers with projects and build configurations and in many other places.

The screenshot shows the TeamCity web interface with the 'Projects' tab selected. The main content area displays a hierarchical list of projects. At the top level, there is 'TeamCity' which contains several sub-projects: 'Gaya Trunk', 'DB Tests', 'IDEA Plugin Tests', 'Server Code Analysis', 'VS Plugin', 'TeamCity Plugins', 'Faredi 7.1.x', 'Dxt Parts', 'TeamCity Inspections Samples', 'Stats', 'Faredi 7.0.x', and 'TeamCity Tools'. Each project entry includes a small icon and a numerical value indicating the count of builds or configurations. The sidebar on the left shows 'Administration' and 'Build Configuration' sections.

Human Friendly Identifiers

Each configurable entity in TeamCity, be it a project or build configuration, always has an identifier |(ID) assigned. These IDs become important if some automated actions are to be performed with this entity. For example, one may want to add a build to the queue right from a build script. Another common task is to fetch artifacts in a build script from the latest build of some build configuration. REST API is often used to create or modify build configurations, VCS roots or even projects. IDs are used for all the above-mentioned cases. Until TeamCity Version 8.0, these IDs were not editable and as such were not human friendly (e.g. **bt125**).

Now, in version 8.0, you can assign your own IDs to projects, build configurations, templates and VCS roots. All these entities now have the **ID** field in their settings. **ID** is a mandatory field, but TeamCity fills it out automatically based on the current object name and the ID of the parent object. Once the ID is provided, it can be used in build scripts and in REST API.

IDs are also used in several other places, like URLs in web interface, file and directory names under **<TeamCity data directory>**. They are also used in **dep.** and **vcsRoot** parameter references.

Since **ID** is a mandatory field, it appears in some other places in the administration interface, like the **Copy** dialog for projects and build

configurations. During upgrade, TeamCity will generate IDs for projects and VCS roots based on the project and VCS root names. It will not generate IDs for build configurations and templates though. If you want human friendly IDs for all build configurations in your project, you can use the **Bulk edit IDs** button in the sidebar of the edit project page.

And last but not least, human friendly IDs are also used across all project configuration files stored on the disk under [<TeamCity data directory>/config](#). This makes it possible to move configuration files from one server to another, provided that IDs used for objects are unique across these servers.

Related topic in our documentation: [Identifier](#)

Project Directory Structure

TeamCity does not store project configuration files in a database, instead it stores them on the disk under the [<TeamCity data directory>/config](#) folder. Configuration files are stored in the xml format. Before version 8.0, TeamCity stored all VCS roots in a single `vcs-roots.xml` file. All build configurations and templates of a project were stored in a single `project-config.xml` file. In version 8.0 the `project-config.xml` file has been divided into several parts, and VCS root definitions were moved under the project directory. So now the project directory containing all project configuration files looks as follows:

```
<TeamCity data directory>/config/projects/<project id>/
  buildNumbers/
  buildTypes/
  vcsRoots/
  pluginData/
  project-config.xml
```

- `buildNumbers` directory contains current build numbers of all build configurations of the project with file names like: `<build configuration ID>.properties`
- `buildTypes` directory contains definitions of build configurations and templates, one file per definition, with file names like: `<build configuration/template ID>.xml`
- `vcsRoots` directory contains definitions of VCS roots, one file per definition, with file names like: `<vcs root ID>.xml`
- `pluginData` directory contains plugin specific configuration files.

Mixed Mode Authentication

TeamCity server supports the following authentication mechanisms: built-in authentication (enabled by default), Windows Domain (with optional NTLM HTTP authentication), and LDAP. However, before version 8.0, these authentication modules were mutually exclusive, i.e. it was not possible to have the built-in and domain authentication at the same time, although such setup could be convenient during migration from one authentication mode to another.

In version 8.0 we implemented a more sophisticated authentication mechanism, which we call "mixed mode". Starting from this build, you can configure several authentication modules, so when a user attempts to log in, TeamCity will try all the modules 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. We also implemented a user interface to simplify editing of the authentication configuration (in the previous versions you had to manually edit the `main-config.xml` file). Finally, presets were created for the most common use cases:

- Default (built-in authentication only)
- LDAP
- LDAP with NTLM
- NTLM

Some modules have the settings which can now be edited from the user interface too. For advanced users there is an advanced mode allowing them to add / remove authentication modules. These options should be handled with great care. On changing authentication modules, all the existing users with their settings are preserved if their usernames in the old and new authentication modules remain the same. If this is not the case, the username for specific authentication module can be specified in the user profile (see [Authentication Settings](#)).

Additionally, there is a special **super user** account with system administrator role. This account is not editable and it does not have a profile. When the server starts, it generates a password for this account and writes it in `teamcity-server.log`. This account can be used for password recovery; all maintenance tasks are performed on behalf of the **super user**.

Related topics in our documentation:

- [Configuring Authentication Settings](#)
- [Super User](#)

Faster Build History Clean-up

TeamCity is critical software for many organizations. Many of our users have distributed teams, working with the server all the time. At the same time, TeamCity must perform some clean-up tasks to remove obsolete builds and their artifacts, clean-up obsolete data from database and

caches. These clean-up activities are also important to keep the server performance at the desired level. Before version 8.0, the clean-up task could take a significant amount of time, and the clean-up duration was unpredictable. This affected projects with distributed teams as the server was not accessible for many team members during the clean-up process.

In version 8.0 we made a lot of performance improvements in the clean-up process. Several time consuming activities were moved to the background. Moreover, it is now possible to set the clean-up execution time for the server.

Besides the general speedup, there are other improvements in the clean-up process:

- Clean-up rules can be defined for a project, in this case they act as default for all configurations and subprojects.
- Orphaned build artifacts and build logs (i.e. files not attached to any build in history) are now cleaned out as well. Thus, the server can be safely stopped during the clean-up process, and the data not yet removed will be cleaned out next time.
- The clean-up progress is now displayed.
- The clean-up process can be stopped by system administrator.

Disk Usage Report

Each build occupies some disk space under <TeamCity data directory> folder. The space is used by build artifacts and build logs. One useful addition in version 8.0 which nicely complements our [clean-up improvements](#) is the **Disk Usage** report. With the help of this report you can easily find the projects consuming most of disk space and configure the clean-up rules accordingly. Besides this, the report provides summary for the total free disk space, number of pinned builds, etc. Disk usage is shown on the build history clean-up page as well.



This screenshot shows a detailed view of disk usage across multiple projects. The main table lists projects and their respective disk usage metrics. Projects include 'Main' (569 MB free), 'Test' (569 MB free), 'Build' (569 MB free), 'J2EE' (569 MB free), 'Mobile' (569 MB free), 'Sitecore' (569 MB free), 'Bazaar' (569 MB free), 'MSBuild' (569 MB free), 'ReSharper' (569 MB free), 'Subversion' (569 MB free), 'Mercurial' (569 MB free), 'TortoiseSVN' (569 MB free), 'Jenkins' (569 MB free), 'Hudson' (569 MB free), 'Gerrit' (569 MB free), 'Trac' (569 MB free), 'TeamCity' (569 MB free), 'Apache' (569 MB free), and 'Tomcat' (569 MB free). A summary table below provides totals for pinned builds (16), pinned artifacts (10), and total free disk space (627.5 MB).

Project	Total Free Space	Pinned Builds	Pinned Artifacts
Main	569 MB free	0	0
Test	569 MB free	0	0
Build	569 MB free	0	0
J2EE	569 MB free	0	0
Mobile	569 MB free	0	0
Sitecore	569 MB free	0	0
Bazaar	569 MB free	0	0
MSBuild	569 MB free	0	0
ReSharper	569 MB free	0	0
Subversion	569 MB free	0	0
Mercurial	569 MB free	0	0
TortoiseSVN	569 MB free	0	0
Jenkins	569 MB free	0	0
Hudson	569 MB free	0	0
Gerrit	569 MB free	0	0
Trac	569 MB free	0	0
TeamCity	569 MB free	0	0
Apache	569 MB free	0	0
Tomcat	569 MB free	0	0
Total	627.5 MB free	16	10

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:



This screenshot displays the build history details for a configuration named 'Bazaar'. It lists individual build steps and their corresponding disk usage. For example, the 'Initial checkout' step uses 2.3 MB, while the 'Post-build cleanup' step uses 400 KB. The 'Total' usage for the configuration is 3.7 MB. A summary table at the bottom provides a breakdown of pinned artifacts (2.3 MB), pinned builds (400 KB), and total free disk space (3.7 MB).

Step	Size
Initial checkout	2.3 MB
Building	100 KB
Post-build cleanup	400 KB
Total	3.7 MB

Server Health Report

The **Server Health** report is another significant addition made in TeamCity version 8.0. This report shows different configuration problems, so called server health items. These items can be global, or project related. There are several reports providing such items, and, as the report is pluggable, it is possible to write a plugin which will report on some specific items. Currently the following reports are available:

- Agents not used for 3 days or more
- Configuration with large build logs
- Possible frequent clean checkout
- Possible frequent clean checkout (Swabra case)
- Similar VCS root usages
- Similar VCS roots
- Unused VCS roots

We are planning to significantly extend the number of reports in the subsequent versions of TeamCity.

Meta-Runner

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.

Let's consider an example of a [meta-runner](#). In this case we'll create a 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. So let's 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 thing we need to do is to prepare a build configuration which will work the same way as the meta-runner we'd like to produce. We'll use the configuration with a single Ant build step. We've chosen Ant because it 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 **Build 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.

Verifying How Build Configuration Works

Once we defined build steps and parameters, we can check how 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 **Extract Meta-Runner** button in the build configuration settings sidebar:



The **Extract Meta-Runner** dialog requires specifying the project where the meta-runner will be created. As with VCS roots and templates, if a meta-runner is created in a project, it will be available in this project and all subprojects down the hierarchy. 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 (by default, the ID will be generated from the meta-runner name).

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:



The current meta-runner usages can be seen at the project **Meta-Runners** tab:

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.

Since the 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.

Shared Resources

Builds can use different resources (databases, web services, etc.). Some of these resources can be accessed concurrently but allow only a limited number of connections, others require exclusive access. To address these cases, we introduce the **Shared Resources** build feature. 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.

Build Problems

In this version TeamCity is even more efficient in detecting build problems. A build problem is something that causes a build to fail. For example, all build failure conditions, once detected in a specific build, actually result in a build problem being added to the build. Once the problem is reported, the build fails. All problems reported for the build are now shown on the build results page and in the build summary popup. TeamCity also tracks the new/not new status for build problems, and allows muting them or assigning investigations for them. Reported problems are shown on the Current problems, Investigations and Muted problems pages. So now you can work with problems the same way you work with tests.

Additionally, there is a new service message that allows reporting on a custom problem for the current build.

Queued Build Page

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

The build results page shows different details for a running or finished build. However some of these details, like Changes, Build parameters and Dependencies are also applicable to the build while it is waiting in the queue. In version 8.0, a queued build has its own page where all these details are displayed. Additionally, the queued build page is a convenient place for builds with snapshot dependencies, because there you can see the progress for all dependencies of the build chain, estimated duration, included changes, status, and so on:

Feature Branches Improvements

Since version 7.1 we have received a lot of feature requests for further improvements in feature branches support. In this release we tried to address the most important of them:

- the build branch name can be specified in the artifact dependency

- the VCS, Schedule and Finish build triggers now support the branch filter setting
- the VCS labeling also supports the branch filter
- Mercurial bookmarks can now be used in the branch specification, so, if you prefer to use bookmarks instead of standard Mercurial branches, you can now fully use 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. This is now supported too.

Related topics in our documentation:

- [Working with Feature Branches](#)
- [Branch Remote Run Trigger](#).

Changes from Sub-repositories in Mercurial

TeamCity has supported Mercurial sub-repositories from the beginning. However, if you use sub-repositories, you're aware that TeamCity did not detect changes in them. When `.hgsubstate` was modified to include recent changes from sub-repository all you could see in TeamCity user interface is commit modifying this `.hgsubstate` file. Starting from version 8.0, you can configure TeamCity to also show changes from a sub-repository brought by the `.hgsubstate` file modification. See how it works in action:

IntelliJ IDEA Project Runner

IntelliJ IDEA Based Make

If you use IntelliJ IDEA, you probably know that since version 12.0 it uses a new external make, i.e. the make which works out of main IntelliJ IDEA process. We worked together with IntelliJ IDEA team to integrate this make with the **IntelliJ IDEA Project** runner in TeamCity. Starting from TeamCity version 8.0, **IntelliJ IDEA Project** runner builds your project using the same make tool which is used by IntelliJ IDEA itself. Basically it means that all of the goodies available in the IntelliJ IDEA make are now available in TeamCity too. To name a few:

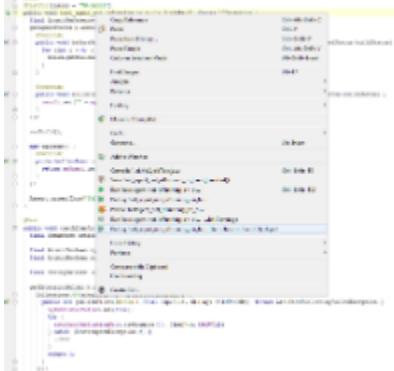
- support for different languages like Groovy, Scala, Closure, Kotlin, etc.
- ability to compile Android projects and build .apk artifacts.
- ability to build many other artifact types supported by IDEA itself (war, ear, etc.).
- faster rebuild, thanks to the highly optimized IDEA external make build process, and even faster incremental building (can be enabled on the IntelliJ IDEA project runner page).
- since the tool is basically the same, there are no any incompatibilities between TeamCity and IntelliJ IDEA anymore.

One additional feature for those who develop for Android in IntelliJ IDEA: the **Inspections (IntelliJ IDEA)** runner now supports Android SDK as well.

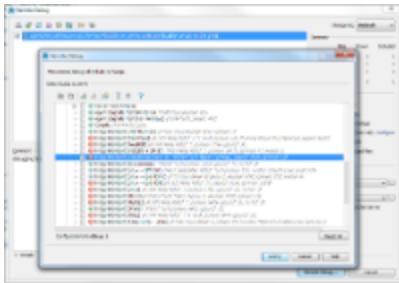
Remote Debug from IDE

Sometimes there are cases when, to fix a test, you have to debug it right on the agent. Usually this happens when the agent environment is unique in some aspect, which causes some test to fail, which is not easy or impossible to reproduce locally. If you happen to use the IDEA Project runner, and if you're using TeamCity plugin for IntelliJ IDEA, then now you have a good tool at hand to help you in such cases.

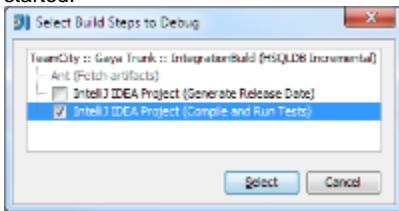
To start remote debugging of a test, you need to select the test and choose an appropriate option from the context menu (the remote debug action is also available in the TeamCity plugin menu):



Once you do this, the TeamCity plugin will ask you to select a build configuration where you want to start the debug session. Actually, 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 has more than one build step, the plugin will ask to choose one or more build steps where the debug session must be started:

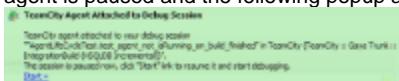


Note that only build steps with the **IntelliJ IDEA Project** runner are supported.

After that a build starts on the agent 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 following popup 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:



Our debug session is not finished yet, now we can either repeat or finish it. If you ever tried to debug a tricky case (although it's always tricky, otherwise we wouldn't have started the debug session), you should understand how important the Repeat action can be. It will repeat the same build step again and it works much faster than starting a new debug session.

Since the remote debug session works with IntelliJ IDEA run configurations, it is important to mention that at the moment not all types of run configurations are supported by TeamCity. The currently supported types are:

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

Test Status in IntelliJ IDEA

You can configure TeamCity IntelliJ IDEA plugin to show the test status obtained from the server right in the editor, near the test name. This indication can save developers time for the cases when a test is muted on the server or is being investigated by some other developer.



Test status indication can be enabled through the menu of the TeamCity plugin for IntelliJ IDEA.

.NET Inspections Improvements

Full MSBuild Model Support

We worked together with our colleagues from the ReSharper team to fix a large set of bugs in the server-side .NET code analysis. Generally, all those bugs could be described as "There are no errors (warnings) in Visual Studio, but TeamCity shows a number of problems in my code." The reason was that the project model which is an input for code analysis was incorrectly constructed outside of Visual Studio. In order to fix this, we now support all of MSBuild-related features usually used by developers. Some of them are:

- MSBuild properties usages in project files ([TW-19854](#), [TW-20837](#), [TW-23892](#), [TW-25898](#))
- Auto-generated code usages ([TW-19220](#), [TW-24921](#))

Analysis Results Caches

We have also received reports about poor performance of code analysis on the server (compared to ReSharper). To mitigate this, all data produced by code analysis which could be reused in future builds on the same build agent is stored outside of the checkout directory (by default) and is cleaned out together with other agent caches.

Other Noteworthy Changes

Updated Look and Feel

As usual, we've made numerous improvements to the overall look and feel. All the icons have been redesigned and now support the new high-resolution displays.

Change Log Page

- commit graphs can be collapsed
- changes can be filtered by comment
- build tags and pin status are shown for build rows

Statistics

We've changed the way how statistics charts are generated, now we use JavaScript based library and charts are rendered on the browser. This makes it possible to make charts more interactive. For example, now you can easily zoom any region of a chart.

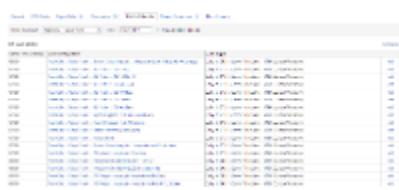
Besides this, some additional features were added:

- Y-axis settings can be changed.
- Chart data can be downloaded in the CSV format.
- Auto-generation of colors for series has been improved.
- A custom color can be set for a custom chart series in the chart configuration file.



Builds Schedule Tab

If you use Schedule triggers a lot, then you probably tried to find a window on your system where you could squeeze yet another build configuration. In version 8.0 this task is much easier thanks to the new **Builds Schedule** tab available on the project level. This tab shows all schedule triggers configured in the project and its subprojects in a single timeline, so you can easily observe what happens and at what time on your system.



Conveniently, the schedule trigger itself now has a timezone setting, which should help if your server and teams working with it are in different timezones.

Remove Build Action

A single build or the whole build chain which the given build participates in can now be removed from the database with the help of the **Remove** action available from the **Build actions** menu.

Plugin Development

TeamCity server and agent API is published in JetBrains Maven repository, so now it should be easy to create plugins with help of Maven. We also provided Maven archetype for TeamCity plugin generation. Read more on this [here](#).

REST API

- Build artifacts can be listed via [REST API](#)
- Build configuration labeling settings can be retrieved or modified by REST

Fixed Issues

There are lots of fixed issues including security-related ones. See the [full list](#).

Previous Releases

[What's New in TeamCity 7.1](#)

Concepts

This part gives a list of basic terms and their definitions, that will help you successfully start and work with TeamCity:

- 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
- Continuous Integration
- Dependent Build
- Difference Viewer
- First Failure
- Guest User
- History Build
- Identifier
- Notifier
- Personal Build
- Pinned Build
- Pre-Tested (Delayed) Commit
- Project
- Remote Debug
- Remote Run
- Role and Permission
- Run Configuration Policy
- Super User
- TeamCity Data Directory
- TeamCity Specific Directories
- User Account
- User Group
- VCS root
- Wildcards

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 Build Agent, the agent will be placed into <TeamCity Home>/buildAgent.

The default directory suggested by the .exe or Java Web Start agent installation is C:\BuildAgent.

The agent stores all related data under its directory and the only place that requires installation/uninstallation into OS is integrating into the automatic start system (e.g. service settings under Windows). See [Setting up and Running Additional Build Agents](#) for details.

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). Unique piece of information defining the agent settings and behavior.
- `agent work directory` (stored under `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 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 build agent using installed build agent service
 - `service.stop.bat` — stops installed build agent service
 - `service.uninstall.bat` — batch file to uninstall currently installed build agent Windows service
- `/conf/` — this folder contains all configuration files for build agent
 - `buildAgent.properties` — [main configuration file](#). This file would be generated by TeamCity server .exe installer, build agent .exe installer and build agent Java Web Start agent installer.
 - `buildAgent.dist.properties` — sample configuration file. You can rename it to 'buildAgent.properties' to create 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 template for creating original configuration
 - `wrapper.conf` — current build agent Windows service configuration. This is 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 build agent launcher
 - `teamcity-agent.log` — main build agent log
 - `wrapper.log` — log from Java Service Wrapper. Available only if build agent is running as windows service
 - `teamcity-build.log` — log from build
 - `upgrade.log` — log from build agent upgrade process
 - `teamcity-vcs.log` — agent-side checkout logs
- `/temp` — temporary folder; path may 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 default temp directory for build process and is cleaned after each build
 - `globalTmp` — temporary folder that is used by the build agent for own temporary files. Is cleaned on 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's [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 `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.

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%/*.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 **Build Parameters** page (in the latter case, the same value of the property for all build agents will be used).

See also:

Concepts: Build Agent | Build Configuration

Administrator's Guide: Assigning Build Configurations to Specific Build Agents | Configuring Build Agent Startup Properties | Configuring Build Parameters | Configuring Agent Requirements

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](#).

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

Please note that TeamCity assumes control over the directory and can delete subdirectories if they are not used by any of the builds.

See also:

Concepts: Build Agent | Build Checkout Directory

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 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.

See also:

Concepts: First Failure

Authentication Modules

Since version 8.0 you can have several *authentication modules* for TeamCity server simultaneously.

There are two types of *authentication modules* in TeamCity now:

- **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*. 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 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 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.
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](#) and Build Configuration restrictions configured for an agent.

If there are several idle agents that can run a build, TeamCity tries to select the fastest one based on the builds history.

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. Only authorized build agents can run builds. The number of simultaneously authorized agents cannot exceed the number of agent licenses in your license pool. 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.
Enabled/ Disabled	<p>Agents are manually enabled/disabled via the web UI. The TeamCity server only distributes builds to agents that are enabled.</p> <p>Disabled agents can still run builds, when the build is assigned to a special agent (e.g. by Triggering a Custom Build). This feature is generally used to temporarily remove agents from the build grid to investigate agent-specific issues.</p>

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](#)

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

Build Artifact

TeamCity contains an integrated lightweight builds' artifact repository.

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

Upon the build finish, TeamCity searches for artifacts in the build [checkout directory](#) according to the specified artifact 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** column of the build entry on those TeamCity pages that list the builds, or you can find them at the [Artifacts](#) tab of the build results page. You can automate artifacts downloading as described in the [Patterns For Accessing Build Artifacts](#) section.

TeamCity stores 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 artifacts are stored under the `<TeamCity data directory>/system/artifacts` directory. 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.

All artifacts stay on the server and are available for download until they are [cleaned up](#).

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 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 cleanup artifacts deletion unless `".teamcity"` is explicitly specified in the pattern.

You can configure publishing for some of 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)

See also:

Concepts: [Dependent Build](#)

Administrator's Guide: [Configuring General Settings](#) | [Configuring Dependencies](#) | [Patterns For Accessing Build Artifacts](#)

Build Chain

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.

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.

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".

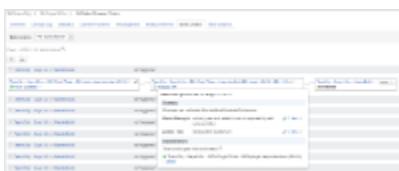
In case when all other parts of the build chain cannot be viewed by the current user, we show a yellow stripe with warning: **You don't have access rights to see its other parts**.

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. You can review build chains on both project and build configuration pages: each of those pages has a Build Chains tab (if snapshot dependencies are configured). The tab displays the list of build chains that contain builds of this project or this build configuration. Note, that build chains are sorted so that the build chain with the last finished build appears at the top of the list.

When expanded, a build chain looks like this:



Here you can see:

- 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.

Click a build in a chain to highlight this build and all its direct dependencies (both upstream and downstream). Since there can be several build chains in a single project, there is an ability to filter them.

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 rerun some build in the chain.

See also:

Concepts: [Dependent Build](#)

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

Build Checkout Directory

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 to the agent incremental patches to update only the files changed since the last build in the given checkout directory.

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

You can specify the checkout directory when configuring **Checkout Settings** on the [Version Control Settings](#) page; however, the default (empty) value is highly recommended. See [Custom checkout directory](#).

If you want to investigate an issue and need to know the directory used by a 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 may 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](#).

Custom checkout directory

In most cases, leaving the checkout directory with the default value (empty in UI) is recommended.

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.

 Note that content of the checkout directory can be deleted by TeamCity under [certain circumstances](#).

Automatic Checkout Directory Cleaning

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).

(See ensuring [free disk space](#) case when the checkout directory can be cleaned automatically.)

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.

The directory cleaning is performed in the background and can be paused by consequent builds.

See also:

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

Build Configuration

A *Build Configuration* is a "type of build" - a set of settings edited in the UI which are used to start a build and group the sequence of the builds in the UI. The settings include VCS settings (from where to checkout sources for the build), what build procedure to run (build steps and environment parameters), triggers (when to start a new build) and others.

A build configurations belongs to a [project](#) and contains builds.

Examples of build configurations are *distribution*, *integration tests*, *prepare release distribution*, "*nightly*" build, *deploy to QA*.

You can explore details of a build configuration on its [home page](#) and modify its settings on the [editing page](#).

In this section:

- [Build Configuration State](#)
- [Build Configuration Status](#)
- [Status Display for Set of Build Configurations](#)

Build Configuration State

A build configuration is characterized by its state which can be *paused* or *active*. By default, when created all configurations are active.

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 [My Changes](#) page. **Since TeamCity 8.1**, the page is called [Changes](#).

To pause or activate a build configuration do one of the following:

- On the Build Configuration Settings page: to pause the configuration, 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.
On the Build Configuration Home Page, click the *Actions button, select Pause triggers in this configuration/Activate triggers in this configuration from the drop-down, add your comment (optional) and click Pause/Activate.

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 executed successfully.
	The last build with this build configuration 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 currently running and failing build, even if the last finished build was successful.
	Indicates that someone has started investigating the problem, or already fixed it. (see Investigating Build Problems).
<i>no icon</i>	There were no finished builds for this configuration, the status is 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 TeamCity web application

To display the display 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 REST API

See also:

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

Build Configuration Template

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.



Starting from version 8.0, build configuration templates support project hierarchy: you can use the templates 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. Now you can associate a build configuration to a template only if the template belongs 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).

How can I create build configuration template?

- **Manually**, like a [regular build configuration](#).
- **Extract** from an existing build configuration: there's **Extract Template** button on the side bar of each build configuration settings. Note, that if you extract template from a build configuration, the original configuration automatically becomes associated with the newly created template.

Associating build configurations with templates

- You can [create new build configurations based on a template](#).
- You can associate any number of existing build configurations with template: there's **Associate with Template** button on the side bar of each build configuration settings.



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

When you *detach build configuration from a template*, all settings from the template will be copied to the build configuration and enabled for editing.

Template which has at least one associated build configuration cannot be deleted.

Starting from version 8.0, you can associate a build configuration to a template only if the template belongs to the current project or one of its parents.

Redefining settings inherited from template

Although a build configuration associated with a template inherits all its settings, it is still possible to redefine a number settings. **Inherited settings cannot be deleted**.

Since TeamCity 8.1, the following settings can now be overridden in a build configuration inherited from a template:

- build number format
- artifact paths
- build options (hanging builds detection, status widget, number of simultaneously running builds)
- VCS checkout mode
- checkout directory
- clean all files before build
- show changes from snapshot dependencies
- execution timeout
- all common build failure conditions, including execution timeout

Prior to Teamcity 8.1, you could redefine settings configured via parameter references or add new items to lists:

Text field settings

When you specify some fixed value in a text field of a template, it is inherited as is and cannot be changed in an associated build configuration. However, in most of the text fields of your template settings (except names (build configuration, parameter, build step), descriptions, agent requirements, typed parameters definitions), you can use a [reference to a build parameter](#) instead of actual value. Thus you can define the actual value of this parameter in each particular associated build configuration separately.

See [below](#) for an example of configuration parameters usage.

Other settings: drop-downs, lists, check boxes, password fields, etc.

These settings are inherited from template as is and cannot be redefined in an associated build configuration.

Collections

A collection of settings, such as parameters, build steps, VCS roots, or build triggers can be extended in an inherited configuration.

- You can add new element to a collection, for example one more VCS root.
- In some cases (parameters, agent requirements, snapshot dependencies) elements can be redefined.
- In some cases you can reorder collection elements in the inherited configuration, for example build steps.

Redefining Template Settings in a Build Configuration

A build configuration created based on a template inherits all the settings defined in the template. You cannot edit these settings for the particular build configuration, whereas modifying them in the template will influence all configurations associated with this template. However, you can bypass this behavior by means of [build parameter references](#).

To introduce a configuration parameter reference use `%ParameterName%` syntax in the template text field. Once introduced, such parameter appears on the **Build Parameters** page of the build configuration template as "undefined".

You can either specify parameter's default value or leave it without any value. You can then define the actual value for the parameter in the build configuration associated with the template.

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 `'+:trunk => .'`. All other settings are equal. It would be useful to have one template to associate with both build configurations, but with means to change checkout rules in each build configuration separately.

To do so, perform the following steps:

1. Extract template from one of those configurations.
2. In template settings navigate to **Version Control Settings**, open Checkout rules dialog and enter there: `%checkout.rules%`
3. For inherited build configuration, open configuration settings page and on the **Build Parameters** page specify the actual value for the `checkout.rules` configuration parameter.
4. For the second build configuration click on the "Associate with template" button and choose the template. Specify appropriate value for `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.

See also:

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

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 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 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.

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

You can download a full build log in a textual form from the Build Results page using the link [!\[\]\(986082884a323475ef59af56b5554821_img.jpg\) Download full build log](#)

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 view errors separately, or 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.

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.

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 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:

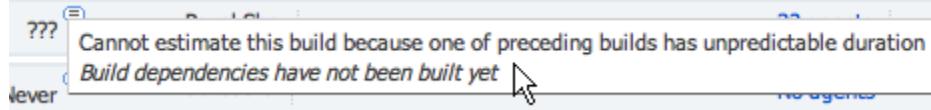
- [Build Queue Tab](#)
 - [Ordering Build Queue](#)
 - [Pausing/Resuming Build Queue](#)
 - [Removing Builds from Queue](#)

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.

Ordering Build Queue

You can [reorder](#) the builds in the queue manually or 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 administrator privileges can [manually Pause/Resume the Build Queue](#).

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.

Removing Builds from Queue

Use the **Remove** button to take the build out of the queue. Next time the build will run according to its triggering settings.

See also:

[Concepts: Build Chain](#)

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

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, the build runner defines how to run a build and report its results. Each build runner has two parts:

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

TeamCity comes bundled with the following runners:

- .NET Process Runner
- Ant
- Command Line — run arbitrary command line
- Duplicates Finder (.NET)
- Duplicates Finder (Java)
- FxCop
- Gradle
- Inspections (IntelliJ IDEA) — a set of IntelliJ IDEA inspections
- Inspections (.NET) — a set of ReSharper inspections
- IntelliJ IDEA Project, and an older version of it: [lpr \(deprecated\)](#)
- Maven
- MSBuild
- MSpec
- MSTest

- NAnt
- NuGet-related runners
- NUnit
- PowerShell
- Rake
- Visual Studio (.sln) — Microsoft Visual Studio 2005/2008/2010 solutions
- Visual Studio 2003 — Microsoft Visual Studio 2003 solutions

Technically, build runners are implemented as plugins.

Build runners are configurable in the **Build Runners** section of the Create/Edit Build Configuration page.

See also:

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

Build State

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

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.

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 did not get to the point to launch the first build step are called "failed to start" and are marked with the icon .

This often is an indication of a configuration error and should usually be addressed by a build engineer rather than a developer if there such roles separation.

For example, VCS repository can be down when build starts, or artifact dependencies can't be resolved, and so on. In case such 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 build configuration status and status of developer changes
- shows "configuration error" stripe for 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 the specific tag
- search for a build with a particular tag
- create an artifact dependency on a build with particular tag

You can assign any number of tags for a single build, for example, "EAP" or "release".

You can tag particular build using **Edit tags** dialog. To open the dialog:

- In a build history table on either **Overview** or **History** tab of the Build Configuration Home Page, click down-arrow button in the **Tags** column for the desired build and then click **Edit** link in the drop-down list.
- In the **Build Actions** drop-down menu on the [Build Results Home Page](#) for the particular build, click **Add build tags** and type the desired tag name in the **Edit tags** dialog.

Clicking the tag you filter out all of the builds in the history and show only builds marked with the tag.

Additionally you can search for builds with particular tags using the [search field](#) on the [Projects](#) page.

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, then you can specify it explicitly using the **Working Directory** field on the settings page of 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](#)

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:

- [My Changes](#) page shows the list of your changes and how they have affected different builds. Since TeamCity 8.1, the page is called **Changes** and has a users selector, so you can see 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 build results page.

Viewing and analyzing changes involves the following possibilities:

- Observing actual changes that are already included in the build, in the **Changes** link on the [Projects](#) page.
- Observing pending changes in the **Pending Changes** tab of the the Build Configuration Home Page.
- Navigating to the related issues in a bug tracking system.
- Navigating to the source code and viewing differences.
- Starting [investigation](#) of a failed build, if your changes have caused a build failure.

See also:

Concepts: [Build Configuration](#)

User's Guide: [Investigating 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> <p> 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.</p>

	running successfully	The change is being integrated into a build that is running successfully.
	successful	The change was integrated into build that finished successfully.
	running and failing	The change is being integrated into a build that is failing.
	failed	The change was integrated into a build that failed.

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

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. The action opens a list of agents/build configurations to clean sources for.

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 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. To do this, just delete `<TeamCity Data Directory>/system/caches` directory.

Automatic Clean Checkout

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.

TeamCity performs automatic clean checkout in the following cases:

- 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
- 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 checkout is performed each time if "Clean all files before build" option is ON in "Version control settings" of the build configuration.

Also, there are cases when agent can delete the build checkout directory e.g. when it [expires](#) or to meet [free disk space requirements](#)

Clean-Up

Clean-up in TeamCity is a feature allowing automatic deletion of data belonging to old builds. Clean-up settings are configured under [Administration | Project-related Settings | Build History Clean-up](#).

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:

- [Clean-up Settings](#)
- [Manual Clean-up Launch](#)
- [Clean-up Rules](#)
- [Clean-up for Dependent Builds](#)
- [Clean-up in Build Configurations with Feature Branches](#)

Clean-up Settings

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

To ensure the server availability, you can 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.

Manual Clean-up Launch

You can run clean-up manually. TeamCity provides the information on the last clean-up duration helping you decide whether to launch the clean-up process at a given moment.

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

Clean-up Rules

A *clean-up* rule defines when and what data to clean. Clean-up rules can be assigned to a project, template or build configuration.

To manage the rules, use the [Administration | Project-related Settings | Build History Clean-up](#) page, **Configure clean-up rules** section.

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 included in/excluded from the clean-up. Use newline-delimited rules following [Ant-like pattern](#). Example: 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 and their associated data (e.g. artifacts) are never deleted by the clean-up procedure.

TeamCity can optionally preserve builds which are used in other builds by [artifact dependency](#).

- **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 dependencies for the builds of the current build configuration. By default, TeamCity does not prevent dependency artifacts cleanup.
- "Do not prevent clean-up" 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.

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 **since TeamCity 8.1**)
- **.NET**: [JetBrains dotCover](#) (bundled), NCover 1.x, 3.x, PartCover, see [Configuring .NET Code Coverage](#)

For integration with other coverage tools, see the related [notes](#).

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
- lpr (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 \(.NET\)](#)

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 (.NET):** 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 \(.NET\)](#)

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.

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.

A build of a build configuration with snapshot dependencies allows reviewing all the dependent 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;
- a queued build does not start until all its snapshot dependency builds are finished. Depending on the snapshot dependencies options and status of the dependency builds, build can start or just marked as failed to start without running;
- 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.

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.



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 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.

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.

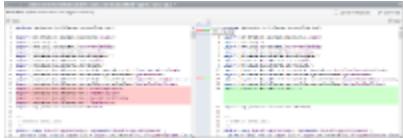
See also:

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

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 TeamCity's UI where the changes lists appear, for example, Projects page, Build Configuration Home Page, or [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:



The window heading displays the file modifications summary:

- the file name alone with its status,
- the changes author,
- the comment for 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](#)

First Failure

For some test failures TeamCity can show "First Failure" build.

This is the build where TeamCity detected the first failure of this test for the same [Build Configuration](#).

I.e. starting from the current build, TeamCity goes back through the build history to find out when this test failed the first time. Builds without this test are skipped, if successful test run was found, searching stops.

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

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.

See also:

Concepts: [Already Fixed In](#)

Guest User

In addition to the available set of roles, TeamCity allows you to turn on guest login, that doesn't require any registration. If enabled, any number of guest users can be logged in to TeamCity simultaneously without affecting each other sessions. Thus, it can be useful for non-committers who just monitor the projects status on the [Projects](#) page.

By default, such user can view projects, their build configurations and download artifacts, and has similar permissions to the [All Projects Viewer](#), although guest user is not a role in the TeamCity terms. As compared with the [Project Viewer](#) role, guest user doesn't have any personal settings, such as [My Changes](#) Page and Profile section (i.e. no way to receive notifications), since it doesn't relate to any particular person.

If guest login is enabled, you can construct an URL to 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`

An administrator can [enable guest login or change guest user username](#) on the **Administration | Global Settings** page.

System administrator can assign additional roles and configure groups for Guest User account via **Guest user settings** section available on the **Users** page.

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 the build with more recent changes. That is, a history build is a build that distracts normal builds flow according to source revisions order.

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

- If you initiate a build on particular changes manually using [Run Custom Build dialog](#).
- If you have a [VCS trigger](#) with quiet period set. During this quiet period other user can start a build with more recent changes. In this case, automatically triggered build will have older source revision when it starts, and will be marked as 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:

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

See also:

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

Identifier

An *ID* is an identifier given by users to TeamCity objects (projects, build configurations, templates and VCS roots, etc.).

The ID is 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 underscore ("_") and should start with a Latin letter.

Using IDs

IDs are used:

- in URLs of the web interface (including [RSS feeds](#), [NuGet feed](#))
- 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.

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](#)

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 [My Changes](#) page (**Since TeamCity 8.1**, the page is called [Changes](#)). 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.

See also:

Concepts: [Pre-tested Commit](#) | [Remote Run](#)

Installing Tools: [IntelliJ Platform Plugin](#) | [Eclipse Plugin](#) | [Visual Studio Addin](#)

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](#).

You can pin or unpin a build in the **Overview** tab of the Build Configuration Home Page, or in the **Build Action** drop-down menu of the [Build Results page](#).

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.

See also diagrams at the http://www.jetbrains.com/teamcity/features/delayed_commit.html.

Submitted code changes first go through testing. If it passes all of the tests, TeamCity can *automatically* submit the changes to version control. From there, it 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 **commit changes if successful** option is selected.

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

For Git and Mercurial the recommended way to use [Branch Remote Run Trigger](#) approach to run personal builds off branches. At this time there is no support for automatic after-the-build merge, see [TW-16054](#).

Matching changes and build configurations

To submit changed files to pre-tested commit or remote run, 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 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, option to initiate the personal build will not be available.

General Flow of a pre-tested commit

- Developer uses Remote Run dialog of TeamCity IDE plugin to select files to be sent to TeamCity.
- Based on the selected files a list of applicable build configurations is displayed. Developer selects the build configurations to test the change against and sets options for pre-tested commit.
- 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 developer's machine. When sent, the change appears on developer's My Changes. 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 build starts, it checks out the latest sources just like normal build and then applies developer's personal changes sent from 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
- TeamCity IDE plugin pings TeamCity server to check if all the selected build configurations have personal builds ready. If a build fails, a notification is displayed in IDE. and the process ends.
- If all the personal builds finish successfully, 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 pre-tested commit initiating), then restores the file contents from the saved "patch", performs the version control commit (reports an error if there was an error like VCS conflict) and restores the just backed up files to bring the working copy in the last seen state. A pre-tested commit in 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](#)

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.

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](#)

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

- [VCS root](#)
- [Build configuration template](#)
- [Shared Resources](#)
- [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 project's association with an agent pool is not propagated to the 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](#)

Remote Debug

The *Remote Debug*, available **since TeamCity 8.0**, 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.



Remote debug sessions can be launched right from IntelliJ IDEA for the builds based on the IntelliJ IDEA Project build steps.

Currently, the following IntelliJ IDEA run configurations are supported:

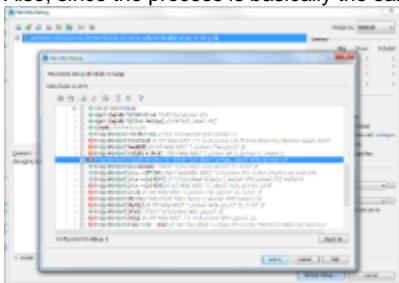
- Java application run configuration
- JUnit run configuration
- TestNG run configuration

Prerequisites:

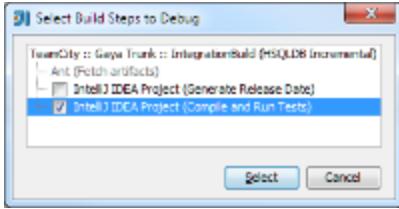
1. an IntelliJ IDEA run configuration on the local developer machine with the [TeamCity plugin for IntelliJ IDEA](#) installed,
2. a build configuration on the TeamCity Server with the [IntelliJ IDEA Project](#) runner as one of the [build steps](#)
3. a remote [TeamCity agent](#) to run this build available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

Debugging Tests Remotely

1. 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).
2. 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.



3. 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.



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

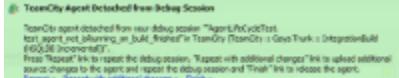


The debug tool window works in the listening mode, i.e. it waits for the agent to connect to it. Once the agent connects, the Java process on the agent is paused and the Agent Attached notification appears in the IDE:



5. 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.

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](#)

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.

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:

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 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 and agent manager's permissions.
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	Agent Manager is a person responsible for customizing and managing the Build Agents ; he/she can change the run configuration policy and enable/disable build agents.
Project Viewer	Project Viewer has read-only access to projects and can only <i>view</i> the project, its parent and subprojects. Project Viewer does not have permissions to view agent details.

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.

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](#)

Super User

The **Super user** login allows you to access the server UI with System Administrator permissions, if you do not remember credentials or need to fix authentication-related settings.

It enables you to log in using an authentication token automatically generated on a server start; each server restart generates a new token. The token is printed in the server console and `teamcity-server.log` (search for "Super user authentication token" text).

To log in as a super user, use an empty username and the authentication token as the password on the login page. On login attempt without username specified, the token is printed into the log.

Instead of using 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 "`teamcity.disable.super.user=true`" [internal property](#).

A super user is not a usual TeamCity user and does not have any personal settings, such as **My Changes** page (**since TeamCity 8.1** the page is called **Changes**) 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 be logged in to TeamCity simultaneously without affecting each other sessions.

See also:

Concepts: [Guest User](#)

the

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.

The location of the directory is set using `TEAMCITY_DATA_PATH` environment variable. If the variable is not set, the TeamCity Data Directory is assumed to be located in the user's home directory (e.g. it is `$HOME/.BuildServer` under Linux and `C:\Users\<user_name>\.BuildServer` under Windows).

The currently used data directory location can be seen on the [Administration | Global Settings](#) page for a running TeamCity server instance. **Since TeamCity 8.1**, you can use the [Browse](#) link to view the TeamCity Data Directory. Clicking the 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).

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.

TeamCity Windows installer configures the TeamCity data directory during installation. The default path suggested for the directory is: `%ALLUSERSPROFILE%\JetBrains\TeamCity`.

On this page:

- Specify Location of the TeamCity Data Directory
 - 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

Specify Location of the TeamCity Data Directory

Set `TEAMCITY_DATA_PATH` environment variable (either system-wide or for the user under whom TeamCity will be started). You might need to restart the computer after that for the changes to take effect.

You will need to restart the TeamCity server after making changes to the setting.

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

Recommendations as to choosing Data Directory Location

Note that 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.

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. Still under rare circumstances the data stored in the directory can be corrupted and be partially lost.

We do not recommend to place the entire TeamCity data directory to a remote/network folder. If a single local disk cannot store all the data, consider placing the data directory on a local disk and mapping `.BuildServer/system/artifacts` to a larger disk with the help of OS-provided filesystem links.

Related feature request: [TW-15251](#).

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
 - `_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>` - directory containing all the settings of a project with id "`<projectId>`" (including build configuration settings and excluding subproject settings). New directories can be created provided they have mandatory nested files. Directory `_Root` 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-settings.xml` file in the directory
 - `vcsRoots` — a directory which contains project's VCS roots settings in the files `_<VcsRootID>.xml`
 - `project-config.xml` — main project configuration, contains configurations of a project's [Build Configurations](#)
 - `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
 - `internal.properties` — file for specifying various [internal TeamCity properties](#). Is not present by default and should be created if necessary
 - `auth-config.xml` — 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 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 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* — a directory where the builds' artifacts are stored. The format of 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 corresponding build's artifacts.
 - *.teamcity* directory in each build's directory stores build's **hidden artifacts**. The files can be deleted manually, if necessary, but build will lack corresponding feature backed by the files (like displaying/using finished build parameters, coverage reports, etc.)
 - *messages* — a directory where **build logs** are stored in internal format. Build logs store build output, compilation errors, test output and test failure details. Build with **internal id** "xxxx" stores its log in CHyy/xxxx.* file, where "yy" are the last two digits of xxxx. The files can be removed manually, if necessary, but corresponding builds will drop build log and failure details (as well as test failure 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.
 - *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 more safe 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 related [comment](#) in our issue tracker on build configurations move 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. 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. Each user account (or the 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.

[Guest User](#) does not belong to the All Users group.



The "default user" roles cannot be edited. Please use defaults in "All User" group.

See also:

Concepts: User Account | Role and Permission

Administrator's Guide: Managing User Accounts, Groups and Permissions

VCS root

VCS Roots in TeamCity

A VCS root is a set of [VCS 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, TeamCity will issue a warning that the changes to the VCS root could potentially affect more than one project. The user is then prompted to either save the changes and apply them to all the projects using the VCS root, or to make a copy of the VCS root to be used by either a specific build configuration or project.

A VCS root can also be created at the level of a build configuration, using the **Version Control Settings** page.

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

Wildcards

TeamCity supports wildcards in different configuration options.

Ant-like wildcard 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 -file1.txt -file2.txt -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>

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
- Testing Frameworks
- Version Control Systems
 - Checkout on Agent
 - Labeling Build Sources
 - Remote Run on Branch
 - Feature Branches
 - VCS Systems Supported via Third Party Plugins
- 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.

TeamCity server is a web application that runs within a capable J2EE servlet container.

Requirements:

- Java (JRE) 1.6+ (Java 1.7 is included in the Windows .exe distribution). TeamCity is tested with the latest Oracle Java 1.7. Both 32-bit and 64-bit Java versions can be [used](#). Note that OpenJDK is NOT supported (while it can also work).
- (for .war distribution) J2EE Servlet (2.5+) container, JSP 2.0+ container based on Apache Jasper. TeamCity is tested under Tomcat 7 which is the recommended server. Tomcat 7 is already included in Windows .exe and .tar.gz distributions. TeamCity is reported to work with Jetty and Tomcat 6.x-7.x.

It is highly recommended to use the .tar.gz distribution (which has Tomcat bundled) and not to customize Tomcat settings unless absolutely necessary.

If you still want to use the .war distribution, note that

- it is recommended to use Tomcat 6.0.35+; earlier versions of Tomcat have some issues which can cause a deadlock in TeamCity on the start-up.
- TeamCity may not work properly if the [Apache Portable Runtime](#) is enabled in Tomcat.



TeamCity with the native MSSQL external database driver is not compatible with Oracle Java 1.6.0_29, due to a bug in Java itself. You can use earlier or later versions of Oracle Java.

Generally, all the recent versions of Windows, Linux and Mac OS X are supported.

The TeamCity server is tested under the following operating systems:

- Linux (Ubuntu, Debian)
- MacOS X
- Windows XP
- Windows Vista/Windows Vista 64
- Windows 7/7x64
- Windows Server 2008
- Windows 8
 - under the Tomcat 7 web application server.

Reportedly works on:

- Windows Server 2008 R2
- Windows Server 2012
- Solaris
- FreeBSD
- IBM z/OS
- HP-UX

Build Agents

The TeamCity Agent is a standalone Java application.

Requirements:

- Java (JRE) 1.6+ (Java 1.7 already included in the Windows .exe distribution). TeamCity is tested with Oracle Java. Both 32-bit and 64-bit Java versions can be used. Note that OpenJDK is NOT supported (while it can also work).

TeamCity agent is tested under the following operating systems:

- Linux
- MacOS X
- Windows XP/XP x64/Vista/Vista x64
- Windows 7/7x64 (recommended choice for Windows installations)
- Windows 8
- Windows Server 2003/2008

Reportedly works on:

- Windows Server 2012
- Windows 2000 (interactive mode only)
- Solaris
- FreeBSD
- IBM z/OS
- HP-UX

Stop Build Functionality

Build stopping is supported on:

- Windows 2000/XP/XP x64/Vista/Vista x64/7/7x64/8
- Linux on x86, x64, PPC and PPC64 processors
- Mac OS X on Intel and PPC processors
- Solaris 10 on x86, x64 processors

Windows Tray Notifier

Windows 2000/XP/Vista/Vista x64/7/7x64/8 with one of the supported versions of Internet Explorer.

Web Browsers

The TeamCity Web Interface is 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:

- Microsoft Internet Explorer 7+ (without the compatibility mode)
- Mozilla Firefox 3.6+
- Opera 9.5+
- Safari 3+ under Mac/Windows
- Google Chrome

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 1.6-1.9 (TeamCity comes bundled with Ant 1.8.2, **since TeamCity 8.1**, Ant 1.8.4)
- 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 and Maven 3.0.5. **Since TeamCity 8.1**, Maven 3.1)
- IntelliJ IDEA Project runner
- Gradle (requires Gradle 0.9-rc-1 or higher)
- Java Inspections and Java Duplicates based on IntelliJ IDEA (requires Java 1.6+)

Supported .Net platform build runners:

- MSBuild (requires .Net Framework or Mono installed on the build agent. **Since TeamCity 8.0.5** Microsoft Build Tools 2013 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, 2005, 2008, 2010, 2012, and **since TeamCity 8.0.5**, 2013) (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. **Since TeamCity 8.0.6** Microsoft Visual Studio 2013 solutions are supported.
- Inspections for .NET based on ReSharper Command Line Tools. Requires .Net Framework 4.0+ installed on the build agent. **Since TeamCity 8.0.6** Microsoft Visual Studio 2013 solutions are supported.
- NuGet runners (supported only for agents running Windows OS. Require NuGet.exe Command Line tool installed on the agents). Supported NuGet versions are 1.4+.

Other runners:

- Rake
- Command Line Runner for running any build process using a shell script
- PowerShell
- .NET Process Runner for running any .NET application (requires .NET installed on the build agent)
- Xcode 3 and 4. **Since TeamCity 8.1** Xcode 5 is 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, 2.6.0 (dedicated build runner)
- TestNG 5.3+
- MSTest 8.x, 9.x, 10.x, 11.x, and (**since TeamCity 8.1**) 12.x. (the dedicated build runner; requires appropriate Microsoft Visual Studio edition installed on the build agent)
- MSpec (requires MSpec installed on the build agent)

Version Control Systems

- Subversion (server versions 1.4-1.7 and higher as long as the protocol is backward compatible). Check Subversion 1.8 compatibility with different TeamCity versions.
- Perforce (requires a Perforce client installed on the TeamCity server)
- Git (requires a Git client installed on the TeamCity server).
- Mercurial (requires the Mercurial "hg" client v1.5.2+ installed on the server)
- Team Foundation Server 2005, 2008, 2010, 2012, and (**since TeamCity 8.0.6**) 2013. Requires Team Explorer installed on the TeamCity server.
- CVS (requires a CVS client installed on the TeamCity server)
- IBM Rational ClearCase, Base and UCM modes (requires the ClearCase client installed and configured on the TeamCity server)
- SourceGear Vault 6 and 7 (requires the Vault command line client libraries installed on the TeamCity server)
- Microsoft Visual SourceSafe 6 and 2005 (requires a SourceSafe client installed on the TeamCity server, available only on Windows platforms)
- Borland StarTeam 6 and up (the StarTeam client application must be installed on the TeamCity server)

Checkout on Agent

The requirements noted are additional to those listed above.

- Subversion (working copies in the Subversion 1.4-1.7 format are supported)
- CVS (a CVS client must be installed on the TeamCity agent machine)
- Git (git v.1.6.4+ must be installed on the agent)
- Mercurial (the Mercurial "hg" client v1.5.2+ must be installed on the TeamCity agent machine)
- Team Foundation Server 2005, 2008 and 2010 (requires Team Explorer to be installed on the build agent, available only on Windows platforms)
- Perforce (a Perforce client must be installed on the TeamCity agent machine)
- IBM Rational ClearCase (the ClearCase client must be installed on the TeamCity agent machine)

Labeling Build Sources

- Subversion
- CVS
- Git
- Mercurial
- Team Foundation Server
- Perforce
- Borland StarTeam
- ClearCase
- SourceGear Vault

Remote Run on Branch

- Git
- Mercurial

Feature Branches

- Git
- Mercurial

VCS Systems Supported via Third Party Plugins

- AccuRev
- Bazaar
- PlasticSCM

Issue Tracker Integration

- JetBrains YouTrack 1.0 and later (tested with the latest version).
- Atlassian Jira 4.4 and later (all major features also worked for version 4.2).
- Bugzilla 3.0 and later (tested with 3.4). 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 3.4.2-3.8 and 4.2-4.3 are supported. Eclipse must be run under JDK 1.5+
- **IntelliJ Platform Plugin:** compatible with IntelliJ IDEA 9.x - 12.x (Ultimate and Community editions) and other IDEs based on the same version of the platform, including JetBrains RubyMine 2.0+, JetBrains PyCharm 1.0+, JetBrains PhpStorm/WebStorm 1.0+, AppCode 1.0+
- **Microsoft Visual Studio** 2005, 2008, 2010, 2012, and (**since TeamCity 8.0.6**) 2013. 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"> Subversion 1.4-1.7 (with Subclipse and Subversive Eclipse integration plugins). Subversion 1.8 (Since TeamCity 8.1) via Subversive/Subclipse/SvnKit. Perforce (P4WSAD 2008.1 - 2010.1, P4Eclipse 2010.1 - 2012.3) ClearCase (the client software is required) CVS Git (the EGit 0.6 - 3.20 Eclipse integration plugin) <p>see also</p>
IntelliJ IDEA Platform	<ul style="list-style-type: none"> ClearCase Git (remote run only) Perforce StarTeam Subversion Visual SourceSafe
Microsoft Visual Studio	<ul style="list-style-type: none"> Subversion 1.4-1.8 (the command-line client is required) Team Foundation Server 2005 and later. Team Foundation Server 2013 is supported since TeamCity 8.0.6. Installed Team Explorer is required. Perforce 2008.2 and later (the command-line client is required)

Code Coverage

IDE	Supported Coverage Tool
Eclipse	IDEA and EMMA code coverage
IntelliJ IDEA Platform	IDEA, EMMA and JaCoCo code coverage
Microsoft Visual Studio	JetBrains dotCover coverage. Requires JetBrains dotCover installed in Microsoft Visual Studio

Supported Databases

See more at [Setting up an External Database](#)

- HSQLDB 1.8/2.x (internal, used by default)
- MySQL 5.0.33+, 5.1.49+, 5.5+ (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 and 2012 (including Express editions); SSL connections support might require [these updates](#).
- PostgreSQL 8+
- Oracle 10g+ (TeamCity is tested with [driver](#) version 10.2.0.1.0)

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

TeamCity installation package is identical for both Professional and Enterprise Editions and is available for download at <http://www.jetbrains.com/teamcity/download/> page.

Following packages are available:

Target	Download	Note
Windows	TeamCity<version number>.exe	Executable Windows installer bundled with Tomcat and Java 1.7 JRE.
Linux, MacOS X	TeamCity<version number>.tar.gz	Package bundled with Tomcat servlet container for Linux, MacOS X or manual installation.
J2EE container	TeamCity<version number>.war	Package for installation into an existing J2EE container.

Install TeamCity

Following the installation instructions:

- Installing TeamCity via Windows installation package
- Installing TeamCity bundled with Tomcat servlet container (Linux, Mac OS X, Windows)
- Installing TeamCity into Existing J2EE Container

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 the build agent 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 .war) based on the details below
- [Download the distribution](#)
- Review [software requirements and hardware requirements notes](#)
- 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 \(Linux, Mac OS X, Windows\)](#)
 - [Starting TeamCity server](#)
 - [Installing TeamCity into Existing J2EE Container](#)
 - [Autostart TeamCity server on Mac OS X](#)
 - [Using another Version of Tomcat](#)
- [Installation Configuration](#)
 - [Troubleshooting TeamCity Installation](#)
 - [Changing Server Port](#)
 - [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](#)

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;
- [.war distribution](#) - for experienced users who want to run TeamCity in a separately installed Web application server.

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
- may provide more convenience features in the future
- come bundled with a build agent distribution which allows for easy TeamCity server evaluation with one agent
- come bundled with devPackage for TeamCity plugin development.

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 X. If you need more build agents, refer to the [Installing Additional Build Agents](#) section.



By default, TeamCity uses an HSQLDB database that does not require configuring. This database works fine for testing and evaluating the system.

For production purposes, using a standalone external database is recommended.

- [Setting up an External Database](#)
- [Migrating to an External Database](#)

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.



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:

- write permissions for the [TeamCity Data Directory](#),
- write permissions for the TeamCity Home directory, 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, Mac OS X, 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 `ClassNotFoundException` 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. The Oracle Java 1.7 is recommended.

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 into an existing web server (.war distribution), start the server according to its documentation. Make sure you configure TeamCity-specific logging-related properties.

If TeamCity is installed using the .exe or .tar.gz distributions, the TeamCity server can be started and stopped by the scripts provided in the `<TeamCity home>/bin` directory.

To start/stop the TeamCity server and one default agent at the same time, use the `runAll` script.

To start/stop only the TeamCity server, use the `teamcity-server` script and pass the required parameters. Start the script without parameters to see the usage instructions.

For example:

- Use `runAll.bat start` to start the server and the default agent
- Use `runAll.bat stop` to stop the server and the default agent

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](#).



Headless mode for TeamCity

If you are running TeamCity on a server that is not running a windowing system, for example, the console mode under Linux, you may encounter the following error when hitting the Statistics page:

```
javax.el.EELException: Error reading 'graphInfo' on type  
jetbrains.buildServer.serverSide.statistics.graph.BuildGraphBean
```

You can resolve this problem by adding `-Djava.awt.headless=true` the server JVM option.

Installing TeamCity into Existing J2EE Container

If TeamCity is the only application in the server, it is recommended to [use](#) the TeamCity distribution bundled with Tomcat web server.

1. 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).
2. 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.
3. 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.
4. 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`.
5. 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.
6. Configure the appropriate [TeamCity Data Directory](#) to be used by TeamCity.
7. Check/configure the TeamCity [logging](#) properties by specifying the `log4j.configuration` and `teamcity_logs` internal properties.
8. 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 [Apache Portable Runtime](#) feature of this container is disabled (actually it is disabled by default). Unfortunately because of bugs in Apache Portable Runtime, TeamCity may not work properly in this case.

Autostart TeamCity server on Mac OS X

Starting up TeamCity server on Mac is quite similar to starting Tomcat on Mac.

- 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
- 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/teamcity-server.sh</string>
<string>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>

```

- 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 browser).
- 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>

```

- That's it. Now TeamCity should autostart when the machine starts.

Using another Version of Tomcat

If you want to use another version of Tomcat web server instead of the bundled one, you have the choices of whether to use the .war TeamCity distribution 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
 - delete the default Tomcat conf directory and replace it with the one provided by TeamCity
 - delete the default Tomcat webapps/ROOT directory and replace it with the one provided by TeamCity

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 [more 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 HTTP/1.1 connector (here the port number is 8111):

```
<Connector port="8111" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443"
           enableLookup="false"
           useBodyEncodingForURI="true"
        />
```

To apply changes, restart the server.

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](#)

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.

TeamCity (both server and agent) requires JRE 1.6 (or later) to operate. Using latest Oracle Server JRE 1.7 is recommended ([download page](#)). It is recommended to use 32 bit installation unless you need to [dedicate more memory](#) to TeamCity server. Please check [64 bit Java notes](#) before upgrade.

For TeamCity agent Java requirements, check [Setting up and Running Additional Build Agents#Configuring Java](#).

The necessary steps to update Java installation depend on the distribution used.

- if your TeamCity installation has bundled JRE (there is `<TeamCity home>\jre` directory), update it by installing newer JRE per installation instructions and copying the content of the resulting directory to replace the content of the existing `<TeamCity home>\jre` directory). If you also run TeamCity agent from `<TeamCity home>\buildAgent` directory, use JVM installation instead of 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 TeamCity server (setting global OS environment variables and system restart is recommended). The variables should point to the home directory of installed JRE or JVM (Java SDK) respectively.
- if you use .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 32 and 64 bit JVM.

It is recommended to use 32 bit JVM unless you need to dedicate more than 1.3Gb of memory (via `-Xmx` JVM option) to the TeamCity process.

If you choose to use x64 JVM please note that the memory usage is almost doubled when switching from 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](#).

The steps to update to 64 bit Java are:

- update Java to be used by the server
- set JVM memory options. Recommended options for 64 bit JVM are `-Xmx4g -XX:MaxPermSize=270m`

Setting Up Memory settings for TeamCity Server

As a JVM application, TeamCity only utilizes memory devoted to the JVM. Memory used by JVM usually consists of: heap (configured via `-Xmx`), permgen (configured via `-XX:MaxPermSize`), internal JVM (usually tens of Mb), and OS-dependent memory features like memory-mapped files. TeamCity mostly depends on the heap and permgen memory and these settings can be configured for the TeamCity application manually by passing `-Xmx` (heap space) and `-XX:MaxPermSize` (PermGen space) options to the JVM running the TeamCity server.

- For initial use of TeamCity for production purposes (assuming 32 bit JVM), the minimum recommended settings are: `-Xmx750m -XX:MaxPermSize=270m`. If slowness or OutOfMemory error occurs, please increase the settings to `-Xmx1200m -XX:MaxPermSize=270m`.
- The maximum settings that you might ever need are (x64 JVM should be used): `-Xmx4g -XX:MaxPermSize=270m`. These settings will be suitable for an installation with more than a hundred of agents and thousands of build configurations and without custom plugins installed.
- If you run TeamCity via the `runAll` or `teamcity-server` scripts or via a Windows service installed, the default settings used are: 512 Mb for the heap and 150 Mb for the PermGen.

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.

Tips:

- 32 bit JVM can use up to 1.3Gb heap memory. If more memory is necessary, 64 bit JVM should be used assigning not less than 2.5Gb. 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 64 bit JVM should be assigned twice as much memory as 32 bit for the same application. If you switch to 64 bit JVM please make sure you adjust the memory settings (both `-Xmx` and `-XX:MaxPermSize`) accordingly. It does not make sense to switch to 64 bit if you dedicate less than double amount of memory to the application.

The recommended approach is to start with initial settings and monitor for the percentage of used memory (see also [TW-13452](#)) at 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 memory values 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

See also:

Installation and Upgrade: Setting up and Running Additional Build Agents

Setting up and Running Additional Build Agents

This page covers:

- [Installing Additional Build Agents](#)

- Necessary OS and environment permissions
- Server-Agent Data Transfers
- Installing Procedure
- Installing via Java Web Start
- Installing via a MS Windows installer
- Installing via ZIP File
- Installing via Agent Push
- Starting the Build Agent
- Stopping the Build Agent
- Automatic Agent Start under Windows
- Using LaunchDaemons Startup Files on MacOsx
- 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.



- If you install TeamCity bundled with a Tomcat servlet container, or opt to install an agent for Windows, both the server and one build agent are installed. This is not a recommended setup for production purposes, 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.
- For production installations it is recommended to adjust the [Agent's JVM parameters](#) to include the `-server` option.

Installing Additional Build Agents

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 under which the Agent is running should have the correct privileges, as described below.

Network

- Agent should be able to open HTTP connections to the server (to the same URL as server web UI)
- Server should be able to open HTTP connections to the agent. The port is determined by "ownPort" property of `buildAgent.properties` file (9090 by default) and the following hosts are tried:
 - host specified in the "ownAddress" property of `buildAgent.properties` file (if any)
 - source host of the request received by the server when agent establishes connection to the server
 - address of the network interfaces on the agent machine

If the agent is behind NAT and cannot be accessed by any of addresses of agent machine network interfaces, please specify `ownAddress` in the agent config.

Common

- 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 TeamCity UI) and accept inbound HTTP connections from the server to the port specified as "ownPort" property in "`<TeamCity agent home>/conf/buildAgent.properties`" file (9090 by default). Please ensure that any firewalls installed on agent, server machine or in the network and network configuration comply with these requirements.
- have full permissions (read/write/delete) to the following directories: `<agent home>` (necessary for automatic agent upgrade), `<agent work>`, and `<agent temp>`.
- 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 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)

For granting necessary permissions for unprivileged users, see Microsoft [SubInACL](#) utility. For example, to grant Start/Stop rights you might need to execute `subinacl.exe /service browser /grant=<login name>=PTO` command.

Linux

- user should be able to run `shutdown` command (for agent machine reboot functionality and machine shutdown functionality when

running in Amazon EC2)

Build-related Permissions

The build process is launched by TeamCity agent and thus shares the environment and is executed under the same OS user that TeamCity agent runs under. Please ensure that TeamCity agent is configured accordingly.

See [Known Issues](#) for related Windows Service Limitations.

Server-Agent Data Transfers



Please be sure to read through this section if you plan to deploy agent and server into non-secure network environments.

During TeamCity operations, both server establishes connections to the agents and agents establish connections to the server.

Please note that by default, these connections are not secured and thus are exposing possibly sensitive data to any third party that can 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 can in theory trick agent into executing arbitrary command and perform other actions with a security impact.

It is recommended to deploy agents and the server into a secure environment and use plain HTTP for agents-to-server communications as this reduces transfer overhead.

It is possible to setup a server to be available via HTTPS protocol, so all the data traveling through the connections established from an agent to the server (incl. download of build's sources, artifacts of builds, build progress messages and build log) can be secured. See [Using HTTPS to access TeamCity server](#) for configuration details.

However, the data that is transferred via the connections established by the server to agents (build configuration settings, i.e., all the settings configured on the web UI including VCS root data) is passed via unsecured HTTP connection. For the time being TeamCity does not provide internal means to secure this data transfers (see/vote for [TW-5815](#)). If you want to secure the data you need to establish appropriate network security configurations like VPN connections between agent and server machines.

Installing Procedure

You can install build agent using any of the following installation options available:

- Via Java Web Start
- Using MS Windows installer
- Download zip file and install manually

After installation, please configure the agent specifying its name and address of TeamCity server in its `conf/buildAgent.properties` file. Then [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 `Unauthorized agents` tab under Agents, where administrators can authorize it. Please note that the tab is only visible for administrators/users with appropriate permission.



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 Java Web Start

1. Make sure JDK 1.6+ is properly installed on the computer.
2. On the agent computer, set up the `JAVA_HOME` environment variable to point to the JDK 1.6+ installation directory.
3. Navigate to the **Agents** tab in the TeamCity web UI.
4. Click the "Install Build Agents" link and then click "Via Java Web Start".
5. Follow the instructions.



You can install the build agent Windows service during the installation process or [manually](#).

Installing via a MS Windows installer

1. Navigate to the **Agents** tab in the TeamCity web UI.
2. Click the "Install Build Agents" link and then click **MS Windows Installer** link to download the installer.

3. Run the `agentInstaller.exe` Windows Installer and follow the installation instructions.



Please ensure the user under whom the agent service is running has appropriate permissions

Installing via ZIP File

1. In TeamCity Web UI, navigate to the **Agents** tab
2. Click the **Install Build Agents** link and then click **download zip file**
3. Unzip the downloaded file into the desired directory.
4. Make sure that you have a JDK or JRE 1.6+ installed (You will need JDK (not JRE) for some build runners like IntelliJ IDEA, Java Inspections and Duplicates). Please ensure that the `JRE_HOME` or `JAVA_HOME` environment variables are set (pointing to the installed JRE or JDK directory respectively) for the shell in which the agent will be started.
5. Navigate to the `<installation path>\conf` directory, locate the file called `buildAgent.dist.properties` and rename it to `buildAgent.properties`.
6. 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 more details on agent configuration
7. 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 manual agent startup.

Installing via Agent Push

TeamCity provides functionality that allows to install a build agent to a remote host. Currently supported combinations of server host platform and targets for build agents are:

- from Unix based TeamCity server build agents can be installed to Unix hosts only (via SSH).
- from Windows based TeamCity server build agents can be installed to Unix (via SSH) or Windows (via `psexec`) hosts.



SSH note

Make sure "Password" or "Public key" authentication is enabled on the target host according to preferred authentication method.

There are several requirements for the remote host that should be met:

Platform	Prerequisites
Linux	Installed JDK(JRE) 1.6+ required. JVM should be reachable with <code>JAVA_HOME</code> (<code>JRE_HOME</code>) global environment variable or be in the global path (i.e. not in user's <code>.bashrc</code> file, etc.) Also required 'unzip' utility and either 'wget' or 'curl'.
Windows	<ul style="list-style-type: none">• Installed JDK/JRE 1.6+ is required.• <code>Sysinternals psexec.exe</code> on TeamCity server required. It has to be accessible in paths. You can install it at Administration Tools page. Note, that PsExec applies additional requirements to remote Windows host (for example, administrative share on remote host must be accessible). Read more about PsExec.

Installation Procedure

1. In the TeamCity Server web UI navigate to **Agents | Agent Push** tab, and click **Install Agent....**



Note, that if you are going to use same settings for several target hosts, you can create a preset with these settings, and use it next time when installing an agent to another remote host.

2. In the **Install agent** dialog, if you don't yet have any presets saved, select "Use custom settings", specify target host platform and configure corresponding settings.
3. You may need to download `Sysinternals psexec.exe`, in which case you will see corresponding warning and a link to [Administration | Tools](#) where you can download it.



You can use Agent Push presets in [Amazon EC2 Cloud profile](#) settings to automatically install build agent to started cloud instance.

Starting the Build Agent

To start the agent manually, run the following script:

- for Windows: <installation path>\bin\agent.bat start
- for Linux and MacOS X: <installation path>\bin\agent.sh start



If you're running build agent on MacOS X and you're going to run Inspection builds, you may need to use the **StartupItemContext** utility:

```
sudo /usr/libexec/StartupItemContext agent.sh start
```

To configure agent to be **started automatically**, see corresponding sections:

[Windows](#)

[Mac OS X](#)

Linux: configure daemon process with `agent.sh start` command to start it and `agent.sh stop` command to stop it.

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 current build finished.

Use `stop force` to request 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).

Automatic Agent Start under Windows

To run agent automatically on machine boot under Windows you can either setup agent to be run as Windows service or use another way of automatic process start.

Using Windows service approach is the easiest way, but Windows applies [some constraints](#) to the processes run in this way.

TeamCity agent can work OK under Windows service (provided all the [requirements](#) are met), but is often not the case for the build processes that you will configure to be run on the agent.

That is why it is advised to run TeamCity agent as use Windows service only if all the build scripts support this.

Otherwise, it is advised to use alternative ways to start TeamCity agent automatically.

One of the ways is to configure automatic user logon on Windows start and then configure TeamCity agent start (via `agent.bat start`) on user logon.

Build Agent as a Windows Service

In Windows, you may want to use the build agent Windows service to allow the build agent to run without any user logged on.

If you use 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, Windows service is started under the **SYSTEM** account. To change it, use the standard Windows Services applet (Control Panel\Administrative Tools|Services) and change the user for the TeamCity Build Agent service.

The following instruction can be used to install the service manually. This procedure should also be performed to install second and following agents on the same machine as Windows services

To install the service:

1. Make sure there is no **TeamCity Build Agent Service <build number>** service already installed, if installed, uninstall the agent.

- Check `wrapper.java.command` property in `<agent home>\launcher\conf\wrapper.conf` file to contain valid path to Java executable in the JDK installation directory. You can use `wrapper.java.command=../jre/bin/java` for agent installed with Windows distribution. Make sure to specify the path of the `java.exe` file without any quotes.
- Run the `<agent home>/bin/service.install.bat` file.

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 Windows `net.exe` utility to manage the service once it is installed.
For example (assuming the default service name):

```
net start TCBuildAgent
```

The file `<agent home>\launcher\conf\wrapper.conf` can also be used to alter agent JVM parameters.

User account that is used to run build agent service should have enough rights to start/stop agent service.



A method for assigning rights to manage services is to use the `SUBINACL` utility from the Windows 2000 Resource Kit. The syntax for this is:
`SUBINACL /SERVICE \\MachineName\ServiceName /GRANT=[DomainName]UserName[=Access]`
See <http://support.microsoft.com/default.aspx?scid=kb;en-us;288129>

Using LaunchDaemons Startup Files on MacOsx

For MacOsx, TeamCity provides ability to load a build agent automatically at the system startup using LaunchDaemons `plist` file.

To use LaunchDaemons `plist` file:

- Install build agent on Mac either via `buildAgent.zip` or via Java web-start
- Prepare `conf/buildAgent.properties` file
- Fix launcher permissions, if needed: `chmod +x buildAgent/launcher/bin/*`
- Load build agent to LaunchDaemon via command:

```
sh buildAgent/bin/mac.launchd.sh load
```



You have to wait several minutes for the build agent to auto-upgrade from the TeamCity server.

- To start the build agent on reboot, you have to copy `buildAgent/bin/jetbrains.teamcity.BuildAgent.plist` file to the `/Library/LaunchDaemons` directory. And if you don't want to run your agent as root (and you probably don't), you have to edit `/Library/LaunchDaemons/jetbrains.teamcity.BuildAgent.plist` file and add section like

```
<key>UserName</key>
<string>your_user</string>
```

Also, make sure that all files under `buildAgent` directory are owned by `your_user` to ensure proper agent upgrade process.

- To stop build agent, run the following command:

```
sh buildAgent/bin/mac.launchd.sh unload
```

If you need to configure TeamCity agent environment you can change `<TeamCity Agent Home>/launcher/conf/wrapper.conf` ([JSW configuration](#)). For example, to make the agent see Mono installed using MacPorts on Mac OS X agent you will need to add the following line:

```
set.PATH=/opt/local/bin%WRAPPER_PATH_SEPARATOR%%PATH%
```

Configuring Java

TeamCity Agent is a Java application and it requires JDK version 1.6 or later to work. Oracle Java SE JDK 1.7 32 bit is recommended ([download page](#)).

(Windows) .exe TeamCity distribution comes with appropriate Java bundled. If you run previous version of TeamCity agent you might need to repeat agent installation to update the JVM.

Using x32 bit JDK is recommended. If you do not have Java builds, you may install JRE instead of JDK.

Using of x64 bit Java is possible, but you might need to double -Xmx and -XX:MaxPermSize memory values for the main agent process (see [Configuring Build Agent Startup Properties](#) and alike [section](#) for the server).

For .zip agent installation you need to install appropriate Java version (make it available via PATH) or available in one of the following places:

- <Agent home>/jre directory
- in the directory pointed to by `JAVA_HOME` or `JRE_HOME` environment variables.

Upgrading Java on Agents

If a build agent uses a Java version older than it is required by agent (Java 1.6 currently), you will see the corresponding warning at the agent's page in web UI. This may happen when upgrading to a newer TeamCity version, which doesn't support an old Java version anymore. To update Java on agents you can do one of the following:

- If appropriate Java version is detected on the agent, the agent page provides an action to upgrade the Java automatically. Upon action invocation the agent will restart using another JVM installation.
- (Windows) Since build agent .exe installation comes bundled with required Java, you can just reinstall the agent using .exe installer obtained from TeamCity server | **Agents** page.
- Install required Java on the agent and restart the agent - it should then detect it and provide an action to use newer Java in web UI.
- Install required Java on the agent and [configure agent](#) to use it.

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.

TeamCity treats equally all agents no matter if they are installed on the same or on different machines.

When installing several TeamCity build agents on the same machine, please consider the following:

- 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.

After having one agent installed, you can install additional agents by following the regular installation procedure (see exception for the Windows service below), but make sure that:

- The agents are installed in the separate directories.
- The agents have distinctive `workDir` and `tempDir` directories in `buildAgent.properties` file.
- Values for `name` and `ownPort` properties of `buildAgent.properties` are unique.
- No builds running on the agents have absolute checkout directory specified.

Moreover, make sure you don't have build configurations with absolute `checkout directory` specified (alternatively, make sure such build configurations have "`clean checkout`" option enabled and they cannot be run in parallel).

Usually, for a new agent installation you can just copy the directory of existing agent to a new place with the exception of its "temp", "work", "logs" and "system" directories. Then, modify `conf/buildAgent.properties` with a new "name", "ownPort" values. Please also clear (delete or remove value) for "authorizationToken" property and make sure "workDir" and "tempDir" are relative/do not clash with another agent.

If you want to install additional agents as services under Windows, do not opt for service installation during installer wizard or install manually (see also a [feature request](#)), then

modify the `<agent>\launcher\conf\wrapper.conf` file so that `wrapper.console.title`, `wrapper.ntservice.name`, `wrapper.ntservice.displayname` and `wrapper.ntservice.description` properties have unique values within the computer. Then run `<agent>\bin\service.install.bat` script under user with sufficient privileges to register the new agent service. Make sure to start the agent for the first time only after configured as described.

See [above](#) for the service start/stop instructions.

See also:

Concepts: Build Agent

Build Agent Configuration

Configuration settings of the build agent are stored in a configuration file <TeamCity Agent Home>/conf/buildagent.properties. The file can also store properties that will be published on the server as **Agent properties** and can participate in the **Agent Requirements** expressions.
Also, 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 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
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

## Optional
## The IP address which will be used by TeamCity server to connect to the build agent.
## If not specified, it is detected by build agent automatically,
## but if the machine has several network interfaces, automatic detection may fail.
#ownAddress=

## Optional
## A port that TeamCity server will use to connect to the agent.
## Please make sure that incoming connections for this port
## are allowed on the agent computer (e.g. not blocked by a firewall)
ownPort=9090
```



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 "name" property is not specified, the server will generate build agent name automatically. By default, this name would 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 restarts loading the new changes automatically.

See also:

Concepts: [Build Agent](#)

Administrator's Guide: [Predefined Build Parameters](#) | [Configuring Agent Requirements](#) | [Configuring Build Parameters](#)

Setting Up TeamCity for Amazon EC2

TeamCity Amazon EC2 integration allows 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 [Cloud-VMWare plugin](#) and [Implementing Cloud support](#).

General Description

It is assumed that the machine images are pre-configured to start TeamCity agent on boot. (See details [below](#)) 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 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 cloud profile number of running instances limit 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. **Since TeamCity 7.1**, if 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.

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 Amazon EC2 image (AMI) with 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](#).

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 setup 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).
 - For Amazon EC2 on Windows, it could be necessary to add a dependency from TeamCity Build Agent service to the `EC2Config` service
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).
7. Configure TeamCity Cloud (e.g. Amazon EC2) support on TeamCity server.

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 has been auto-updated.

Estimating EC2 Costs

Usual Amazon EC2 pricing applies. Please note that Amazon charges can depend on the specific configuration implemented to deploy TeamCity. We advice 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.

Installing Additional Plugins

For the list of available plugins, see [plugins list](#)

Steps to install a TeamCity plugin:

1. Shutdown the TeamCity server.
2. Copy the zip archive with the plugin into the <TeamCity Data Directory>/plugins directory.
3. Start the TeamCity server: the plugin files will be unpacked and processed automatically. The plugin will be available in the **Plugins List** in the **Administration** area.



To uninstall a plugin, shutdown the TeamCity server and remove the zip archive with the plugin from the <TeamCity Data Directory>/plugins directory.

If the plugin uses obsolete, pre-4.0 packaging:

Installing Jar-packaged Server-side Plugin

1. Shutdown the TeamCity server.
2. Copy the plugin jar file into the `<TeamCity web application>/WEB-INF/lib` directory. The default TeamCity web application is `<TeamCity home>/webapps/ROOT`.
3. If the plugin requires any libraries, copy them into the `<TeamCity web application>/WEB-INF/lib` directory also.
4. Start the TeamCity server.

Installing Agent-side Plugin (pre-4.0 packaging)

- Copy the agent plugin into the `<TeamCity web application>/WEB-INF/update/plugins` directory.

All the agents will be upgraded automatically. No agent or server restart is required.

Installing Agent Tools

TeamCity allows you to install additional tools (i.e. files/binary distributions) on all the agents.

This 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 tool to be distributed can be a separate folder or a .zip archive:

- If you use a separate folder, TeamCity will use the folder name as the tool name on all agents.
- If you use a zip file, TeamCity will use 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.

To distribute a set of files to all agents, perform the following:

1. Create the `.tools` directory under `<TeamCity Data Directory>/plugins`.
2. Under the `.tools` directory, create a directory or a .zip file containing the files you want to distribute to all the agents.
3. TeamCity monitors the content of this folder so the tool will be automatically distributed to all agents and the files will appear in the `<Agent Home Directory>/tools` folder. No server restart is necessary. Agents will restart in the process of obtaining the tool.

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.<your tool name>`.

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 remove the previously installed tool from all agents, delete the .zip file or the folder with the tool from the `<TeamCity Data Directory>/plugins/.tools` directory. The tool will be removed from all agents.

Upgrade Notes

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 than 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.

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 in your settings 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

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 no longer included in the "`build`" node. It has become an element instead of attribute to match the full build data representation.

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.

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 than regularly if there are many builds on the server. Following cleanups will then run a bit faster than in previous versions.

Changes from 8.0 to 8.0.3

No noteworthy changes.

Changes from 7.1.x to 8.0

Project and Build Configuration IDs

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=<locators>" 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|TW:Teamcity Queue Manager, 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-archives.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).

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](#)). 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 form `.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/subfolder/...` 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 `starteam://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

Upgrade



Unless specifically noted, TeamCity does not support downgrade. It is strongly recommended to [back up your data](#) before any upgrade.

Before upgrading TeamCity:

1. 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](#)
3. [Download](#) the new TeamCity version
4. Carefully review the [Upgrade Notes](#)

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 upgrade, 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. See the effective release date at the [page](#).

Please note that the licensing policy in TeamCity versions 5.0 and above are different from that of the previous TeamCity versions. Review the [Licensing Policy](#) page and the [Licensing and Upgrade](#) section on the official site.

If you are 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](#) JetBrains sales department.

Upgrading TeamCity Server

TeamCity supports upgrades from any of the previous versions to the current one. Downgrades are not supported unless specifically noted. 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.



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.

Upgrading Using Windows Installer

1. [Create a backup](#).
2. If you have any of the Windows service settings customized (like server [memory settings](#)), store them to repeat the customizations later. The same applies to customizations of the bundled Tomcat server (like port, https protocol, etc.), if any.
3. Run the new installer. Confirm uninstalling the previous installation. When prompted, specify the <TeamCity data directory> used by the previous installation.
4. Make sure you have the external database driver [installed](#) (this applies only if you use external database).
5. [Restore](#) any customizations of Windows services and Tomcat configuration that you need
6. Start up the TeamCity server (and agent, if it was installed together with the installer).
7. Review the [TeamCity Maintenance Mode](#) page to make sure there are no problems encountered, and confirm the upgrade by clicking corresponding button. Only after that all data will be converted to the newer format.



If you are upgrading from TeamCity 6.0 to a higher version, you can perform backup during the upgrade process right from the [TeamCity Maintenance Mode](#) page.

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. 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.
3. Unpack the new archive to the location where TeamCity was previously installed.
4. If you use Tomcat server (your own or bundled in .tar.gz TeamCity distribution), it is recommended to delete content of the work directory. Please note that this may affect other web applications deployed into the same web server.
5. Make sure you have the external database driver [installed](#) (this applies only if you use external database).
6. If you use custom plugins that do not reside in <TeamCity Data Directory>, [install them](#).
7. Specify additional [TeamCity server startup properties](#), if required.
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 corresponding button. Only after that all the configuration data and database scheme are updated by TeamCity converters.

Upgrading Build Agents

- [Automatic Build Agent Upgrading](#)
- [Upgrading the Build Agent to 4.0](#)
 - [Upgrading Build Agent Launcher](#)
 - [Upgrading the Build Agent Windows Service Wrapper](#)

Automatic Build Agent Upgrading

On starting newer TeamCity server, TeamCity agents connected to the server are updated automatically. The agent (agent.bat, agent.sh, or agent service) will download the latest agent upgrade 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 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 is to

- back up `<Agent Home>/conf/buildAgent.properties` file.
- uninstall/delete existing agent.
- install the new agent version.
- restore 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 agent under Windows using service, you can also need to upgrade 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 new way of managing Windows service (service wrapper) for the build agent: Java Service Wrapper library.

One of advantages of using new service wrapper is ability to change any JVM parameters of the build agent process.

1.x versions installed Windows service under name **agentd**, while 2.x versions use **TeamCity Build Agent Service <build number>** name.

The service wrapper will not be migrated to new version automatically. You do not need to use the new service wrapper, unless you need its functionality (like changing agent JVM parameters).

To use new service wrapper you should uninstall 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 `<agent>/launcher/conf/wrapper.conf` file.
5. Delete `<agent>/launcher`.
6. Rename `<agent>/launcher.latest` to `<agent>/launcher`.
7. Edit `<agent>/launcher/conf/wrapper.conf` file. Check that the '`wrapper.java.command`' property points to the `java.exe` file. Leave it blank to use registry to lookup for java. Leave '`java.exe`' to lookup `java.exe` in PATH. For a standalone agent the service value should be `../jre/bin/java`, for and agent installation on the server the value should be `.../..../jre/bin/java`. The backup version of `wrapper.conf` file may 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 for 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 on the TeamCity startup you see the TeamCity Maintenance page, that means the TeamCity instance requires technical maintenance which for security reasons is to be performed by a **system administrator** who has access to the computer where 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 the 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 *Maintenance Authentication 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 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 up, do it at this point. Once TeamCity converts the data, downgrade won't be possible. If you need to return to earlier TeamCity version, you will be able to do so only by restoring the data from corresponding backup. Please refer to the [TeamCity Data Backup](#) section for instructions.

When you are sure you have your data backed up, 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.

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 description of what is stored where on [Manual Backup and Restore](#) page.

This page covers external database setup for the first use with TeamCity. If you evaluated TeamCity with internal database and want to preserve the data while switching to an external database, please refer to [Migrating to an External Database](#).

By default, TeamCity runs using an internal database based on the HSQLDB database engine. 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 out of disk space condition). Also, internal database can 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.**

The database connection settings are configured in `<TeamCity Data Directory>\config\database.properties` file. The file is a Java properties file. If the file is not present, TeamCity automatically uses internal database.

Since TeamCity 8.1, you can configure the database to be used on the TeamCity start: all you need to do is select the type of the database and specify database connection settings. You may also need to download the JDBC driver for your database.

The information below provides instruction on manual configuration of the external database to be used by TeamCity.

This page covers:

- Selecting External Database Engine
- General Steps
- Database Configuration Properties
- Database Driver Installation
- MySQL
- PostgreSQL
- Oracle
- Microsoft SQL Server

Selecting External Database Engine

TeamCity supports MySQL, PostgreSQL, Oracle and MS SQL databases.

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.

General Steps

1. If you already ran TeamCity but do not want to preserve any data, delete TeamCity Data Directory.



If you delete TeamCity Data Directory, all the data you entered into TeamCity will be lost. To preserve your data, please refer to the [migration guide](#).

2. Run TeamCity with the default settings to create the `<TeamCity Data Directory>`.
3. Shutdown the TeamCity server.
4. Perform database-specific steps described below.
5. Start the server.



Please note that TeamCity actively modifies its own database schema. 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.

Database Configuration Properties

The database connection settings are configured in `<TeamCity Data Directory>\config\database.properties` file. If the file is not present, TeamCity automatically uses internal database.

TeamCity uses Apache DBCP for database connection pooling. Please refer to <http://commons.apache.org/dbcp/configuration.html> for detailed description of configuration properties. Example configurations for each of supported databases are provided in the sections below.



- For all supported databases there are template files with database-specific properties, which you can use. These templates are located in the `<TeamCity Data Directory>/config` directory and have the following name format: `database.<database_type>.properties.dist`. In order to use a template, copy it to `database.properties` and then modify it to specify correct properties for your database connections.

Database Driver Installation

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 `<TeamCity Data Directory>/lib/jdbc` directory (create it if necessary).

Please note that the .jar files should be compiled for the Java version not greater than that used for running TeamCity version, otherwise you might see "Unsupported major.minor version" errors related to the database driver classes.

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 user in MySQL that will be used by TeamCity must be granted all permissions on the TeamCity database.

On TeamCity server side

Installation:

- 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.
- Place `mysql-connector-java-*-.bin.jar` from the downloaded archive into the `<TeamCity Data Directory>/lib/jdbc`.
- Create an empty database for TeamCity in MySQL and grant permissions to modify this database to a user from which TeamCity will work with this database. For this you can execute the following SQL commands from MySQL console:

```
create database <database name> default charset utf8;
grant all privileges on <database name>.* to <user>@localhost identified by '<password>';
```

- In the `<TeamCity Data Directory>/config` folder rename `database.mysql.properties.dist` file to `database.properties` and specify the required settings in this file:

```
connectionUrl=jdbc:mysql://<host>/<database name>
connectionProperties.user=<user>
connectionProperties.password=<password>
```

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 a user which TeamCity will use to work with this database.

On TeamCity server side

1. Download the PostgreSQL JDBC4 driver from <http://jdbc.postgresql.org/download.html> and place it into the <TeamCity Data Directory>/lib/jdbc.
2. In <TeamCity Data Directory>/config rename database.postgresql.properties.dist file to database.properties and specify the required settings in this file:

```
connectionUrl=jdbc:postgresql://<host>/<database name>
connectionProperties.user=<user>
connectionProperties.password=<password>
```



TeamCity doesn't specify which schema should be used for its tables. By default, PostgreSQL creates tables in the 'public' schema (the 'public' is the name of the schema). TeamCity can also work with other PostgreSQL schemas. To switch to another schema, do the following:

1. 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 username has to be specified in the 'database.properties' in TeamCity, and has to be in lower case.
The schema has to be empty (it should not contain any tables).

2. Start TeamCity.

Oracle

Supported versions

On Oracle server side

1. Create an Oracle user account/schema for TeamCity.
 - Make sure to set up the database to use UTF8
 - Grant the CREATE SESSION, CREATE TABLE, EXECUTE ON SYS.DBMS_LOCK permissions to a user which TeamCity will use 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

1. Get the Oracle JDBC driver.
Supported driver versions are 10.2.0.1.0 and higher.

Place the following files:

- ojdbc6.jar
- 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. In the <TeamCity Data Directory>/config folder rename database.oracle.properties.dist file to database.properties and specify the required settings in this file:

```
connectionUrl=jdbc:oracle:thin:@//<host>:1521:<servicename>
connectionProperties.user=<user>
connectionProperties.password=<password>
```

Microsoft SQL Server

Supported versions

On MS SQL server side

1. Create a new database. Using case sensitive collation (collation name ending with '_CS_AS') is recommended and is mandatory for the certain functionality (like using non-Windows build agents).
2. Create TeamCity user and ensure that this user is the owner of the database (grant the user dbo rights). This requirement is necessary because the user needs to have ability to modify the database schema.
If you're going to use SSL connections, ensure that the version of MS SQL server and the version of java (on the TeamCity side) are compatible. We recommend using the latest update of SQL server:
 - SQL Server 2012 - all versions
 - SQL Server 2008R2 - Service Pack 2 or Service Pack 1 cumulative update 6
 - SQL Server 2008 - Service Pack 3 cumulative update 4
 - SQL Server 2005 - only with JDK 6 update 27 or lower on the TeamCity side
See [details](#) on compatibility issues.
3. Allocate sufficient transaction log space. The requirements vary depending on how intensively the server will be used. In the beginning, it's recommended to setup 1Gb for small installations and 2-4Gb for large ones.

On TeamCity server side

You can use either [MS native JDBC driver](#) (recommended) or [JTDS JDBC driver](#) one.

Native driver

1. Download the MS **sqljdbc** package from the [Microsoft Download Center](#) and unpack it. Let us assume the directory where you've unpacked the package into is called **sqljdbc_home**.
2. Copy the sqljdbc4.jar from the just downloaded package into the **TeamCity Data Directory**/lib/jdbc directory.
3. In the <TeamCity Data Directory>/config folder rename database.mssql.properties.dist file to database.properties file and specify the following required settings in this file:

```
connectionUrl=jdbc:sqlserver://<host>:1433;databaseName=<database_name>
connectionProperties.user=<user>
connectionProperties.password=<password>
```

If you use named instance you can specify the instance name in the connection URL, like the following:

```
connectionUrl=jdbc:sqlserver://<host>\\\<instance_name>:1433;databaseName=<database_name>
...
```

If you prefer to use MS SQL integrated security (Windows authentication), follow the additional steps:

1. Ensure that your Java bitness is the same as Windows bitness (in other words, [use 64-bit Java with 64-bit Windows and 32-bit Java with 32-bit Windows](#)).
2. Copy the **sqljdbc_home** /enu/auth/x86/sqljdbc_auth.dll (in case of 32-bit system) or **sqljdbc_home** /enu/auth/x64/sqljdbc_auth.dll (in case

of 64-bit system) into your Windows/system32 directory (or another directory denoted in %PATH%). Ensure that there are no other sqljdbc_auth.dll files in your system).

3. In the <TeamCity data directory>/config folder create file database.properties and specify the connection URL (with no user names or passwords) in this file:

```
connectionUrl=jdbc:sqlserver://<host>:1433;databaseName=<database name>;integratedSecurity=true
```

More details about setup integrated security for MS SQL native jdbc driver can be found [here \(for MS SQL 2005\)](http://msdn.microsoft.com/en-us/library/ms378428(v=sql.100).aspx) and [here \(for MS SQL 2008\)](#).

JTDS driver



The JTDS driver does not support Unicode operations. Using the Microsoft native JDBC driver is recommended.

1. Download the latest 1.2.x JTDS driver distribution file (zip file), unpack the jtds-* .jar driver jar and place it to <TeamCity Data Directory>/lib/jdbc. (TeamCity is not tested with the driver version 1.3+. It also requires that TeamCity server is run under JDK 1.7+).
2. In the <TeamCity Data Directory>/config folder rename database.mssql.properties.dist file to database.properties and specify the required settings in this file:

```
connectionUrl=jdbc:jtds:sqlserver://<host>:1433/<database name>
connectionProperties.user=<user>
connectionProperties.password=<password>
connectionProperties.instance=<instance_name>
```

To use Windows authentication (SSPI) to connect to your SQL Server database, make sure there are no connectionProperties.user and connectionProperties.password properties specified in the database.properties file and also copy jtds-XXX-dist\x86\SSO\ntlmauth.dll file from the JTDS driver package to <TeamCity Home>\bin (when TeamCity is run under x86 JVM), for x64 JVM use jtds-XXX-dist\x64\SSO\ntlmauth.dll file. Also setup TeamCity server (service or process) to be run under user account that has access to the database.



The jtds driver doesn't know a "default" port value, so the port number in the connectionUrl is a mandatory parameter.

Please make sure SQL Server is configured to enable TCP connections on the port used in the connectionUrl.
If you use named instance you can specify the instance name by following means:

- Add the "instance" property into the connection URL, like the following:
`connectionUrl=jdbc:jtds:sqlserver://<host>:1433/<database name>;instance=sqlexpress`
- Or, specify corresponding property in the database.properties file:
`connectionProperties.instance=<instance_name>`

See also:

Installation and Upgrade: Migrating to an External 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

These steps describe switching to another database *without preserving* the build history and user data. See [Full Migration](#) below 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. You will be asked to provide the [maintenance authentication token](#).

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>
```



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). You will be asked to provide the [maintenance authentication token](#).

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\teamcity-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: 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

1. On the **General** tab of the page, type your new password in the **Password** and **Confirm password** fields.

 If you don't see these fields, this means that TeamCity uses the authentication scheme other than the default one, and it is not possible to change your password in TeamCity.

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 [My Changes](#), [since TeamCity 8.1 - Changes](#) page,
- 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

Viewing your Roles and Permissions

- 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).

- Subscribing to Email, Jabber, IDE, Windows Tray Notifications
 - What Will Be Watched
 - Which Events Will Trigger Notifications
- Customizing RSS Feed Notifications
 - Feed URL Generator Options
 - Additional Supported URL Parameters
 - Example

Subscribing to Email, Jabber, IDE, Windows Tray Notifications

TeamCity allows you to flexibly adjust the notification rules, so that you receive notifications only on the events that you are interested in. To subscribe to notifications:

- 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.
- Click the required notifications type:
 - Email Notifier:** to be able to receive email notifications, your email address must be specified at 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.
- 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 TeamCity server email and Jabber settings. System Administrators can also [change the templates](#) used for notifications.

What Will Be Watched

Builds affected by my changes	Check to monitor only the builds that contain your changes. Make sure your Version Control Username Settings are correct.
Builds from the project	Select a project 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 .
Builds from the selected build configurations	Check to monitor all of the builds of the selected build configurations. Hold the Ctrl key to select several build configurations.

System wide events

Select to be notified about system wide events.

Which Events Will Trigger Notifications

The build fails	Check this option to receive notifications about all of the failed builds in the specified projects and build configurations. If investigation for a build configuration is assigned to someone, the failed build notification is sent only to that user. If the Builds affected by my changes option is selected in the Watch area, you will receive notifications about the builds including your changes and all the following builds until a successful one. See the next two options as well.
• Ignore failures not caused by my changes	<i>This option can only be enabled when the Watch builds affected by my changes option is used.</i> Check this option not to be notified when a build fails without any new problems after a build with your changes.
• Only notify on the first failed build after successful	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.
The build is successful	Check this option to receive notifications when a build of the specified build configurations executed successfully.
• 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 <i>not</i> be sent.
The first error occurs	Check this option to receive notifications about a "failing" build as soon as the first build error is detected, even before the build has finished.
The build starts	Check this option to receive notifications as soon as a build of the specified build configurations starts.
The build fails to start	Check this option to receive notifications when a build of the specified build configurations fails to start .
The build is probably hanging	Check this option to receive notifications when TeamCity identifies a build of the specified build configurations as hanging .
An investigation is updated	Check this option to receive notifications on changing a build configuration or test investigation status, t.g. someone is investigating the problem, or problems were fixed, or the investigator changed.
Tests are muted or unmuted	Check this option to receive notifications on the test mute status change in the affected build configurations.
An investigation is assigned to me	<i>This option is available only if the System wide events option is selected in the Watch area.</i> Check the option to be notified each time you start investigating a problem.



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.

Note that you can reorder the notification rules specified.



Note, that you may already have some notification rules configured by your System Administrator for the user group you're included in.

- 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.

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	Specify which build configurations to display in the Select build configurations or projects field. The following options are available: <ul style="list-style-type: none">• With the external status enabled: if this option is selected, the next field shows the list of build configurations with the Enable status widget option set, and build configurations visible for everybody without authentication.• 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.• All: Select this option to show a complete list of build configurations. The option requires HTTP authorization. Selecting this option enables the Feed authentication settings field. If the external status for the build configuration is not enabled, the feed will be available only for authorized users.
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	Specify the types of builds to be informed about: <ul style="list-style-type: none">• All builds• Only successful• Only failed
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.
Include credentials for HTTP authentication	Check this option 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.
TeamCity User, Password	Type the user name and password which will be used for HTTP authorization. These fields are only available when the Include credentials... option is checked.
Copy and paste URL to your feed reader or Subscribe	This field displays a 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 corresponding section 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

For a project developer it is important to understand whether his commit brought a build failure or not. On the [My Changes](#) page you can review the commits you've made and how they have affected builds. **Since TeamCity 8.1**, the page has a users selector and you can see changes made by any TeamCity user. The page was renamed to [Changes](#).



Changes made by a user are displayed correctly only when appropriate [VCS usernames](#) are defined.

Your changes are presented in a timeline. By default the page doesn't show your commits to the build configurations that you hid on the Projects dashboard. If you want to remove this filter and view all build configurations, click the [show all](#) link on the top of the page.



From this page you can:

- View all your commits, and changes included into your personal builds.
- View how your changes have affected the builds.
- See whether there are new failed tests caused by your changes.
- Navigate to the issue tracker, if [issue tracker integration is configured](#).
- Open possibly problematic files in your 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 suspicious builds with your change. For personal changes, all builds are shown.

Note, that problems which have an investigator/responsible are not considered critical (unless you are the investigator).

On the right side of the page you can see the visual representation of how your commit has affected different builds. The legend is following:

	Build failure <i>with</i> new problems
	Build failure <i>without</i> new problems
	There is a successful build with this change
	No build has run with this change

See also:

Concepts: Build State | Change

Working with Build Results

In TeamCity all information about a particular build, whether it is running or finished, is accumulated on the **build results** page. However, some data is accessible only after the build is finished. The build results can be accessed the Build Configuration home page and from various places in the TeamCity web UI where a build number appears as a link.

The screenshot shows the TeamCity interface for a build configuration named 'Geym Trunk'. The top navigation bar includes links for Overview, History, Change Log, Issue Log, Statistics, and Compatible Agents. The 'Overview' tab is selected. Below the tabs, there's a summary section with 'Pending changes' (0 pending changes) and 'Current status' (idle). A 'Recent history' table lists four completed builds (Build #52181, #52180, #52148, #52147) with their results (Success), artifacts (None), changes (Victory Bedrock...), start time (01 Aug 13), and duration (00:00:00).

Results	Artifact	Changes	Started
Success	None	Victory Bedrock... (1)	01 Aug 13
Success	None	Victory Bedrock... (1)	01 Aug 13
Success	None	Karl Macaw... (1)	01 Aug 13
Success	None	Victory Bedrock... (1)	01 Aug 13

The following information is available on the page:

- Build Overview
 - Failed 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 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 and higher, using jstack utility
- Non-Windows, with JDK 1.5 and higher, using jstack utility

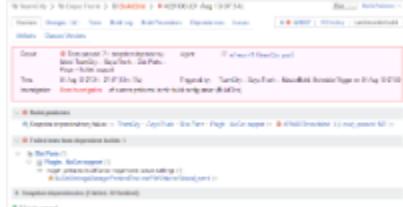
The information on the tab may vary depending on the build runners enabled. If configured, you will see the **Code Coverage Summary** and a link to the full report or/ and the number of duplicates found in your code and a link opening the **Duplicates** tab, etc. Refer to the sections below for details 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.

Failed Tests

The tests failed 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
- [Label the build sources](#)
- Configure the VCS settings of the build configuration (if you have enough permissions).

The **Changes** tab provides advanced filtering capabilities for the list of changes. For each change (when expanded) you can:

- Explore the change in details
- View the difference between the current and the previous versions of the modified files with the TeamCity [difference viewer](#) by clicking the name of a modified file
- [Trigger a custom build](#) with this change
- Open modified files in your IDE
- Review the change in an [external change viewer](#), if configured by the administrator.

Select the **Show graph** to see 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.

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">Select the type of items to view: tests, suites, packages/namespaces or classes.Type in the string (e.g. test name from the list) thus providing change of scope for the list.Select a test status.
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

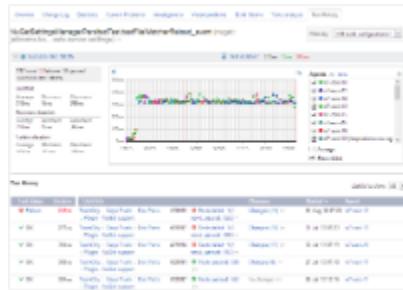
To navigate to the history of a particular test, click the arrow next to the test name and select **Test History** from the drop-down.
There are several places where tests are listed and from where you can open Test History.

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

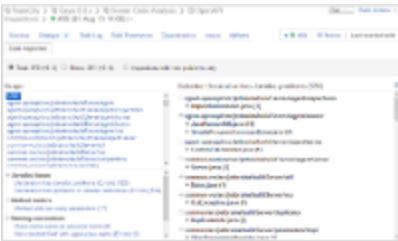
If you have code coverage configured in your build runner, a dedicated tab with the full HTML code coverage report appears on the build results page:



File	Coverage %	Delta Coverage %
src/main/java/com/company/project/service/SomeService.java	85.0%	+1.0%
src/main/java/com/company/project/repository/SomeRepository.java	90.0%	+1.0%
src/main/java/com/company/project/controller/SomeController.java	80.0%	+1.0%
src/main/java/com/company/project/entity/SomeEntity.java	75.0%	+1.0%
src/main/java/com/company/project/serviceimpl/SomeServiceImpl.java	82.0%	+1.0%
src/main/java/com/company/project/repositoryimpl/SomeRepositoryImpl.java	88.0%	+1.0%
src/main/java/com/company/project/controllerimpl/SomeControllerImpl.java	78.0%	+1.0%
src/main/java/com/company/project/entityimpl/SomeEntityImpl.java	73.0%	+1.0%
src/main/java/com/company/project/service/SomeServiceTest.java	85.0%	+1.0%
src/main/java/com/company/project/repository/SomeRepositoryTest.java	90.0%	+1.0%
src/main/java/com/company/project/controller/SomeControllerTest.java	80.0%	+1.0%
src/main/java/com/company/project/entity/SomeEntityTest.java	75.0%	+1.0%
src/main/java/com/company/project/serviceimpl/SomeServiceImplTest.java	82.0%	+1.0%
src/main/java/com/company/project/repositoryimpl/SomeRepositoryImplTest.java	88.0%	+1.0%
src/main/java/com/company/project/controllerimpl/SomeControllerImplTest.java	78.0%	+1.0%
src/main/java/com/company/project/entityimpl/SomeEntityImplTest.java	73.0%	+1.0%

Code Inspection Results

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 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 Build Problems | Viewing Tests and Configuration Problems
Administrator's Guide: Creating and Editing Build Configurations

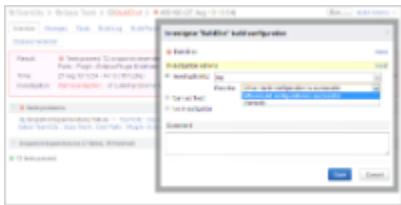
Investigating Build Problems

When for some reason a build fails, TeamCity provides handy means of figuring out which changes might have caused the build failure and lets you 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.

A subject for the investigation is not necessarily the whole build, it can also be some particular test. For each project you can find out what problems are currently being investigated and by whom using the dedicated **Investigations** tab.

To start 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 and click **Assign**.



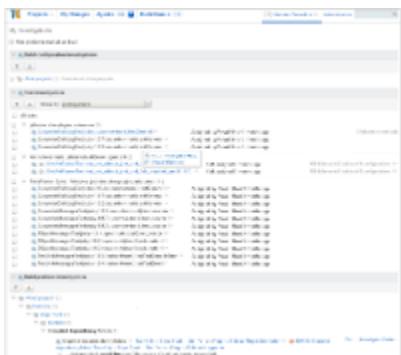
Note that normally an investigation is automatically removed once the build configuration becomes green or the test passes. However, **starting with TeamCity 7.1**, you can specify that an investigation should be resolved manually. This is useful in case of so called "flickering tests", i.e. when a test fails from time to time and the "green" status of a build is not an indicator of the problem resolution.

To investigate a problem in more than one build configuration, click **more** and select the desired build configurations.

To start an investigation of 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 **Investigation**.

My Investigations

To view all problems assigned to you to investigate, click the box with a number next to your name.



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 a test failure.

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.

Since **TeamCity 7.1**, investigations assigned to you are also highlighted in the Web UI if you enable the "Highlight my changes and investigations" option on your [profile settings](#).

See also:

Concepts: Build Configuration Status

User's Guide: Viewing Tests and Configuration Problems

Viewing Tests and Configuration Problems

To view problematic build configurations and tests in your project, open the Project Home page and go to the **Current Problems** tab.

In this section you will read about:

- Using Current Problems Tab
 - Viewing Problems on Overview Page
 - Viewing Tests Failed within Last 120 Hours

Using 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.

You can also view all tests failed within the last 120 hours for the whole project or a particular build configuration using the corresponding link on the right of the page. For details, refer to the section below.

Viewing Problems on Overview Page

Starting with TeamCity 7.1, you can see the number of tests failed in build configurations as well as other problems right on the Overview page.



When the data in the problematic build configuration is not expanded, the link under the tests number takes you to the Current Problems page. 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.

Viewing Tests Failed within Last 120 Hours

To view all tests failed within the last 120 hours for a project, navigate to the Project Home page, select the **Current Problems** tab and click the **View all tests failed within the last 120 hours** link. The **Problematic Tests** tab is opened.

To view problematic tests for a specific build configuration, use the **Show problematic tests for** drop-down list.

For each test you can view the total number of test runs, the failure count and last time the test failed.

Clicking the test details icon  opens the **Test History** tab with filterable data pertaining to the test.

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 separated into several tabs.

The number of tabs on the page may vary depending on your server or project configuration, e.g. **dependencies**, 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** the build configuration, and, if applicable, the number of [queued builds](#), 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 and 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

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:

- 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.



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 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. |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:** This chart tracks the maximum amount of time it took to fix the tests of the 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 .

Moreover, you can [add custom charts](#). In comparison with project-level charts, it is not possible to disable pre-defined charts on the build configuration level.

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.

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. In TeamCity you can search builds by build number, tag, build configuration name and other different parameters specifying one or several keywords and use [Lucene](#) search query syntax to get more precise results.

For complete list of available search fields (keywords) please refer to [Complete List of Available Search Fields](#) section.

In this section:

- [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 Query

In TeamCity you can search for builds using the Lucene query syntax. Though in TeamCity search query has two major differences, please refer to the [Lucene](#) documentation for complete syntax rules description.

To narrow your search and get more precise results you can use available search fields - indexed parameters of each build. Please refer to the [Complete list of available search fields](#) for more details.

Differences from Lucene Syntax

When using search query in TeamCity, please pay attention to the following major differences in query syntax from Lucene native syntax:

1. By default, TeamCity uses AND operator in query. That is, if you type in the following query: "failed @agent123", then you will get a list of all builds that have keyword "failed" in any of its search fields, and were run on build agent, which name is "agent123".
2. By default, TeamCity uses "prefix search", not exact matching like Lucene. For example, if you search for "c:main", TeamCity will find all builds of the build configuration which name starts with "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 which to search items similar in 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 the 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 usage of "*" and "?" wildcards in the build query.



Please do not type an asterisk sign, *, at the beginning of the search term because it can take a significant amount of time for TeamCity to search its database. For example, *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 specific agent.	agent:unit-77, or agent:agent14*
build		Find all builds that include changes with specific string.	build:254 or build:failed
changes		Find all builds that include changes with specific string.	changes:(fix test)
committers		Find all build that include changes committed by specific developer.	committers:ivan_ivanov
configuration	c	Find all builds from the specific build configuration.	configuration:IPR c:(Nightly Build)
file_revision		Find all builds that contain a file with specific revision.	file_revision:5
files		Find all build that include files with specific filename.	files:
labels	l	Find all builds that include changes with specific VCS label.	label:EAP l:release
pin_comment		Find all builds that were pinned and have specific word (string) in the pin comment.	pin_comment:publish
project	p	Find builds from specific project.	project:Diana p:Calcutta
revision		Find all builds that include changes with specific revision (e.g., you can search for builds with specific changelist from Perforce, or revision number in Subversion, etc.).	revision:4536
stamp		Find all builds that started at the specific time (search by timestamp).	stamp:200811271753
status		Find all builds with specific text in the build status text.	status:"Compilation failed"
tags	t	Find all builds with specific tag.	tags:buildserver t:release
tests		Find all builds that include specific tests.	tests:
triggerer		Find all builds that were triggered by specific user.	triggerer:ivan.ivanov
vcs		Find builds that have specific VCS.	vcs:perforce

Shortcuts

In addition to above mentioned search fields, you can use two following shortcuts in your query:



Please pay attention that when you use these shortcuts, you should not insert colon after it. That is, the query syntax is as follows: <shortcut><value to search>

Shortcut	Description	Example
----------	-------------	---------

#	Search for build number.	#<number>, e.g. #1234
@	Find all builds that were run on the specific agent.	@<agent's name>, e.g. @buildAgent1

Using Double-Colon

You can use double-colon sign (::) to search for 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 (for the list see table below). These magic keywords are formed of the '\$' sign and a word itself. The word can be shortened up to the one (first) syllable, that is, the \$labeled, \$l, and \$lab keywords will be equal in query. For example, to search for pinned builds of the "Nightly build" configuration in the "Mega" project you can type 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 which name is started with "Master" from projects with names started 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:Mai \$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.

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:

- Testing Frameworks
- Code Quality Tools
- TeamCity Configuration and Maintenance
- Managing Projects and Build Configurations
- Managing Licenses
- 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

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)
 - Maven2 (when tests are run by [Surefire Maven plugin](#); tests reporting occurs after each module test run finish)
 - [IntelliJ IDEA](#) project (when run with appropriate IDEA run configurations)
- NUnit for the following runners:
 - The NAnt (`nunit2` task)
 - The MSBuild ([NUnit community](#) or [NUnitTeamCity](#) tasks)
 - Microsoft Visual Studio Solution runners (2003, 2005 and 2008)
 - Any runner provided [TeamCity Addin for NUnit](#) is installed.
- MSTest 2005, 2008, 2010 (On-the-fly reporting is not available due to MSTest limitations)
- MSpec

There are also testing frameworks that have embedded support for TeamCity. e.g. [Gallio](#) and [xUnit](#).

See also [external 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 available [since TeamCity 8.1](#) supports [Ant](#), [IntelliJ IDEA Project](#), [Gradle](#) and [Maven](#) build runners.

.Net Tools

ReSharper-powered Tools

These are available if you use Visual Studio.

- [Inspections \(.NET\)](#) gathers results of JetBrains ReSharper Code Inspections in your C#, VB.NET, XAML, XML, ASP.NET, JavaScript, CSS and HTML code.
- [Duplicates Finder \(.NET\)](#) 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](#)

See more on [Configuring .NET Code Coverage](#).

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.

The following reports are supported:

- [FindBugs](#) (code inspections only)
- [PMD](#)
- [Checkstyle](#)
- [JSLint XML reports](#)

and the following code duplicates tools:

- [PMD Copy/Paste Detector XML 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.

TeamCity Configuration and Maintenance



Server configuration is only available to the [System Administrators](#).

To edit the server configuration:

1. On the [Administration](#) page click **Global Settings**.
2. From this page you can:
 - View <TeamCity data directory> and <artifacts directory>;
 - Specify the server URL;
 - View the current authentication scheme, configure the TeamCity welcome message and enable/disable anonymous login. Authentication scheme can be configured manually, as described in the [Configuring Authentication Settings](#) section.

This section also includes:

- [TeamCity Data Backup](#)
- [Configuring Authentication Settings](#)
- [TeamCity Startup Properties](#)
- [Configuring Server URL](#)
- [Configuring TeamCity Server Startup Properties](#)
- [Configuring UTF8 Character Set for MySQL](#)
- [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](#)

TeamCity Data Backup



This section describes backup options available for TeamCity 6.x and above. If you need to perform backup of earlier versions, please refer to the corresponding section in an appropriate version of documentation.

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. Restore is possible via the `maintainDB` console tool. Some limitations on the backed up data apply. 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, but requires the server be not running and has no limitations. Restore is possible the via the `maintainDB` console tool.
- **Manual backup:** is suitable if you want to manage the backup procedure manually.

If you need to back up the build agent's data, refer to [Backing up Build Agent's Data](#).

For instructions on how to restore the data from backup, refer to the [Restoring TeamCity Data from Backup](#) page.



We strongly urge you to make the backup of TeamCity data before upgrading. Note that TeamCity server **does not support downgrading**.

The recommended approach is either to perform backup described under [Manual Backup and Restore](#) or run backup from the [web UI](#) regularly (e.g. automated via [REST API](#)) with the level "Basic" - 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](#) for details on what is stored in the directories.

Note that for large production TeamCity installations export and import of the 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 **TeamCity Backup** page of the **Administration** section, specify backup parameters, as described below and start backup process.

Option	Description
Backup file	<p>Specify the name for the backup file, the extention (.zip) will be added automatically. By default, TeamCity will store the backup file under <TeamCity Data Directory>/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 <TeamCity Data Directory>/config/backup-config.xml file. For example:</p> <pre><backup-settings> ... <general> <backup-dir path="C:/TC-Backups" /> </general> ... </backup-settings></pre>
add time stamp 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="background-color: #e0f2e0; padding: 10px;"> <ul style="list-style-type: none">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 <i>time stamp suffix</i> option, which allows to avoid this.Time stamp suffix has specific format: sorting backup files alphabetically will also sort them chronologically.</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. You can select the "basic" scope, which includes server settings, projects and builds configurations, plugins and database, to reduce the resulting file size and the time spent on backup. However, you'll be able to restore only those settings which were backed up.</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 backup created during server running. To include the builds, consider another [creating backup](#) approach while the server is not running.
- Backup process takes some time that depends on how many builds there are in 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 won't 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 specific structure that does not depend on OS or database type you use. Thus, you can use the backup file to restore your data even on another Operating System, or with another database. If you'll change the contents of this file manually, TeamCity won't be able to restore your data.

On the **History** tab of **Administration | TeamCity Backup** page you can review 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. If you have performed backup by means of `maintainDB` utility, they 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

In TeamCity you can back up server data, [restore it](#), and [migrate between different databases](#) using the `maintainDB.bat | sh` utility. For the data backup, TeamCity also provides web UI part, in the **Administration** section of the TeamCity web UI.

`maintainDB` utility is located in the `<TeamCity Home>/bin` directory. It is only available in TeamCity `.tar.gz` and `.exe` distributions.

To get short reference for all available `maintainDB` options, run `maintainDB` from the command line with no parameters.

This section covers:

- [Backing up Data](#)
 - [Performing TeamCity Data Backup with maintainDB Utility](#)
 - [maintainDB Usage Examples for Data Backup](#)

Backing up Data

TeamCity allows backing up the following data:

- Server settings and configurations, which includes all server settings, properties, and project and build configuration settings
- Database
- Build logs
- Personal changes
- Custom plugins and database drivers installed under [TeamCity Data Directory](#).

Backup of the following data is **not supported**:

- build artifacts (because of their size). If you need the build artifacts, please also backup content of `<TeamCity Data Directory>/system/artifacts` directory manually.
- data used by various plugins (NuGet feed, etc.) and audit settings diff data. If you want to preserve it, please also backup content of `<TeamCity Data Directory>/system/pluginData` directory manually.
- TeamCity application manual customizations under `<TeamCity server home>`, including used server port number
- TeamCity application logs (they also reside under `<TeamCity server home>`).
- Any manually created files under `<TeamCity Data Directory>` that do not fall into previously mentioned items.

By default, if you run `maintainDB` utility with no optional parameters, build logs and personal changes will be omitted in the backup.

The default directory for the 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.

Default format of the backup file name is `TeamCity_Backup_<timestamp>.zip`; the `<timestamp>` suffix is added in the 'YYYYMMDD_HHMMSS' format.

Performing TeamCity Data Backup with maintainDB Utility



Before starting data backup you need to stop TeamCity server.

To create data backup file, from the command line start `maintainDB` utility with the `backup` command:

```
maintainDB.[cmd|sh] backup
```

To specify type 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

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
```

If backup is started when the TeamCity server is running, running and queued builds are not included into backup. If the TeamCity server is shut down, all builds are included into backup.

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>
```

The above command will result in creating new zip-file with specified name in the default backup directory.

To add timestamp suffix to the 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 backup file in the custom directory, run maintainDB with -F option:

```
maintainDB backup -F <absolute path to the custom backup directory>
or
maintainDB backup --data-dir <absolute path to the custom backup directory>
```

See also:

Installation and Upgrade: Setting up an External Database | Migrating to an External Database

Manual Backup and Restore

Server Manual Backup

Other ways to create a backup are [available](#). You can use these instructions if you want fine-grained control over the backup process or need to use a specific procedure for your TeamCity backups.



Before performing the backup procedures, you need to **stop** the TeamCity server.

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 logs and 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 the database. The `<TeamCity Data Directory>/system/buildserver.*` files store internal database (HSQLDB) data. You should back them up if you use HSQLDB (the default setting).

Database Data

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 HSQLDB, 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 external database, back up your database schema used by TeamCity using 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 didn't:

- 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, pursue the backup procedure of the servlet container used.

Log files

If you need TeamCity log files (which are mainly used for problem solving or debug purposes), back up the `<TeamCity home directory>/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 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

When restoring database, ensure 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 another database.

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.



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 or more recent versions can be restored using the same or more recent TeamCity versions.

To restore a TeamCity server from a backup file:

1. Make sure the TeamCity server is not running.
2. Create the empty target [TeamCity Data Directory](#)
3. [Configure your external database](#).
To restore into an internal database, use the `.BuildServer\config\database.hsqldb.properties.dist` file as a [template](#).
4. Place the database drivers into `lib/jdbc` sub directory.
5. Use the `maintainDB` utility located in the `<TeamCity Home>/bin` directory (only available in TeamCity `.tar.gz` and `.exe` distributions).
6. Use the `restore` command:

```
maintainDB[cmd|sh] restore -F <full file name of TeamCity backup file> -A <path to TeamCity Data Directory> -T <path to the database.properties file of the target database>
```

The `-A` argument can be omitted if you have the `TEAMCITY_DATA_PATH` environment variable set.

The `-T` argument can be omitted if you want to restore the data into the same database the backup was created from.

By default, `maintainDB` looks for the specified backup file in the default backup directory: `<TeamCity Data Directory>/backup`. If the file is not found, the process will be aborted with an error. To override this setting, you can specify the *absolute path* to the desired backup file in a custom directory using the `-A` option.

By default, if no other option other than `-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: `-C`, `-D`, `-L`, and `-P`.



To get the reference for the available options of `maintainDB`, run the utility without any command or option.

Restoring Data from Backup to Another Database

You can restore backed up data into a different database; the type of the source (from which the data is backed up) and target (to which the data will be restored) databases do not matter. For instance, you can restore data from a HSQL database to a MySQL database, as well as restore a backup of MySQL database to a new MySQL database.

Essentially, restoring data from a backup file to a different database is a migration process.

To restore database data to a different database:

1. If you need to preserve properties of the target database, create a new `database.properties` file from a template, which corresponds to the type of the target database, or copy the existing one to a temporary directory.
2. Run the `maintainDB` utility with the `restore` command and the `-T` option:

```
maintainDB restore -F <backup file> -T <absolute path to the database.properties file of the target database>
```

All backed up scopes will be restored and the database will be restored to a new one.

To restore database only, use the `-D` option.

To resume the database restore after an interruption:

Run the `maintainDB` utility with the `restore` command with the required options and the additional `--continue` option:

```
maintainDB restore -F <backup file> -T <absolute path to the database.properties file of the target database> --continue
```

See also:

[Administrator's Guide: Creating Backup via maintainDB command-line tool](#)

Configuring Authentication Settings

Out-of-the-box TeamCity supports five **authentication modules** including 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)

The default authentication includes the [Built-in](#) and [Basic HTTP Authentication](#) modules.

Configuring Authentication Modules

There are two ways you can configure authentication modules used by Teamcity:

- via the Web UI
- By editing `<TeamCity data directory>/config/auth-config.xml` file

Configuring Authentication Modules Using Web UI

The currently used authentication modules are displayed on the [Administration | Authentication](#) page.

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](#)
- [LDAP with NTLM](#)
- [NTLM](#)

Advanced Mode

Use the advanced mode to add / remove authentication modules:

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.

The changes made in the UI will be reflected in the `auth-config.xml` file.

Configuring Authentication Modules Using `auth-config.xml`

You only need to use this approach if editing via Web UI is not appropriate (e.g. you need to change the authentication settings before the first TeamCity server start).

To enable an authentication module, edit the `<TeamCity data directory>/config/auth-config.xml` file on the server machine as follows:

1. Add the `<auth-module>` tag inside the `<{{<auth-modules>}}>` tag inside the `<{{<auth-config>}}>` tag.
2. Specify the `type` attribute for the newly created `<auth-module>` tag.

The following values are supported for the `type` attribute (the values are case-insensitive):

- For credentials authentication modules:
 - Default for Default Authentication
 - NT-Domain for Windows Domain Authentication
 - LDAP for LDAP Authentication
- For HTTP authentication modules:
 - HTTP-Basic for Basic HTTP Authentication
 - HTTP-NTLM for NTLM HTTP Authentication

You can also provide some properties for each authentication module depending on the module type. Each property is specified as the `<property>` tag inside the corresponding `<auth-module>` tag. Each `<property>` tag must contain the `key` attribute with a property key. The property value is specified as the text inside the `<property>` tag.

Also, TeamCity plugins can provide [additional authentication modules](#). The server restart is NOT needed after editing this file. The changes will be reflected in the Web UI.

User Authentication Settings

TeamCity administrator specifies authentication settings for users on their profile page.

Since version 8.0, TeamCity maintains a common user list shared among all the authentication modules. This means that users and their settings remain the same after changing TeamCity server authentication modules if their usernames are the same in the old and new authentication modules. But a user can now have different TeamCity username, LDAP username and Windows domain username. The administrator has to specify the user's LDAP/NT username on his/her profile page to make the user be able to login via LDAP/NT authentication. By default, the TeamCity username is equal to LDAP and Windows domain usernames.

When a user attempts to log in, TeamCity will try all the configured modules 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.

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.

There can also 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.

Example of the relevant `auth-config.xml` file section:

```

<auth-config>
    <!-- List of configured authentication modules in certain order -->
    <auth-modules>
        <auth-module type="Default" />
        <auth-module type="LDAP" />
        <auth-module type="HTTP-Basic" />
        <auth-module type="HTTP-NTLM" />
    </auth-modules>
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
    <guest-login allowed="true" />
</auth-config>

```

Credentials Authentication Modules

Built-in Authentication

Users and their passwords are maintained by TeamCity. New users are added by the TeamCity administrator (in the **Administration** area on the **Users** page) or users are self-registered if the `<property key="freeRegistrationAllowed">true</property>` property is set. 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.

Configuration of `<TeamCity data directory>/config/auth-config.xml`:

```

<auth-config>
    <auth-modules>
        <auth-module type="Default">
            <!-- "freeRegistrationAllowed" property specifies whether the users are allowed to
self-register (true|false). Default is "false". -->
            <property key="freeRegistrationAllowed">false</property>

            <!-- "usersCanChangeOwnPasswords" property specifies whether the users are allowed to
change own passwords for default authentication (true|false).
                Even if it is set to "false" administrator still can change the password of any
user. Default is "true". -->
            <property key="usersCanChangeOwnPasswords">true</property>
        </auth-module>
        <!-- Another authentication modules can be configured here as well -->
    </auth-modules>
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
    <guest-login allowed="true" />
</auth-config>

```

Windows Domain Authentication

See also the [NTLM HTTP Authentication](#) page for information on single sign-on based on Windows domain authentication (transparent login without entering credentials manually).

Configuration of `<TeamCity data directory>/config/auth-config.xml`:

```

<auth-config>
    <auth-modules>
        <auth-module type="NT-Domain">
            <!-- "allowCreatingNewUsersByLogin" property specifies what TeamCity must do when user
specifies correct
                NT domain username and password, but does not yet exist in TeamCity (so on first
successful login via NT domain).
                "true" (default) means to allow login and create such TeamCity user, while "false"
means to deny login for such user. -->
            <property key="allowCreatingNewUsersByLogin">true</property>

            <!-- With "defaultDomain" property specified users can login using just their domain
usernames without domain itself. -->
            <property key="defaultDomain">YourDomain</property>
        </auth-module>
        <!-- Another authentication modules can be configured here as well -->
    </auth-modules>
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
    <guest-login allowed="true" />
</auth-config>

```

To log into TeamCity, users should provide their user name in the DOMAIN\user.name form and their domain password. The <username>@<domain> login name syntax is also supported.

It is also possible to log in using only a username if the domain is specified in the "<property key="defaultDomain">YourDomain</property>" property tag (see above).

Since TeamCity 7.1, when running under Windows, the TeamCity server uses Waffle library for authentication by default. Under Linux, JCIFS library is used for the Windows domain login.

If the allowCreatingNewUsersByLogin property is set to true, a new user account will be created on the first successful login. 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.

The following settings in the <TeamCity data directory>/config/ntlm-config.properties file are obsolete and are not recommended for usage. Please comment them out and report any issues that you have with the configuration.

```

# ntlm.compatibilityMode=true
# teamcity.ntlm.use.jcifs=true

```

The following settings in <TeamCity data directory>/config/ntlm-config.properties have no effect anymore and should be removed:

```

ntlm.defaultDomain=domain

```

jCIFS Library Specific Configuration

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 server restart.

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.

Configuration of <TeamCity data directory>/config/auth-config.xml:

```
<auth-config>
  <auth-modules>
    <auth-module type="HTTP-Basic" />
    <!-- Another authentication modules can be configured here as well -->
  </auth-modules>
  <!-- Welcome message displayed to users on login form -->
  <login-description>Welcome to TeamCity, your team building environment!</login-description>
  <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
  <guest-login allowed="true" />
</auth-config>
```

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 synchronization.

- [Authentication](#)
 - [Manual auth-config.xml Configuration](#)
 - [ldap-config.properties Configuration](#)
 - [Configuring User Login](#)
 - [Active Directory](#)
 - [Advanced Configuration](#)
- [Synchronization](#)
 - [Common Configuration](#)
 - [User Profile Data](#)
 - [User Group Membership](#)
 - [Creating and Deleting User](#)
 - [Username migration](#)
- [Debugging LDAP Integration](#)

LDAP integration might be not trivial to configure, so it might require some trial and error approach to get the right settings.

It is recommended to configure LDAP on a test server before switching it on on the production one. [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.

Authentication

If you need to limit the users who can log into TeamCity to LDAP (e.g. Active Directory) users, you need to configure LDAP settings and enable LDAP authentication for the server (refer to [Authentication Settings](#) page for details).

▼ In case you need to change authentication settings without accessing web UI, there is a way: click to expand

Manual auth-config.xml Configuration

You only need to use this approach if editing via Web UI is not appropriate (e.g. you need to change the authentication settings before the first TeamCity server start).

Configure <TeamCity data directory>/config/auth-config.xml as follows:

```

<auth-config>
    <auth-modules>
        <auth-module type="LDAP">
            <!-- "allowCreatingNewUsersByLogin" property specifies what TeamCity must do when user
            specifies correct
                LDAP username and password, but does not yet exist in TeamCity (so on first
                successful login via LDAP).
                "true" (default) means to allow login and create such TeamCity user, while "false"
                means to deny login for such user. -->
            <property key="allowCreatingNewUsersByLogin">true</property>
        </auth-module>
        <!-- Another authentication modules can be configured here as well -->
    </auth-modules>
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
    <guest-login allowed="true" />
</auth-config>

```

On each user login, authentication is performed by direct login into LDAP with the credentials entered in the login form. TeamCity does not store the user passwords in this case.

If the Allow creating new users on the first login option is selected (allowCreatingNewUsersByLogin property is set to true in the XML file), a new user account will be created on the first successful login. 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.

TeamCity stores user details and settings in its own database. Refer to [Synchronization](#) section below for information on ability to retrieve common user properties from LDAP.

ldap-config.properties Configuration

LDAP connection settings are configured in `<TeamCity data directory>/config/ldap-config.properties` file. See also the `ldap-config.properties.dist` file as an example.



Please note that all the values in the file should be properly [escaped](#)

`ldap-config.properties` file is re-read on any modification so you do not need to restart the server to apply changes in the file. Please make sure you always back up previous version of the file: if you misconfigure LDAP integration, you may no longer be able to log in into TeamCity. Already logged-in users are not affected by the modified LDAP integration settings because users are authenticated only on login.

The mandatory property in `ldap-config.properties` file is `java.naming.provider.url` that configures the server and root DN. The property stores 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 space character.



- For samples of `ldap-config.properties` file please refer to the [Typical LDAP Configurations](#) page.
- The supported configuration properties are documented inside comments of `ldap-config.properties.dist` file.

Configuring User Login



Since TeamCity 5.0 a new login scheme is applied: the user is *first* searched in LDAP, then authenticated using the DN retrieved during the search. It makes the configuration more flexible and also several properties redundant (`formatDN` and `loginFilter`), though in some particular cases using `formatDN` is simpler.

Backwards compatibility: the first step (search) is optional, thus one can login with username entered to the login form (formatted using `formatDN` option, etc.).

You might want to configure the integration so that users enter only the username in TeamCity login form, but LDAP server can require prefixes or suffixes to the name. This can be addressed with `teamcity.auth.formatDN` property (**Since TeamCity 4.5**, previous versions require "`formatDN`" as the property name). The property defines the transformation that is applied to the entered username before the LDAP login is tried.

The property should reference the entered username as `$login$`. Note that TeamCity stores the username for the user in unmodified form as entered by the user. This is necessary to allow mapping of LDAP user back to TeamCity user, refer to the `teamcity.users.username` property description below.

Example:

```
teamcity.auth.formatDN=uid=$login$,ou=people,dc=company,dc=com
```

Some LDAP servers support multiple formats of username. Usually, you would want to restrict the users from entering different variations of the username (otherwise, a new TeamCity user will be created for each variation). The restriction can be configured via the help of `teamcity.auth.loginFilter` property (**Since TeamCity 4.5**, previous versions require "loginFilter" as the name of the property). The property configures regular expression that entered username should comply to.

Example (allow any username):

```
teamcity.auth.loginFilter=.+
```



It is recommended to configure `teamcity.auth.formatDN` and `teamcity.auth.loginFilter` properties so that username entered by user matches one of the LDAP-stored fields for this user.

By default, login format is restricted to a name without "\", "/" and "@" characters. This format applies only **Since TeamCity 4.5**, in previous versions the pattern is to match `DOMAIN\name` patterns.

By default, TeamCity stores the entered login as the username in database. This value is important if you want to synchronize user information, e.g. display name or e-mail, with LDAP data (see the details below). That's why it can be useful to configure the login filter so that user may login with the username stored in LDAP only.

If LDAP server accepts several variations of the login name and they cannot be filtered by `teamcity.auth.loginFilter` (or it is undesirable to limit the login names to particular format), TeamCity provides ability to automatically fetch the username from LDAP. If `teamcity.users.acceptedLogin` property is configured, it will be used to find a user by the entered login name in LDAP and then the user will be stored/matched in TeamCity database with the username that will be retrieved from the attribute defined by `teamcity.users.username` property.

This logic is automatically turned on if `teamcity.users.acceptedLogin` property is defined. In this case `teamcity.users.base` (root DN for users search) and `teamcity.users.username` are mandatory.

Also, you can define a `teamcity.users.login.filter` property with a filter to apply when searching for a user in LDAP. The property may have a `$login$` substring that will be substituted with the actual login name entered by the user on TeamCity login form.

Provided these options are configured, on each login attempt TeamCity will perform a look-up in LDAP searching for the entered login and retrieve the username.

Please note that in certain configurations (for example, with `java.naming.security.authentication=simple`) login information will be sent to the LDAP server in not-encrypted form. For securing the connection you can refer to [corresponding Sun documentation](#). Another option is to configure communications via ldaps protocol.

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 `java.naming` options to the `ldap-config.properties` file: they will be

passed to underlying Java library. 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

LDAP synchronization is only available **since TeamCity 4.5**.

LDAP synchronization allows to:

- Retrieve user's profile data from LDAP
- Update user groups membership based on LDAP groups
- Automatically create and remove users in TeamCity based on information retrieved from LDAP

Periodically, TeamCity fetches data from LDAP and updates users in TeamCity. You can review the last synchronization run statistics and schedule new synchronization in **LDAP Synchronization** section of [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 user credentials which are used by TeamCity to connect to LDAP and retrieve data), `teamcity.users.base` and `teamcity.users.filter` (specify the settings to search for users), and `teamcity.users.username` (the name of LDAP attribute containing the username).

TeamCity should be able to fetch users from LDAP and map them to the existing TeamCity users. Mapping between the LDAP user and TeamCity user is configured by `teamcity.users.username` property.

Example:

```
teamcity.users.username=sAMAccountName
```

User Profile Data

When 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 TeamCity user's profile. If modified in user's profile in TeamCity, the data will no longer be updated from LDAP for the modified fields.

The user's profile synchronization is performed on user creation and also periodically for all users.

All the user fields synchronization properties store the name of LDAP field to retrieve the information from.

The list of supported user settings:

- `teamcity.users.username`
- `teamcity.users.property.displayName`
- `teamcity.users.property.email`
- `teamcity.users.property.plugin:notifier: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 web UI and then listing the properties via [REST API#Users](#).

User Group Membership

TeamCity can automatically update users membership in groups based on the LDAP-provided data.

You will need to create groups in TeamCity manually and then specify the mapping on LDAP groups in [TeamCity data directory](#) /config/ldap-mapping.xml file. Use the [ldap-mapping.xml.dist](#) file as an example.

In the file you need to specify correspondence between TeamCity user group (determined by group id) and LDAP group (specified by group DN). For user group membership to work you also need to set the following properties in ldap-config.properties file: teamcity.options.groups.synchronize (enables user group synchronization), teamcity.groups.base and teamcity.groups.filter (specify where and how to find the groups in LDAP) and teamcity.groups.property.member (specifies the LDAP attribute holding the members of the group).

On each synchronization run, TeamCity updates the membership of users in groups that are configured in the mapping. Please note that TeamCity synchronizes membership only for users residing directly in the groups. Subgroups are not processed.

If either LDAP group or TeamCity that is configured in the mapping is not found, an error is reported. You can review the errors occurred during last synchronization run in "LDAP Synchronization" section of [server settings](#).

See also [example settings](#) for Active Directory synchronization.

Creating and Deleting User

TeamCity can automatically create users in TeamCity, if they are found in one of the mapped LDAP groups.

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 had previously logged in using 'DOMAIN\user' name, thus the string 'DOMAIN\user' had been stored in TeamCity as the username. In order to synchronize the data with LDAP 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 to find the 'DOMAIN\user' username when login with just 'user', and replace it with either 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 on [here](#)

Debugging LDAP Integration

Internal LDAP logs are stored in `logs/teamcity-ldap.log` file in 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, please use "debug-ldap" logging preset. See [TeamCity Server Logs](#) for details.

Typical LDAP Configurations

This page contains samples of `ldap-config.properties` file for different configuration cases.

- Basic LDAP Login
- Basic LDAP Login for Users in Specific LDAP Group Only
- Active Directory With User Details Synchronization
- Active Directory With Group Synchronization

Basic LDAP Login

The backup LDAP server is specified. Provided users can log in to LDAP with "EXAMPLE\Username", they log in to TeamCity also as "EXAMPLE\Username", the username stored in TeamCity is "Username".

```

# 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

# Allow to log in with 'EXAMPLE\username', but cut off 'EXAMPLE' in TeamCity username.
teamcity.auth.loginFilter=EXAMPLE\\\\\\\$+
teamcity.users.login.capture=EXAMPLE\\\\\\(.*)

# No synchronization, just login.
teamcity.options.users.synchronize=false
teamcity.options.groups.synchronize=false

```

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.

```

java.naming.provider.url=ldap://example.com:389/DC=example,DC=com

# Windows username for user to browse LDAP
java.naming.security.principal=RealUsername
# Windows password for user to browse LDAP
java.naming.security.credentials=User'sPaSsWorD

# Root node containing all the LDAP users (full entry DN is "CN=Users,DC=example,DC=com")
teamcity.users.base=CN=Users

# 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=
retrieving TeamCity username form the "sAMAccountName" LDAP entry attribute
teamcity.users.username=sAMAccountName

# Allow only username part without domain
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.



Fix to eliminate double users creation (EXAMPLE/Bob and Bob)

```

java.naming.provider.url=ldap://example.com:389/DC=example,DC=com

# Login using 'sAMAccountName' value.
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)

# LDAP credentials for TeamCity plugin.
java.naming.security.principal=CN=teamcity,CN=Users,DC=example,DC=com
java.naming.security.credentials=secret

# User synchronization: on, synchronize display name and e-mail.
teamcity.options.users.synchronize=true
teamcity.users.base=CN=users
teamcity.users.filter=(objectClass=user)
teamcity.users.username=sAMAccountName
teamcity.users.property.displayName=displayName
teamcity.users.property.email=email

# Group synchronization: disabled.
teamcity.options.groups.synchronize=false

```

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

# Allow to enter anything, but after that format it into 'EXAMPLE\login'.
teamcity.auth.formatDN=EXAMPLE\\$login$

# LDAP credentials for TeamCity plugin.
java.naming.security.principal=teamcity
java.naming.security.credentials=secret

# Synchronize both users and groups. Remove obsolete TeamCity users, but don't create new ones
automatically.
teamcity.options.users.synchronize=true
teamcity.options.groups.synchronize=true
teamcity.options.createUsers=false
teamcity.options.deleteUsers=true
teamcity.options.syncTimeout=3600000

# Search users from the root: 'DC=example,DC=com'.
teamcity.users.base=
teamcity.users.filter=(objectClass=user)
teamcity.users.username=sAMAccountName

# Search groups from 'CN=groups,DC=example,DC=com'.
teamcity.groups.base=CN=groups
teamcity.groups.filter=(objectClass=group)
teamcity.groups.property.member=member

```

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, Kerberos or Negotiate HTTP authentications.

Generally, it allows users to log in into the TeamCity server 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

"NTLM HTTP" module is configured on the **Administration | Authentication** page under the "HTTP authentication modules" section.

▼ There is also an older way to configure the settings directly in the settings file.

Configure `<TeamCity data directory>/config/auth-config.xml` as follows:

```

<auth-config>
    <auth-modules>
        <auth-module type="HTTP-NTLM">
            <!-- "allowProtocols" property specifies the enabled protocols. Default is "ntlm".
            Possible protocols are "ntlm", "negotiate", "kerberos".
                Protocol names are case insensitive and can be separated by comma, semicolon or
                spaces. Order is not important. -->
            <property key="allowProtocols">ntlm,negotiate</property>

            <!-- "forceProtocols" property specifies the protocols, which TeamCity uses to force
            NTLM HTTP authentication (i.e. to initiate NTLM HTTP authentication process
                instead of redirecting to login page when user is not authenticated). Default is
                "" (empty). Possible protocols are "ntlm", "negotiate", "kerberos".
                    Protocol names are case insensitive and can be separated by comma, semicolon or
                    spaces. Order is not important. -->
            <property key="forceProtocols"></property>

            <!-- "allowCreatingNewUsersByLogin" property specifies what TeamCity must do when user
            succeeded to login
                via NTLM HTTP authentication, but does not yet exist in TeamCity (so on first
                successful login via NTLM HTTP authentication).
                    "true" (default) means to allow login and create such TeamCity user, while "false"
                    means to deny login for such user. -->
            <property key="allowCreatingNewUsersByLogin">true</property>
        </auth-module>
        <!-- Another authentication modules can be configured here as well -->
    </auth-modules>
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
    <guest-login allowed="true" />
</auth-config>

```

If the `allowCreatingNewUsersByLogin` property is set to `true`, a new user account will be created on the first successful login. 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.



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.

Since TeamCity 8.0, NTLM HTTP authentication does not require to use [Windows domain authentication](#) anymore. Now you can use it with any login module (to use it with some login modules, you might need to specify your Windows domain username on your profile page).

The protocols supported include NTLMv1, NTLMv2, Kerberos and Negotiate.

Requirements

1. The authenticating user should be logged in to the workstation with the domain account that is to be used for the authentication.
2. The user's web browser should support NTLM HTTP authentication.

Enabling NTLM HTTP Authentication

After NTLM HTTP authentication module is configured, users will see a link on 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 "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 usual case an unauthenticated user will be simply redirected to the TeamCity login page.

Since version 7.1.1 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 necessity 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. NTLM HTTP authentication module (as well as Windows domain credentials authentication module) does not have such functionality, so it can be possible for some users to login using Windows domain account even if they are not allowed to login via LDAP. To solve this problem, you should set `Allow creating new users` on the first login option for the corresponding authentication module.

With this property set, a user will be able to login via 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 custom NT domain username specified on the user's profile page.

Configuring client

According to 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

To enable guest login to TeamCity:

1. Navigate to **Administration** page and click **Authentication**.
2. Select **Allow to login as a guest user** option.
The **Login as a guest user** link appears on the **Login** page.

To customize which projects guest user has access to:

- Click the **Configure guest user roles** link to configure the **roles**.

See also:

Concepts: User Account | Role and Permission
Administrator's Guide: Configuring Authentication Settings | Managing Users and User Groups

Changing user password with default authentication scheme



This is obsolete. **Since in TeamCity 8.0** log in as Super User to reset the password for any user via UI.

The following procedure is suitable for default authentication scheme and default database (HSQLDB) only.

To change user password:

1. Shutdown server
2. Switch to the <TeamCity home>/webapps/ROOT/WEB-INF/lib directory
3. Invoke the following command:
Windows platform:

```
java -cp server.jar;common-api.jar;commons-codec-1.3.jar;util.jar;hsqldb.jar;log4j-1.2.12.jar  
ChangePassword <username> <new password> <TeamCity data directory>
```

Unix platform:

```
java -cp server.jar:common-api.jar:commons-codec-1.3.jar:util.jar:hsqldb.jar;log4j-1.2.12.jar  
ChangePassword <username> <new password> <TeamCity data directory>
```

You can skip the <TeamCity Data Directory> option, if you are using default path for TeamCity data files: <user home>/.BuildServer

See also:

Concepts: User Account | Role and Permission
Administrator's Guide: Configuring Authentication Settings | Managing Users and User Groups

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
 - Server is Run Via Shell Script
 - TeamCity Server is Run as Windows Service

TeamCity internal properties

TeamCity has some properties that are not exposed in the UI and are meant for debug use only. If you need to set such a property (e.g. asked by TeamCity support), you can:

- either set it as a `-D<name>=<value>` JVM option (see below)
- or add these TeamCity-specific properties into 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.

Once you create the file, you can 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**.

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 [Server is Run Via Shell Script](#).

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.

Server is Run Via Shell Script

If you run the server using the `runAll` or `teamcity-server` scripts or as a Windows service with TeamCity 7.1 and above, you can 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 -XX:MaxPermSize=270m`)
- `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. You might need to reboot the machine after the environment change for the changes to have effect.

TeamCity Server is Run as Windows Service

TeamCity 7.1

Since [TeamCity 7.1](#), you will need to set environment variables as described in [Server is Run Via Shell Script](#) to affect TeamCity run as a service.

TeamCity 7.0 and previous versions

For TeamCity 7.0 and previous versions please use the instructions below:

To edit JVM server options run Tomcat's service configuration editor by executing the command

```
tomcat7w.exe //ES//TeamCity
```

in `<TeamCity home>\bin` directory and then edit the Java Options on the Java tab (for more information see [Tomcat 6 documentation](#)).

To change heap memory dedicated to the JVM, change the "Maximum memory pool" setting.

See also:

Concepts: TeamCity Data Directory

Administrator's Guide: Configuring Build Agent Startup Properties

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

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

This document describes how to configure various TeamCity server clients to use HTTPS for communicating with the server. 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. See how to do this for Tomcat here: <http://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html>. See also a feature request: [TW-12976](#).

It's also a common approach to setup a middle server like Apache that will handle HTTPS but will use Tomcat to handle the requests. In the setup please make sure that the middle server/proxy has correct URL rewriting configuration, see also [How To...#Set Up TeamCity behind a proxying server](#) section.

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 to use `https://` links to the TeamCity server instead of `http://`.

If your certificate is not valid:

- To enable HTTPS connections from 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 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), See the [section below](#) for configuring the JVM installation used by the connecting application.

Configuring JVM

Configuring JVM for authentication with server certificate

If your certificate is ***valid** (i.e. it was signed by a well known Certificate Authority like Verisign), then the Java clients should work with HTTPS without any additional configuration.

If your certificate is not valid:

- To enable HTTPS connections from Java clients:
 - save server certificate to a file
 - locate the JRE used by the client
 - If there is a JDK installed (like for IntelliJ IDEA), <path to JRE installation> should be <path to used JDK>/jre
 - For TeamCity agent installed under Windows, use "<agent installation path>/jre" as "<path to JRE installation>".
 - import the server certificate into the JRE installation keystore using keytool program:

```
keytool -importcert -file <cert file> -keystore <path to JRE installation>/lib/security/cacerts
```

 Note: `-importcert` option is only available starting from Java 1.6. Please use keytool from Java 1.6+ to perform these commands.

By default, Java keystore is protected by password: "changeit"

Configuring JVM for authentication with client certificate

Importing client certificate

If you need to use 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 Java keystore and supply the keystore to the JVM used by the connecting process.

1. If you have your certificate in **p12** file, you can use the following command to convert it to a Java keystore. Make sure you use `keytool` from JDK 1.6+ because 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 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 JVM and if `keypass` is different, one may get error: "`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 [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-----  
MIIGUjCCBDggAwIBAgIEAKmKxzANBgkqhkiG9w0BAQQFADBwMRUwEwYDVQQDEwxK  
...  
-----END CERTIFICATE-----
```

Let's assume it's 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 bin\idea.exe.vmoptions file (one option per line).

For the TeamCity build agent, see [agent startup properties](#).

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.

Prior to TeamCity 8.1, the thresholds can be changed by modifying the following internal properties:



Note that the size is to be specified in kilobytes.

Property	Default	Description
teamcity.diskSpaceWatcher.softThreshold	50000 (50 Mb)	displays a warning
teamcity.diskSpaceWatcher.threshold	10000 (10 Mb)	pauses the build queue

Since TeamCity 8.1, the teamcity.diskSpaceWatcher.softThreshold property is removed. The thresholds can be changed by modifying the following internal properties:

Property	Default	Description
teamcity.diskSpaceWatcher.threshold	500000 (500 Mb)	displays a warning
teamcity.pauseBuildQueue.diskSpace.threshold	50000 (50 Mb)	pauses the build queue

See also:

[Administrator's Guide: Free disk space on agent](#)

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](#)
 - [General Logging Configuration](#)
 - [Changing Logging Settings](#)
- [Reading Logs](#)

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:

teamcity-server.log	General server log
teamcity-activities.log	Log of user-initiated and main build-related events
teamcity-vcs.log	Log of VCS-related activity
teamcity-notifications.log	Notifications-related log
teamcity-clouds.log	(off by default) Cloud-integration (Amazon EC2) -related log
teamcity-sql.log	(off by default) Log of SQL queries, see details
teamcity-http-auth.log	(off by default) Log with messages related to NTLM 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-maintenance.log	(off by default) logs of back-up/ restore/ migration performed with maintainDB tool

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 server logs right from TeamCity web UI under **Administration | Diagnostics | Server Logs**.

Also, users with System Administrator role can change active logging preset for the server logs for the server logs under **Administration | Diagnostics** page, **Troubleshooting, Debug logging** subsection. Choosing a preset will change logging configuration until the server is restarted (see [TW-14313](#)).

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.

General Logging Configuration

By default TeamCity searches for log4j configuration in a `../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 `../logs` directory by default.

The configuration options values can be changed via corresponding "log4j.configuration" and "teamcity_logs" JVM options or [internal properties](#). For example: `log4j.configuration=file:../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.

Changing Logging Settings

If you want to fine-tune the log4j configuration, you can edit `teamcity-server-log4j.xml`. If the server is running, it 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).

The logs are rotated by default. When debug is enabled, it makes sense to [increase the number](#) of rotating log files. While doing so, please ensure there is sufficient free disk space available.

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" />
  ...
  ...
```

To save more rolling files, tweak "value" attribute of "maxBackupIndex" element:

```
<appender ...>
  <param name="maxBackupIndex" value="10" />
  ...
  ...
```

For 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.

See also:

[Administrator's Guide: Viewing Build Agent Logs](#)
[Troubleshooting: Reporting Issues](#)

Build Agents Configuration and Maintenance

In this section:

- [Agent Pools](#)
- [Configuring Build Agent Startup Properties](#)
- [Viewing Agents Workload](#)
- [Viewing Build Agent Details](#)
- [Viewing Build Agent Logs](#)

Agent Pools

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.

Project builds can be run only on build agents from pools assigned to the project. 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. Also with agent pools it is easier to calculate required agents capacity.

You can find all agent pools configured in TeamCity at the **Agents | Pools** tab. To be able to add\remove pools, 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](#).

Another permission, "Change agent pools associated with project", by default granted to the users assigned a project administrator role, allows users to assign and un-assign projects and agents to/from pools. A user can assign and un-assign projects and agents from a pool only if he/she has "Change agent pools associated with project" for all projects associated with the pool.

To create a new agent pool, you only need to specify its name; to populate it with agents, just 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.

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: 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 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>
```



Please ensure to 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.

The screenshot shows the TeamCity interface with the 'Agents' tab selected. The 'Matrix' tab is active. The matrix displays build times and resource usage for various agents and build configurations over a specified time range. The matrix is color-coded by resource usage, with green indicating higher usage and yellow indicating lower usage.

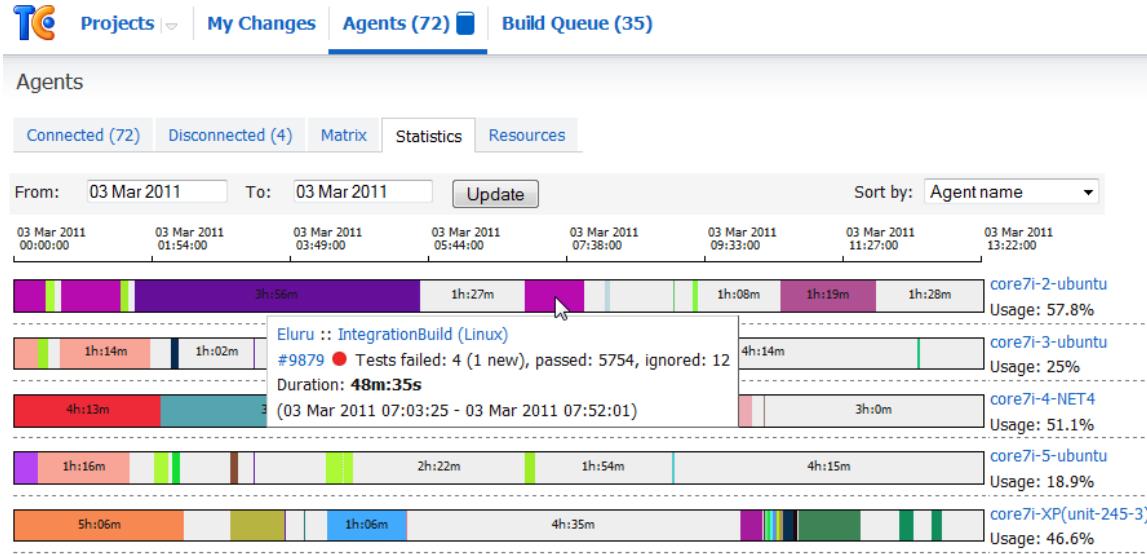
Build Configuration / Agent	Total	Build Times (m:ss)								Dot
		unit-011 (FreeB...)	unit-132	NetFX 2.0 + VS ...	w7-esxi4-10	w7-esxi4-11	unit-111	w7-esxi4-1		
Total	2345h:35m	86h:08m	47h:38m	44h:43m	44h:14m	43h:18m	42h:58m	42h:41m		
Eluru	671h:21m	28,6%	84h:35m		17h:06m	6h:34m	10h:52m	10h:56m	17h:35m	
IntegrationBuild (FreeBSD)	84h:35m	3,6%	84h:35m							
IntegrationBuild (Oracle)	67h:38m	2,9%					2h:48m	4h:24m		
IntegrationBuild (HSQLDB)	44h:53m	1,9%					1h:15m			
IntegrationBuild (.NET Runners...)	44h:33m	1,9%			4h:21m	6h:43m	4h:25m	4h:49m		
IntegrationBuild TFS	44h:16m	1,9%				3h:28m				
IntegrationBuild (MySQL)	43h:11m	1,8%								
IntegrationBuild (Linux)	42h:13m	1,8%								
IntegrationBuild (Java Based R...	42h:02m	1,8%		20m:20s						
IntegrationBuild (PSQL)	32h:40m	1,4%								
IntegrationBuild (Solaris)	28h:05m	1,2%								
IntegrationBuild (MS SQL)	27h:57m	1,2%								
Inspections & Duplicates	22h:09m	0,9%		15h:54m	2h:01m					
Compile	18h:42m	0,8%								
IntegrationBuild (Sybase)	17h:48m	0,8%								

1. Select the time range and click **Update** to reload the matrix.
2. Click to navigate to [agent's details](#).

3. Total build duration during specified time.
4. Total build duration for particular build configuration, for builds that were run during specified time range.
5. Total duration of the particular build configuration builds that were run on the particular build agent during specified time period.
6. Build agent is compatible with the build configuration, but no builds were run during specified time range.
7. 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 it's name, or navigate to the **Agents** page, find it in the list of connected, disconnected or authorized agents and click its name.

For each connected agent TeamCity provides following details:

- Agent's status. [Learn more about enabled/disabled agents](#).
- Agent's name, host name and IP address, port number, operating system, agent's CPU rank and current version number.
- Information about build currently running on the agent with the link to the [build results](#).
- Options to:
 - [Clean sources](#)
 - Open Remote Desktop. Available for agents running on the Windows operating system, if RDP client is installed and registered on the agent.
 - Reboot agent machine. Available to users who have "Reboot build agent machines" permissions. See [below](#) for additional configuration of the feature.
 - Dump threads on agent.
- Compatible and incompatible build configurations with the reason of incompatibility.
- Build runners supported by build agent.
- Agent parameters including system properties, environment variables and configuration parameters. Refer to the [Configuring Build Parameters](#) page for more information on different types of parameters.

Configuring Agent Reboot Command

Agent reboot is performed by executing an OS-specific command.

Under certain circumstances the command might need customization specific to the OS environment. If the agent reboot does not work by default, you can add `teamcity.agent.reboot.command` configuration parameter in `buildagent.properties` 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
```

See also:

Concepts: [Build Agent | Run Configuration Policy](#)

Installation and Upgrade: [Installing and Running Additional Build Agents](#)

Viewing Build Agent Logs

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 the logs in UI, on [Agent Logs](#) tab for an agent.

Log Files

TeamCity uses **Log4J** for internal logging of events. Default build agent Log4J configuration file is `<agent home>/conf/teamcity-agent-log4j.xml`

See the comments in the file for enabling 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.

File name	Description
<code>teamcity-agent.log</code>	General build agent log
<code>teamcity-build.log</code>	<code>stdout</code> and <code>stderr</code> output of builds run by the agent
<code>upgrade.log</code>	log of the build agent upgrade (logged by the upgrading process)
<code>launcher.log</code>	log of the agent's monitoring/launching process
<code>wrapper.log</code>	(only present when the agent is run as Windows service or by Java Service Wrapper) output of the process build agent launching process

You can configure location of the logs by altering the value of `teamcity_logs` property (passed to JVM via `-D` option).

You can also change Log4J configuration file location by changing value of `log4j.configuration` property.

See corresponding documentation section on how to pass the options.

For additional option on logging tweaking please consult [TeamCity Server Logs#log4jConfiguration](#) section.

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" />`".

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 if amount of available memory is too low. There are several warning types reported:

Low pool memory

Is reported when memory usage in a single memory pool is more than 90% after garbage collection. High server activity could cause such memory usage.

Low PermGen memory

Is reported when more than 95% of PermGen memory pool is used. Usually high PermGen usage is not a problem as it mostly isn't used for data. However PermGen overflow could crash TeamCity server so it's recommended to increase PermGen size. Contact with TeamCity team if the warning is still showing after pool increase.

Low total memory

Is reported when more than 90% of total memory is in use for the last 5 minutes and more than 20% of CPU resources are being spent on garbage collection. Lasting memory lack could result in performance degradation and server unstability as well.

Heavy GC overload

Is reported when memory cleaning takes in average more than 50% of CPU resources. It usually means really heavy problems with memory resulting in high performance degradation.

Customization

Several [internal properties](#) could 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 to set up minimal warning threshold. Affects all tracked memory pools except PermGen

`teamCity.memoryUsageMonitor[<Pool name>].warningThreshold` could be used to modify single memory pool threshold (including PermGen). Spaces should be escaped or changed to '_' signs

`teamCity.memoryUsageMonitor.gcWarningThreshold` allows to set up 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](#).

Administration 3 Disk Usage					
Project Artifacts Directory	Total free disk space: 447 MB Project: 289 MB (size: 2.2 GB since artifacts, 4.0 KB pending, archive) (133 files, 163 files by type, 17 files by size, 46 projects)				
Build Configuration	Path	Name	Type	Size	Free
		Root		1,089,500 KB	8,953,034 KB
	Project A	Project A - Main Project	Build Configuration	88,197 KB	959,302 KB
	Project A	Project A - Sub-Project	Build Configuration	38,197 KB	959,302 KB
	Project B	Project B - Main Project	Build Configuration	91,890 KB	91,890 KB
	Project B	Project B - Sub-Project	Build Configuration	81,890 KB	91,890 KB
	Project C	Project C - Main Project	Build Configuration	81,890 KB	77,900 KB
	Project C	Project C - Sub-Project	Build Configuration	1,890 KB	77,900 KB
	Project D	Project D - Main Project	Build Configuration	896 KB	896 KB
	Project D	Project D - Sub-Project	Build Configuration	896 KB	896 KB
	Project E	Project E - Main Project	Build Configuration	896 KB	896 KB
	Project E	Project E - Sub-Project	Build Configuration	896 KB	896 KB
	Project F	Project F - Main Project	Build Configuration	896 KB	896 KB
	Project F	Project F - Sub-Project	Build Configuration	896 KB	896 KB
	Project G	Project G - Main Project	Build Configuration	896 KB	896 KB
	Project G	Project G - Sub-Project	Build Configuration	896 KB	896 KB
	Project H	Project H - Main Project	Build Configuration	896 KB	896 KB
	Project H	Project H - Sub-Project	Build Configuration	896 KB	896 KB
	Project I	Project I - Main Project	Build Configuration	896 KB	896 KB
	Project I	Project I - Sub-Project	Build Configuration	896 KB	896 KB
	Project J	Project J - Main Project	Build Configuration	896 KB	896 KB
	Project J	Project J - Sub-Project	Build Configuration	896 KB	896 KB
	Project K	Project K - Main Project	Build Configuration	896 KB	896 KB
	Project K	Project K - Sub-Project	Build Configuration	896 KB	896 KB
	Project L	Project L - Main Project	Build Configuration	896 KB	896 KB
	Project L	Project L - Sub-Project	Build Configuration	896 KB	896 KB
	Project M	Project M - Main Project	Build Configuration	896 KB	896 KB
	Project M	Project M - Sub-Project	Build Configuration	896 KB	896 KB
	Project N	Project N - Main Project	Build Configuration	896 KB	896 KB
	Project N	Project N - Sub-Project	Build Configuration	896 KB	896 KB
	Project O	Project O - Main Project	Build Configuration	896 KB	896 KB
	Project O	Project O - Sub-Project	Build Configuration	896 KB	896 KB
	Project P	Project P - Main Project	Build Configuration	896 KB	896 KB
	Project P	Project P - Sub-Project	Build Configuration	896 KB	896 KB
	Project Q	Project Q - Main Project	Build Configuration	896 KB	896 KB
	Project Q	Project Q - Sub-Project	Build Configuration	896 KB	896 KB
	Project R	Project R - Main Project	Build Configuration	896 KB	896 KB
	Project R	Project R - Sub-Project	Build Configuration	896 KB	896 KB
	Project S	Project S - Main Project	Build Configuration	896 KB	896 KB
	Project S	Project S - Sub-Project	Build Configuration	896 KB	896 KB
	Project T	Project T - Main Project	Build Configuration	896 KB	896 KB
	Project T	Project T - Sub-Project	Build Configuration	896 KB	896 KB
	Project U	Project U - Main Project	Build Configuration	896 KB	896 KB
	Project U	Project U - Sub-Project	Build Configuration	896 KB	896 KB
	Project V	Project V - Main Project	Build Configuration	896 KB	896 KB
	Project V	Project V - Sub-Project	Build Configuration	896 KB	896 KB
	Project W	Project W - Main Project	Build Configuration	896 KB	896 KB
	Project W	Project W - Sub-Project	Build Configuration	896 KB	896 KB
	Project X	Project X - Main Project	Build Configuration	896 KB	896 KB
	Project X	Project X - Sub-Project	Build Configuration	896 KB	896 KB
	Project Y	Project Y - Main Project	Build Configuration	896 KB	896 KB
	Project Y	Project Y - Sub-Project	Build Configuration	896 KB	896 KB
	Project Z	Project Z - Main Project	Build Configuration	896 KB	896 KB
	Project Z	Project Z - Sub-Project	Build Configuration	896 KB	896 KB

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:

Configuration: Configuration - Main - Main Build					
Builds in disk: 418, with 26 builds in 0.0d					
	Average duration: 170.20 MS, step: 0.24 MS				
	Median duration: 169.25 MS, step: 0.24 MS				
	Max duration: 303.00 MS, step: 0.24 MS				
	Total disk usage: 239,834 KB, step: 1 KB				
	Total build duration: 170.20 MS, step: 0.24 MS				
	Clean up disk				
	Clean up log				
Builds for Configuration: Configuration - Main - Main Build					
	Builds in disk: 418, with 26 builds in 0.0d				
	Average duration: 170.20 MS, step: 0.24 MS				
	Median duration: 169.25 MS, step: 0.24 MS				
	Max duration: 303.00 MS, step: 0.24 MS				
	Total disk usage: 239,834 KB, step: 1 KB				
	Total build duration: 170.20 MS, step: 0.24 MS				
	Clean up disk				
	Clean up log				

Server Health

Since TeamCity 8.0, the **Server Health** report is available containing 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.

On this page:

- [Scope and Severity](#)
- [Visibility Options](#)
- [Issue Categories](#)
 - [Critical Errors](#)
 - [Database Related Problems](#)
 - [Possible Frequent Clean Checkout](#)
 - [Custom Checkout Directory](#)
 - [Build Files Cleaner \(Swabra\) Settings](#)
 - [Configurations with Large Build Logs](#)
 - [VCS Root Related Problems](#)

- Unused VCS Roots
- Similar VCS roots
- Similar VCS Root Usages (AKA instances)
- Trigger Rules for Unattached VCS roots
- Redundant Trigger
- Unused Build Agents
- Suggested Settings
- Extensibility

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 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 the display, use the **Show** option.

The visibility changes will be listed on the **Audit** page in the **Administration** area.

Issue Categories

Currently, TeamCity alerts you to the following configuration issue categories:

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.

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 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.

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.

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

A rule of a [VCS Trigger](#) or [Schedule Trigger](#) can reference a VCS root which is not attached to the build configuration. This is often a bug in the settings.

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.

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

Suggested Settings

Since TeamCity 8.1, 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.

Managing Projects and Build Configurations

In this section:

- Creating and Editing Projects
- Creating and Editing Build Configurations
- Working with Meta-Runner
- Copy, Move, Delete Build Configuration
- Working with Feature Branches
- Triggering a Custom Build
- Ordering Build Queue
- Muting Test Failures
- Changing Build Status Manually
- Customizing Statistics Charts
- Archiving Projects

See also:

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

Creating and Editing Projects

To create a new project:

On the **Administration | Projects** page select one of the options:

- Create Project from URL
- Create Project

Create Project from URL

The option is available **since TeamCity 8.1**.

1. Click the **Create subproject from URL** button.
2. On the **Create New Project from URL** page, specify the project settings:

Setting	Description
Parent Project	Select the parent project form the drop-down.
Repository URL	A VCS repository URL. 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 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 will suggest build triggers, failure conditions and build features. Depending on the build configuration settings, it can suggest some additional configuration options.

Create Project

1. Click the **Create subproject** button.
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 ID of the project
Description	Optional description for the project.

- 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.

- Create build configurations (select build settings, configure VCS settings, and choose build runners) for the project.
- Assign build configurations to specific build agents.

See also:

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

Creating and Editing Build Configurations

On this page:

- [Creating New Build Configuration](#)
- [Creating Build Configuration Template](#)
- [Creating Build Configuration From Template](#)
- [Creating Maven Build Configuration](#)
 - [Maven SCM URL Format](#)
 - Examples of the SCM URL for Supported SCM Providers
 - [Configuring Settings](#)

TeamCity provides several ways to create a new **build configuration** for a project. You can:

- [Create a new build configuration](#).
- [Create a build configuration template, and then create a build configuration based on the template](#).
- [Create a Maven build configuration by importing settings from pom.xml](#)

Creating New Build Configuration

- On the **Administration > Projects** page, select the desired project in the list. (Alternatively, select the project from the **Projects** popup, and click the **Edit Project Settings** link on the right). On the project settings page, click **Create build configuration**.
- Specify general settings for the build configuration, then click the **VCS settings** button.
- [Create/edit VCS roots and specify VCS-specific settings](#), then click the **Add Build Step** button.
- On the **New Build Step** page, configure the first build step by selecting a desired build runner from the drop-down list, and [defining its settings](#). 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 are not executed.
- Additionally, configure [build triggering options, dependencies, properties and variables](#) and [agent requirements](#).

Creating Build Configuration Template

Creating a [build configuration template](#) is similar to creating a new configuration. On the Project settings page, 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 using the **Extract Template** option.

Creating Build Configuration From Template

To create a build configuration based on a template, on the Project settings page locate the desired template and click **Edit**. Then, click the **Create Build Configuration** button on the right side-bar of the page and specify a name for the new configuration. The settings set up in the template cannot be edited in a configuration created from this template. If you want to change them, modify them in the template's settings.

However, note that this will affect all configurations based on this template.

Alternatively, you can create a build configuration based on a template by clicking **Create build configuration** and selecting **Create from template** on the project settings page.

Creating Maven Build Configuration



Deprecated. Since Teamcity 8.1 use [create project from url](#)

To create a new build configuration automatically from the `POM.xml` file, on the Project settings page click **Create Maven build configuration**. Specify the following options:

Option	Description
POM file	Specify the POM file to load configuration data from. You can either provide an URL to the <code>pom.xml</code> file in a repository or upload <code>pom.xml</code> file from your local working PC. The URL to the POM file should be specified in the Maven SCM URL format .
Username	Username to access the VCS repository.
Password	Password to access the VCS repository.
Goals	Provide goals for the Maven build runner to be executed with the new configuration.
Triggering	Select the check box to set automatic build triggering on snapshot dependency.



TeamCity reads the name and the VCS root URL parameters for the new build configuration from the provided POM file. For non-Maven projects, if there's no VCS root URL provided in the `pom.xml`, then the process will be aborted with an error.

When a new Maven configuration is created, any further configuring is similar to editing an ordinary build configuration.

Refer to Maven documentation on the SCM Implementation for the following supported version control systems:

- Subversion: <http://maven.apache.org/scm/subversion.html>
- Perforce: <http://maven.apache.org/scm/perforce.html>
- StarTeam: <http://maven.apache.org/scm/starteam.html>

Maven SCM URL Format

The general format for a SCM Url is

```
scm:<scm_provider><delimiter><provider_specific_part>
```

As a delimiter, use either a colon ':' or, if you use a colon for one of the variables (for example, a windows path), you can use a pipe '|'.

For more information, refer to the official Maven SCM documentation page.

In TeamCity you can use simplified SCM URL format:

- If the protocol defined in the provider-specific part unambiguously identifies the SCM-provider, then the `scm:<scm_provider>:` prefix of the URL can be omitted. For instance, the "`scm:startteam:startteam://host.com/trunk/pom.xml`" URL will be valid in the "`startteam:/host.com/trunk/pom.xml`" format. In any other case, for example if HTTP protocol is used for SVN repository, then the SCM-provider must be specified explicitly:
`svn:http://svn.host.com/trunk/project/pom.xml`
- The `scm` prefix can be omitted, or can be replaced with `vcs` prefix.

Examples of the SCM URL for Supported SCM Providers

The following URLs will be considered equal:

- Subversion:

```
scm:svn:svn://svn.company.com/project/trunk/pom.xml  
or  
vcs:svn:svn://svn.company.com/project/trunk/pom.xml  
or  
svn:svn://svn.company.com/project/trunk/pom.xml  
or  
svn://svn.company.com/project/trunk/pom.xml
```

**Note**

Please note that "svn:<http://svn.company.com/project/trunk/pom.xml>" URL does not equal to the "<http://svn.company.com/project/trunk/pom.xml>".

- StarTeam:

```
scm:starteam:starteam://host.com/project/view/pom.xml  
or  
starteam:starteam://host.com/project/view/pom.xml  
or  
starteam://host.com/project/view/pom.xml
```

- Perforce:

```
scm:perforce:user@//main/myproject/pom.xml  
or  
perforce:user@//main/myproject/pom.xml
```

Configuring Settings

Configuring settings of a build configuration is described on the following pages:

- Configuring General Settings
- Configuring VCS Settings
- 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

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	An ID of the build configuration unique across all build configurations and templates in the system.
Description	An optional description for the build configuration.

Build Number Format	This value is assigned to the build number. For more information, refer to the Build Number Format 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 Artifact Paths section below.
Build Options	<p>Additional options for this build configuration. For more information, refer to the following sections below:</p> <ul style="list-style-type: none"> • Explicit Paths • Paths Patterns • Hanging Build Detection • Enable Status Widget <ul style="list-style-type: none"> • HTML Status Widget • Limit Number of Simultaneously Running Builds

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 Build counter field.
%build.vcs.number.<VCS_root_name>%	the revision used for the build of the VCS root with <VCS_root_name> name. Read more on the property.
%property.name%	a value of the build property with corresponding name. All the Predefined Build Parameters are supported (including Reference-only server properties).



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 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. [Read more](#) about build artifacts.

On the **General Settings** page of the build configuration, you can specify explicit paths to build artifacts or patterns to define artifacts of a build.

Explicit Paths

If you know the names of your build artifacts and their exact paths, you can specify them here. **Since TeamCity 8.1**, if you have a build finished on an agent, you can browse the checkout directory and select artifacts from a tree. TeamCity will place the paths to them into the input field.

Paths Patterns

The **Artifact Paths** field also supports newline- or comma-delimited patterns of the following format:

```
file_name|directory_name|wildcard [ => target_directory|target_archive ]
```

The format contains:

- **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.

- `target_directory` — the directory in the resulting build's artifacts that will contain the files determined by the left part of the pattern. This path is a relative one with the root being the root of the build artifacts.
- `target_archive` — 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`, `.jar`, `.tar.gz`, or `.tgz`.

The source file being absolute paths are supported, but those configurations are not recommended in favor of paths relative to the [build checkout directory](#).

An 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 must not be absolute. Non-relative paths will produce errors during the build.



Note, that same `target_archive` name can be used multiple times, for example:

```
*/*.*html => report.zip  
*/*.*css => report.zip!/css/  
Relative paths inside zip archive can be used, if needed.
```

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.

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.

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.

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.

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, preferably 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
```

Configuring VCS Settings

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: [ClearCase](#), [CVS](#), [Git](#), [Mercurial](#), [Perforce](#), [StarTeam](#), [Subversion](#), [Team Foundation Server](#), [SourceGear Vault](#), [Visual SourceSafe](#).

Setting up VCS parameters is one of the mandatory steps during creating a build configuration in TeamCity.
On the 2nd page of the build configuration ([Version Control Settings](#) page) do the following :

1. Attach an existing [VCS root](#) to your 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](#).
2. When VCS root is attached, specify the [checkout rules](#) for it — specifying checkout rules provides 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.
3. Define how project sources reach an agent via [VCS Checkout Mode](#).
4. Additionally, you can add a label into the version control system for the sources used for a particular build by means of [VCS Labeling](#) feature. **Since TeamCity 8.1**, VCS Labeling can be added as a 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 is a set of [VCS 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, TeamCity will issue a warning that the changes to the VCS root could potentially affect more than one project. The user is then prompted to either save the changes and apply them to all the projects using the VCS root, or to make a copy of the VCS root to be used by either a specific build configuration or project.

A VCS root can also be created at the level of a build configuration, using the **Version Control Settings** page.

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 ID of VCS root across all VCS roots in the system. VCS root ID can be used in parameter references to VCS root parameters and REST API . If not specified, will be generated automatically from VCS root parameters.
Repository URL	URL to VCS repository. Supports URLs in different formats, like: <code>http(s)://</code> , <code>svn://</code> , <code>ssh://git@</code> , <code>git://</code> and others as well as URLs in Maven format.
Checking interval	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.  Some public servers may block access if polled too frequently.
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 \(JetBrains\)](#)
- [Mercurial](#)
- [Perforce](#)
- [StarTeam](#)
- [Subversion](#)

- Team Foundation Server
- SourceGear Vault
- Visual SourceSafe



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

Since TeamCity 8.1, 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.

The currently supported VCS are Git, Mercurial, and Subversion. TFS and Perforce are partially supported. The URL formats for auto-discovery are listed below.

VCS URL Formats

VCS	URL Formats
Git	<ul style="list-style-type: none"> • http(s) urls • git://, ssh://git@ • Maven like urls: http://maven.apache.org/scm/git.html
Mercurial	<ul style="list-style-type: none"> • http(s) urls • Maven like urls: http://maven.apache.org/scm/mercurial.html
Subversion	<ul style="list-style-type: none"> • http(s) urls + URLs in Maven format: http://maven.apache.org/scm/subversion.html • svn://, svn+ssh://
TFS	<ul style="list-style-type: none"> • Maven like: scm:tfss:https://tfs10.codeplex.com:\$/maventest/ExampleProject • http url: http://tfs2012.labs.intellij.net:8080/tfs/team-unit\$/TeamCity/vs-addin-trunk • url with spaces: http://tfs2012.labs.intellij.net:8080/tfs/team-unit\$/TeamCity/vs-addin-trunk
Perforce	<ul style="list-style-type: none"> • Maven like: http://maven.apache.org/scm/perforce.html • same as Maven but without scm: prefix, for example: perforce://main:1666/myproject/
Vault	http(s) urls from Vault SourceCode Control web which contain "repid" parameter, e.g. <ul style="list-style-type: none"> • http://vault-server.example.net/VaultService/VaultWeb/Default.aspx?repid=1709&path=\$
CVS	no support yet
ClearCase	no support yet

ClearCase

Initial Setup

First of all you need to have an installed ClearCase client on the TeamCity server 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	A local path on the TeamCity server machine to the ClearCase view created during the initial setup . 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 (TW-21545). Do not use the view you are currently working with. TeamCity calls the update procedure <i>before</i> checking for changes and building patch, and thus it might lead to problems with checking in changes.
Relative path within the view	the path relative to the "ClearCase view path" that limits the sources to watch and checkout for the build.
Branches	Branches are used in the "-branch" parameter for the "lshistory" command to significantly improve its performance. 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. You can also choose the "custom" option and leave the text field blank - then the "-branch" parameter will not be used for the "lshistory" command at all. If you specify or TeamCity detects several branches, then "lshistory" will be called for every branch and all results will be merged.
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



A CVS client needs to be installed on the server machine and, if the agent-side checkout is used, on the agents.

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](#):

- [Common Options](#)
- [PServer Protocol Settings](#)
- [Ext Protocol Settings](#)

- SSH Protocol Settings (internal implementation)
- Local CVS Settings
- Advanced Options

Common Options

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 local connection only the path to the CVS repository should be used. TeamCity supports the following connection methods:</p> <ul style="list-style-type: none"> • pserver • ext • ssh • local
Checkout Options	<p>Click one of the radio buttons to define the way CVS will fill in and update the working directory. The following options are available:</p> <ul style="list-style-type: none"> • Checkout HEAD revision • Checkout from branch • Checkout by Tag
Quiet period	Since there are no atomic commits in CVS with help of this setting you can instruct TeamCity not to take (detect) change if specified period of time is not passed since the change date. This should 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 password.
Password File Path	Click this radio button to specify the path to the .cvspass file.
Connection Timeout	Specify the connection timeout.
Proxy Settings	Specify the proxy settings.
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.

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 proxy options above .

Local CVS Settings

Option	Description
Path to 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 the server mode. The default value is 'server'

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 (JetBrains)

Git support in TeamCity is implemented as a plugin.

- General Settings
 - Branch Matching Rules
 - Supported Protocols for Server Side Checkout
- Authentication Settings
- Server Settings
- Agent Settings
 - Git executable on the agent
- Internal Properties
- Limitations
- Known Issues
- Development Links



Git needs to be installed on the server machine and, if the agent-side checkout is used, on the agents.

Common VCS Root properties [are described here](#).

This page contains description of the Git-specific fields of the VCS root settings.



Important Notes

- Remote Run is supported in the IntelliJ IDEA and Eclipse plugins, pre-tested commit is not yet supported in any of the IDE plugins.
- 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.

General Settings

Option	Description
Fetch URL	The URL of the remote Git repository.
Push URL	The URL of the target remote Git repository.
Default branch	Set to <code>refs/heads/master</code> by default. This is the default branch used in the absence of a branch specification or when a 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, i.e. <code>refs/heads/</code> in Git (with the optional <code>*</code> placeholder). Note that only one asterisk is allowed and each rule has to start with a new line. See details in the section below .
Use tags as branches	Allows you to use git tags in branch specification. By default, tags are ignored.
Username style	Defines a way TeamCity binds a VCS change to the user. 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.
Username for tags/merge	A custom username used for labeling

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 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 Protocols for Server Side Checkout

The following protocols are supported for the server-side [checkout mode](#):

- ssh: (e.g. `ssh://git.somewhere.org/repos/test.git`, `ssh://git@git.somewhereElse.org/repos/test.git`, scp-like syntax: `git@git.somewhere.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. It is supported for the agent-side checkout only if git 1.7.3+ client is installed on the agent. See TW-18711 .
Private Key	<p>Valid only for SSH protocol. A private key must be in 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"> • Uploaded Key: uses the key(s) uploaded to the project. See SSH Keys Management for details. • Default Private key - Uses the mapping specified in <code><USER_HOME>/ .ssh/config</code> if the file exists or the private key file <code><USER_HOME>/ .ssh/id_rsa</code> (the files are required to be present on the server and also on the agent if the agent-side checkout is used). • Custom Private Key - Supported only for server-side checkout, see TW-18449. 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.

For all the available options to connect to GitHub, please see the [comment](#).

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>.BuildServer/system/caches/git</code> and <code>.BuildServer/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 <code>%env.TEAMCITY_GIT_PATH%</code> , the automatically detected git will be used, see Git executable on the agent 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.

Git executable on the agent

The path to the git executable can be configured in the agent properties by setting the value of the `TEAMCITY_GIT_PATH` environment variable.

If the path to git is not configured, the git-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 - try to run git.exe at:
 - C:\Program Files\Git\bin
 - C:\Program Files (x86)\Git\bin
 - C:\cygwin\bin
- for *nix - try to run git at:
 - /usr/local/bin
 - /usr/bin
 - /opt/local/bin
 - /opt/bin

If git wasn't found in any of these locations, try to run the git accessible from the \$PATH.

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.

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. Prior to 8.0.4 the default was 600.
teamcity.git.fetch.timeout	1800	The fetch process idle timeout. The fetch process reports what it is doing in the stdout and is killed if there is no output during this timeout. It is deprecated in favor of teamcity.git.idle.timeout.seconds since 8.1-EAP3, the git-plugin uses it only if its value is greater than the value of teamcity.git.idle.timeout.seconds. Its default value was 600 before 8.1-EAP3.
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
teamcity.git.monitoring.expiration.timeout.hours	24	When fetch in a separate process is used, it makes thread-dumps of itself and stores them under TEAMCITY_DATA_DIR/system/caches/git/<git-XXX>/monitoring (directory mapping can be found in TEAMCITY_DATA_DIR/system/caches/git/map). Thread dumps are useful for investigating problems with cloning from remote repository. This parameter specifies how long thread-dumps are to be stored.
teamcity.server.git.gc.enabled	false	Whether TeamCity should run git gc 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 git gc
teamcity.git.cleanupCron	0 0 2 * ? *	Cron expression 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 TW-14947
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	2	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.use.local.mirrors	false	When checkout on agent: whether TeamCity should clone to the local agent a mirror first and then clone to the working directory from this local mirror. This option speed-ups clean checkout, because only the build working directory is cleaned. Also if a single root is used in several build configurations, cloning will be faster.
teamcity.git.use.shallow.clone	false	When checkout on agent: run fetch with the option '--depth=1' if the agent uses local mirrors. This property can be set either in the agent properties or as a parameter in the build configuration.
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, because Git cannot clone a subdirectory of a repository. You can only map the whole repository to a specific directory using the following checkout rule `+ : .=>subdir`. The rest of the checkout rules are not supported.

Known Issues

- Tagging is not supported over the HTTP protocol
- `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 run fetch in the server process, set the `teamcity.git.fetch.separate.process` internal property to `false`.
- 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 [jgit-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

Since TeamCity 8.1, you can upload an SSH private key right into the project via the TeamCity web interface.

Uploading SSH Key to TeamCity Server

- Go to the **Administration** area | <Project> page | **Project Settings** on the left of the page.
- Click **SSH Keys**. On the page that opens, click **Upload SSH Key**.
- 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/<project>/pluginData/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.



Access to the TeamCity data directory should be kept secure, as the keys are stored in unmodified/unencrypted form on the file system.

Once the key is uploaded, a Git VCS root can be configured to use this uploaded key.

SSH Key Usage

The uploaded and referenced 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.

Common VCS Root properties [are described here](#). The section below contains the description of Mercurial-specific fields and options.

Mercurial support in TeamCity is implemented as a plugin.

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: + - :branch_name, where <code>branch_name</code> is specific to the VCS (with the optional * 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 TeamCity blogpost .
	<div style="background-color: #ffffcc; padding: 5px;"> Bookmarks support requires Mercurial 2.4 installed on the TeamCity server.</div>
Use tags as branches	Allows you to use tags in branch specification. By default, tags are ignored.
Clone repository to	Provide the path to a parent directory on the TeamCity server where a cloned repository will be created (applicable to "Automatically on server" checkout mode only). Leave blank to use the default path.
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. See more below.

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 first checks the value of the internal property `teamcity.hg.server.path` and if the property is set, its value is used. Secondly, the plugin tries to use the path from the settings of VCS root: if the path is equal to `%teamcity.hg.agent.path%`, it uses hg as a path, otherwise uses a value specified in the root settings.

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
<code>teamcity.hg.pull.timeout.seconds</code>	3600	Maximum time in seconds for pull operation to run
<code>teamcity.hg.server.path</code>	<code>hg</code>	Path to the hg executable on the server (see Path to hg executable detection for the details).

Agent configuration for Mercurial:

Property	Default	Description
<code>teamcity.hg.use.local.mirrors</code>	false	When checkout on agent: whether TeamCity should clone to local agent mirror first and then clone to working directory from this local mirror. This option speed-ups clean checkout, because only build working directory is cleaned. Also if single root is used in several build configurations clone will be faster.
<code>teamcity.hg.pull.timeout.seconds</code>	3600	Maximum time in seconds for pull operation to run
<code>teamcity.hg.agent.path</code>	<code>hg</code>	Path to hg executable on the agent (see Path to hg executable detection 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.



A Perforce client must be installed on the TeamCity server.

If you plan to use the agent-side [checkout mode](#), note that a Perforce client must be installed on the agents, and path to p4 executable must be added to the PATH environment variable.

- P4 Connection Settings
- Checkout Settings
- Other Settings

P4 Connection Settings

Option	Description
Port	Specify the Perforce server address. The format is host:port.
Client	<p>Click this radio button to directly specify the client workspace. The workspace should already be created by Perforce client applications like P4V or P4Win. Only mapping rules are used from the configured client workspace. The client name is ignored.</p> <div style="background-color: #fce4ec; padding: 10px;"> <p> Performance impact When this option is used with checkout on server mode, internal TeamCity source caching on the server side is disabled, which may worsen the performance of clean checkouts. Also, with this option, snapshot dependencies builds are not reused.</p> </div>
Client Mapping	<p>Click this radio button to specify the mapping of the depot to the client computer.</p> <p>If you have Client mapping selected, TeamCity handles file separators according to OS/platform of a build agent where a build is run. To enforce specific line separator for all build agents, use Client having LineEnd option specified in Perforce instead of Client mapping. Alternatively you can add an agent requirement to run builds only on specific platform.</p> <div style="background-color: #e6f2ff; padding: 10px;"> <p> Tip Use team-city-agent instead of the client name in the mapping.</p> </div> <p>Example:</p> <pre>//depot/MPS/... //team-city-agent/... //depot/MPS/lib/tools/... //team-city-agent/tools/...</pre>
User	Specify the user login name.
Password	Specify the password.
Ticket-based authentication	Check this option to enable ticket-based authentication.

Checkout Settings

Option	Description
Label to checkout	<p>Since TeamCity 7.1. If you need to check out sources not with the latest revision, but with 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 sources revision will be used for checkout. You can also specify changelist number here.</p> <div style="background-color: #fce4ec; padding: 10px;"> <p> It is recommended to use agent-side checkout if you use symbolic labels. With server-side checkout on label, TeamCity will perform full checkout.</p> </div>
Workspace options (checkout on agent)	If needed, you can set here the following options for <code>p4 client</code> command: Options, SubmitOptions, and LineEnd.

Other Settings

Path to P4 executable	Specify the path to the Perforce command-line client: <code>p4.exe</code> file). This path will be used both for server-side checkout and for agent-side checkout. If you need different values for this path on different build agents when using agent-side checkout, you can set the value using <code>TEAMCITY_P4_PATH</code> environment variable in <code>buildAgent.properties</code> file
Charset	Select the character set used on the client computer.

Support UTF-16 encoding	Enable this option if you have UTF-16 files in your project.
-------------------------	--

When 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 `teamcity.perforce.workspace.prefix` configuration parameter at the **Build Parameters** page with the prefix in the value.

See also:

Administrator's Guide: [VCS Checkout Mode](#)

StarTeam

This page describes the fields and options available when setting up VCS roots using StarTeam:

- StarTeam Connection Settings
 - Notes on Directory Naming Convention
- Changes Checking Interval
- VCS Root Project

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"> • As stored in repository — EOL symbols are preserved as they are stored in the repository. No conversion is done. • 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.
Directory naming	Define the mode for naming the local directories. Select one of the following: <ul style="list-style-type: none"> • 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. • 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.
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* names 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 the fields and options available when setting up VCS roots using Subversion:

- SVN Connection Settings
- SSH settings
- Checkout on agent settings
- Labeling settings
- Changes Checking Interval
- VCS Root Project
- Authentication for SVN externals
- Subversion 1.8 support
- Miscellaneous

You do not need Subversion client to be installed on TeamCity server or agents. TeamCity bundles Java implementation of SVN client ([SVNKit](#)).

SVN Connection Settings

Option	Description
URL	Specify the SVN URL that points to the project sources.
User Name	Specify the SVN user name.
Password	Specify the SVN password.
Default Config Directory	Check this option to make this the default configuration directory for the SVN connection.
Configuration Directory	If the Default Config Directory option is checked, you must specify the configuration directory.
Externals Support	<i>Check one of the following options to control the SVN externals processing</i>
Full support (load changes and checkout)	If selected, TeamCity will check out all configuration's sources (including sources from the externals) and will gather and display information about externals' changes in the Changes tab.
Checkout, but ignore changes	If selected, TeamCity will check out the sources from externals but any changes in externals' source files will not be gathered and displayed in the Changes tab. You might 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.
Ignore externals	If selected, TeamCity will ignore any configured "svn:externals" property, and thus TeamCity will not check for changes or check out any source file from the SVN externals.



Note that if you have anonymous access for some path within SVN, 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 SSH private key.
Private Key File Password	Enter the password to the SSH private key.
SSH Port	Specify the port that SSH is using.

Checkout on agent settings

Option	Description
Working copy format	<p>Select format of the working copy. Available values for this option are 1.4 (used as default in versions previous to TeamCity 5.0), 1.5 (current default), and 1.6.</p> <p>This option defines format version of Subversion files, which are located in <code>.svn</code> directories, when checkout on agent mode is used. Specified format is important in two cases:</p> <ul style="list-style-type: none"> 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. 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 checkout on agent mode.
Revert before update	<p>If the option is selected, then TeamCity always runs "<code>svn revert</code>" command before updating sources; that is, it will revert all changes in versioned files located in the working directory. When the option is disabled, and local modifications are detected during the update process, TeamCity runs "<code>svn revert</code>" after the update.</p> <p>TeamCity does not delete non-versioned files in working directory during the revert.</p>

Labeling settings

Option	Description
Labeling rules	Specify a newline-delimited set of rules that defines the structure of the repository. See the detailed format description for more details.

Authentication for SVN externals

TeamCity doesn't allow to specify SVN externals authentication parameters explicitly, in user interface. To authenticate on the SVN externals server, the following approaches are used:

- authenticate using same credentials (username/password) as for main repository
- authenticate without explicit username/password. In this case, credentials should be already available to 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 SVN external URL has the same prefix, as the main repository (there is a match > 20 characters), TeamCity tries main repository credentials first, and if failed, tries to connect without username/password (so they picked up from SVN configuration directory)
- if SVN external URL noticeably differs from the main repository, TeamCity tries to connect without username/password, and if failed, tries using credentials from the main repository

Subversion 1.8 support

Please see this page: [Subversion 1.8 support](#)

Miscellaneous

Directories are not considered changed when they have "svn:mergeinfo" Subversion property changes only. See [details](#).

See also:

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

TeamCity Subversion 1.8 support

This page provides information on Subversion 1.8 support in different TeamCity versions.

Feature	7.1.x	8.0.4	8.1
1.8 working copy (checkout on agent) TW-29404	✗	✗	✓
svn:// protocol support	supported with a patch	✓	✓

http(s):// protocol support	supported with a patch		
file:// protocol support TW-29404			
svn+ssh:// protocol support TW-29404			
Eclipse plugin TW-31245			
Visual Studio TeamCity addin TW-30892			

Team Foundation Server

This page contains descriptions of the fields and options available when setting up a VCS root for connection to Microsoft Team Foundation Server:



TFS integration is only supported for Windows machines. It is also required to have `Team Explorer` installed and correctly functioning on TeamCity server.

If you want to use agent-side checkout, `Team Explorer` should also be installed on all the agents which will run the related builds.

See the [section](#) for the supported versions.

Option	Description
Team Foundation Server Url	<p>Team Foundation Server URL in the following format:</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <pre>http[s]://<server>:<port></pre> </div> <p>Starting with TFS 2010, the URL can also include sub-path:</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <pre>http[s]://<server>:<port>/<path></pre> </div>
User Name	Specify user to access Team Foundation Server. This can be a user name or <code>DOMAIN\UserName</code> string. Use blank to let TFS select user account that is used to run TeamCity Server (or Agent for agent-side checkout)
Password	Enter the password of the user entered above.
Root	<p>Specify the root using the following format:</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> <pre>\$<project name><project catalogue></pre> </div>

Agent-Side Checkout

Agent-Side checkout is only supported on Windows agent machine (for Linux and Mac agent you may use server-side checkout). It is also required to have `Team Explorer` installed on agent machine.

TeamCity automatically creates TFS workspace for each [checkout directory](#) used. The workspace is created on behalf of user account that is specified in the VCS root settings.

Created TFS workspaces are automatically removed after a two-week idle time.

TFS does not allow to have several workspaces on a machine mapped to the same directory. TeamCity TFS agent-side checkout may attempt to remove intersecting workspaces in order to create new workspace that matches specified VCS root and checkout rules. Note, it is unable to remove workspaces created by another user.

Hosted TFS Login

You need to complete the following steps:

- Install Team Explorer 2012 Update 1 (or Visual Studio 2012 Update 1) to TeamCity server (and build agents)
- Login to TFS web site, click to edit user profile and specify "ALTERNATE AUTHENTICATION CREDENTIALS" in "CREDENTIALS" tab
- In TeamCity VCS settings use ##LIVE##\<YOUR_LOGIN> as login

For more details, please see [post](#) or [doc](#)

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

This page contains descriptions of the fields and options available when setting up VCS roots using Visual SourceSafe:

- [VSS Settings](#)
- [Changes Checking Interval](#)
- [VCS Root Project](#)



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

Option	Description
Path to srcsafe.ini	The full path of the VSS configuration file <code>srcsafe.ini</code> of the project repository. If the TeamCity server is run as a Windows service, make sure this path does not use mapped drives. If the file is placed on a network drive use syntax like <code>\vss-server\share\srcsafe.ini</code> where vss-server is a server name and share is a name of the shared directory.
Project	Specify the mandatory path to the project tree, starting with <code>\$/</code> .
User Name	Specify the mandatory name of the user on Visual SourceSafe server.
Password	Enter the password, that corresponds to the user name.

VCS Checkout Rules

VCS Checkout Rules allow you to checkout part of the VCS root configured and map directories from the version control to subdirectories in the build checkout directory on 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 do not 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.



Please note that exclude checkout rules (in the form of "-:") will generally only speed up server-side checkouts. 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, exclude checkout rules should generally be avoided for the agent-side checkout. Please refer to the [VCS Checkout Mode](#) page for more information. This does not apply to Perforce and TFS agent-side checkout, where exclude rules are processed in an effective manner.

To add a checkout rule, click the [edit checkout rules](#) link on the build configuration's **Version Control Settings** page and a pop-up window will appear where you can enter the rule.

The general syntax of a single checkout rule is as follows:

```
+ | -: VCSPath [=gt; AgentPath]
```

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
+ : . => Path	Checks out the root into Path directory
- : PathName	Excludes PathName (note: the path must be a directory and not a filename)
+ : VCSPath => .	Maps the VCSPath from the VCS to the Build Agent's default work directory
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 `scr/samples` path to the `samples` path in the Build Agent's default work directory. The second rule maps the contents of the `scr` path to the `production/sources` on the build agent, except `src/help` which was excluded by the first rule and `scr/samples` which was mapped to a different location by the third rule.

See also:

[Administrator's Guide: VCS Checkout Mode](#)

VCS Checkout Mode

The VCS Checkout mode is a configuration that sets how project sources reach an agent.

Please note that this mode affects only sources checkout. Current revision and changes data retrieving logic is executed by the TeamCity server and thus TeamCity server should have access to VCS server in any mode.

Agents do not need any additional software for automatic checkout modes.

TeamCity has three different VCS checkout modes:

Checkout mode	Description
---------------	-------------

Automatically on server	<p>This is the default setting. The TeamCity server will export the sources and pass them to an agent before each build. Since sources are exported rather than checked out, no administrative data is stored in the file system and version control operations (like check in or label or update) can not be performed from the agent. TeamCity optimizes communications with the VCS servers by caching the sources and retrieving from the VCS server only the necessary changes (read more on that). Unless Clean checkout is performed, 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="background-color: #ffffcc; padding: 10px;"> <p> Note about usage</p> <ul style="list-style-type: none"> • 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, server side checkout is usually more effective. • Note that in some cases this checkout mode can lower load produced on VCS repositories, especially if Clean Checkout is performed often, due to the caching of clean patches by the server. • Note that in server checkout mode the administration directories (like <code>.svn</code>, <code>CVS</code>) are not created on the agent. </div>
Automatically on agent	<p>The build agent will check out the sources before the build. This checkout mode is supported only for CVS, Subversion, TFS, Mercurial, Perforce, ClearCase and Git. Agent-side checkout frees more server resources and provides the ability to access version control-specific directories (<code>.svn</code>, <code>CVS</code>, <code>.git</code>); 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> <div style="background-color: #ffffcc; padding: 10px;"> <p> Note</p> <ul style="list-style-type: none"> • Agent checkout is usually more effective with regard to data transfers and VCS server communications. As long as agent side checkout is just a source update or checkout, it creates necessary administration directories (like <code>.svn</code>, <code>CVS</code>), and thus allows you to communicate with the repository from the build: commit changes and so on. • Machine-specific settings (like configuring SSL communications, etc.) have to be configured on each machine using agent-side checkout. • "Exclude" VCS Checkout Rules 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" VCS Checkout Rules are not supported for Git when using checkout on an agent due to Git limitations. • Certain version controls can provide additional options when agent-side checkout is used. For example, Subversion. </div>
Do not check out files automatically	<p>TeamCity will not check out any sources. The build script has to contain the commands to check out the sources. Please note that TeamCity will accurately report changes only if the checkout is performed on the revision specified by the <code>build.vcs.number.*</code> properties passed into the build.</p>

See also:

Administrator's Guide: [VCS Checkout Rules](#)

Configuring Build Steps

When creating a build configuration, it is important to configure the sequence of build steps to be executed. 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 [General Settings](#)); 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.

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.
- **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. (This option is available [since TeamCity 8.0](#))
- **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 (the second attempt to stop the build will work).



- 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.

For the details on configuring individual build steps, refer to:

- [.NET Process Runner](#)
- [Ant](#)
- [Command Line](#)
- [Duplicates Finder \(.NET\)](#)
- [Duplicates Finder \(Java\)](#)
- [FxCop](#)
- [Gradle](#)
- [Inspections](#)
- [Inspections \(.NET\)](#)
- [IntelliJ IDEA Project](#)
- [Maven](#)
- [MSBuild](#)
- [MSpec](#)
- [MSTest](#)
- [NAnt](#)
- [NuGet](#)
- [NUnit](#)
- [PowerShell](#)
- [Rake](#)
- [Visual Studio \(sln\)](#)
- [Visual Studio 2003](#)
- [Ipr \(deprecated\)](#)
- [Xcode Project](#)

See also:

Concepts: Build Runner

.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 build working directory if it differs from the build checkout directory .
.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.  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.

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 comes bundled with Ant 1.8.2, and **since TeamCity 8.1-** with Ant 1.8.4.

In this section:

- Support for Running Parallel Tests
- Ant Runner Settings
 - Ant Parameters
 - Java Parameters
 - Test parameters
 - Code Coverage

Support for Running Parallel Tests

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.

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 (Currently the VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)
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 build working directory if it differs from the checkout directory.
Targets	Use this text field to specify valid Ant targets as a list of space-separated strings. 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 distributive bundled with TeamCity (Ant 1.8.2, since TeamCity 8.1 , Ant 1.8.4).  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.

Java Parameters

Option	Description
JDK home path	Use this field to specify the path to the JDK which should be used to run the build (launch Ant). If the field is left blank, the value of JAVA_HOME environment variable is used. (The variable can come from the build configuration settings, or agent environment, or properties). If JAVA_HOME is not found, TeamCity uses the Java home of the build agent process itself.  Starting with TeamCity 5.1 in many cases agents are able to detect the installed Java automatically. The agent will set appropriate environment variables for each detected Java version: JDK_14, JDK_15, JDK_16 and so on. You can specify the reference to such environment variable in the JDK home path, like: %env.JDK_15%.
JVM command line parameters	You can specify such JVM command line parameters as, for example, <i>maximum heap size</i> or parameters enabling <i>remote debugging</i> . These values are passed by the JVM used to run your build. Example: <div style="border: 1px dashed #ccc; padding: 5px; width: fit-content;">-Xmx512m -Xms256m</div>

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 filesset containing included tests only and processes it before all other filesets. It also patches other filesets to exclude tests added to the automatically generated filesset. 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.
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 <i>blinking</i> tests).
Run new and modified tests first	If checked, before any other test, TeamCity will run the tests added or modified in the change lists included in the running build.



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.

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

With this build runner you can run any script supported by OS.
In this section:

- [Command Line Runner Settings](#)
- [General Settings](#)

Command Line Runner Settings

General Settings

Option	Description
Working directory	Specify the Build Working Directory if it differs from the build checkout directory.
Run	Select whether you want to run an executable with parameters or custom shell/batch scripts.
Command executable	Specify the executable file of the build runner. <i>The option is available if "Executable with parameters" is selected in the Run dropdown.</i>
Command parameters	Specify parameters as a space-separated list. <i>The option is available if "Executable with parameters" is selected in the Run dropdown.</i>
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. <i>The option is available if "Custom script" is selected in the Run dropdown.</i> <div style="background-color: #e0f2e0; padding: 10px;"> ✓ When TeamCity meets a string surrounded by %-sign in the script, it treats such string as a reference to a parameter. To avoid this, use double %, i.e.: %%notParameter%%. </div>

See also:

Concepts: [Build Runner](#) | [Build Checkout Directory](#) | [Build Working Directory](#)
Administrator's Guide: [Configuring Build Steps](#)

Duplicates Finder (.NET)

The **Duplicates Finder (.NET)** Build Runner is intended for catching similar code fragments and providing a report on discovered repetitive blocks of C# and Visual Basic .NET code in Visual Studio 2003, 2005, 2008, 2010, and 2012 solutions. Since TeamCity 8.0.6, Visual Studio 2013 is also supported.



This runner requires .NET Framework 4.0 (or higher) to be installed on the agent where builds will run.

This page contains reference information about the following **Duplicates Finder (.Net)** Build Runner fields:

- Sources
- Duplicate Searcher Settings

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: src\MySolution.sln
Exclude	Enter newline-delimited Ant-like wildcards to exclude files from the duplicates search (for example, */generated{*}{}*.cs). The entries should be relative to the checkout root.

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 <i>namespace specifications</i> will be recognized as duplicates. <div style="border: 1px dashed #ccc; padding: 5px;"><pre>NLog.Logger.GetInstance().Log("abcd"); A.Log.Logger.GetInstance().Log("abcd");</pre></div>
Discard literals	If this option is checked, similar lines of code with different literals will be recognized as duplicates. <div style="border: 1px dashed #ccc; padding: 5px;"><pre>myStatusBar.SetText("Not Logged In"); myStatusBar.SetText("Logging In...");</pre></div>
Discard local variables	If this option is checked, similar code fragments with different local variable names will be recognized as duplicates. <div style="border: 1px dashed #ccc; padding: 5px;"><pre>int a = 5; a += 6; int b = 5; b += 6;</pre></div>
Discard class fields name	If this option is checked, the similar code fragments with different field names will be recognized as duplicates. <div style="border: 1px dashed #ccc; padding: 5px;"><pre>Console.WriteLine(myFoo); Console.WriteLine(myBar); ... where myFoo and myBar are declared in the containing class</pre></div>

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): <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>
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.

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, thus an IntelliJ IDEA project file (.ipr) or directory (.idea) is required to configure the runner. 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 duplicates finder engine, TeamCity requires either IntelliJ IDEA project file\directory, or Maven pom.xml to be specified here.

Path to the project	Depending on the type of project selected in the Project file type , specify here: <ul style="list-style-type: none"> 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). For Maven project: path to the pom.xml file. This information is required by this build runner to understand the structure of the project. <div style="background-color: #e0f2f1; padding: 10px; margin-top: 10px;"> The specified path should be relative to the checkout directory. </div>
Detect global libraries and module-based JDK in the *.iml files	<i>This option is available if you use an IntelliJ IDEA project.</i> 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. <div style="background-color: #ffcccc; padding: 10px; margin-top: 10px;"> 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	<i>This option is available if you use an IntelliJ IDEA project.</i> Click this button to reparse your IntelliJ IDEA project and import the build settings right from the project, for example the list of JDKs. <div style="background-color: #ffffcc; padding: 10px; margin-top: 10px;"> 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	Enter a path to a Build Working Directory if it differs from the Build Checkout Directory . 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 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 Set value to field , specify a path to the 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	Use this field to specify the JDK home for the project. <div style="background-color: #ffffcc; padding: 10px; margin-top: 10px;"> 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 <code>JAVA_HOME</code> 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 ../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 home path	Use this field to specify the path to your custom JDK which should be 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 computer, or from <code>env.JAVA_HOME</code> property specified in the build agent configuration file (<code>buildAgent.properties</code>). If neither of these values is specified, TeamCity uses the Java home of the build agent process itself.
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: <pre>-Xmx512m -Xms256m</pre>

Duplicate Finder Settings

Option	Description
Test sources	<p>If this option is checked, the test sources will be included in the duplicates analysis.</p> <p>i 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.</p>

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	<p>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.</p> <p>Initial version:</p> <pre> 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); } } </pre> <p>After analysing code for duplicates without distinguishing methods, the duplicated fragments can be extracted:</p> <pre> 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); } </pre>
Distinguish types	<p>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.</p> <pre> new MyIDE().updateStatus() new TheirIDE().updateStatus() </pre>
Distinguish literals	<p>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.</p> <pre> myWatchedLabel.setToolTipText("Not Logged In"); 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 %system.FxCopRoot% 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	<i>The option is available when autodetect installation is selected.</i> 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	<i>The option is available when Specify installation root is selected.</i> Enter the path to the FxCop installation root on the agent machine or the path to an FxCop executable relative to the Build Checkout Directory .
	 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 Build Checkout Directory and separated by spaces. Enter exclude wildcards to refine the included assemblies list.
FxCop project file	Enter the path relative to the Build Checkout Directory 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 Build Checkout Directory 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 Gradle wrapper, you should have properly configured Gradle Wrapper scripts checked in to your Version Control.

In this section:

- [Gradle Parameters](#)
- [Run Parameters](#)
- [Java Parameters](#)
- [Build properties](#)
- [Code Coverage](#)

Gradle Parameters

Option	Description
Gradle tasks	Specify Gradle task names separated by space. For example: <code>:myproject:clean :myproject:build</code> or <code>clean build</code> . If this field is left blank, the 'default' is used. Note, that TeamCity currently supports building Java projects with Gradle. Groovy/Scala/etc. projects building has not been tested.
Incremental building	TeamCity can make use of 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 bin 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. The default is <code>JAVA_HOME</code> environment variable or the agent's own Java.
JDK home path	<i>The option is available when <Custom> is selected above.</i> 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 <code>env.JAVA_HOME</code> property specified in the build agent configuration file (<code>buildAgent.properties</code>). If these both values are not specified, TeamCity uses Java home of the build agent process itself.
JVM command line parameters	You can specify such JVM command line parameters as, e.g., <i>maximum heap size</i> or parameters enabling <i>remote debugging</i> . These values are passed by the JVM used to run your build. Example: -Xmx512m -Xms256m

Build properties

TeamCity build properties are available in the build script via the `teamcity` property of the project. This property contains the map with all defined system properties (see [Defining and Using Build Parameters](#) for details). The example below contains a task that will print all available build properties to the build log (it must be executed by the buildserver):

```
task printProperties << {
    teamcity.each { key, val ->
        println "##tc-property name='${key}' value='${val}'"
    }
}
```

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. 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

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, or Maven pom.xml to be specified here.
Path to the project	Depending on the type of project selected in the Project file type , specify here: <ul style="list-style-type: none">• 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).• For Maven project: the path to the pom.xml file. This information is required by this build runner to understand the structure of the project. <div style="background-color: #e0f2f1; padding: 10px; margin-top: 10px;"><p> The specified path should be relative to the checkout directory.</p></div>
Detect global libraries and module-based JDK in the *.iml files	<i>This option is available if you use an IntelliJ IDEA project to run the inspections.</i> 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. <div style="background-color: #ffcccc; padding: 10px; margin-top: 10px;"><p> 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.</p></div>
Check/Reparse Project	<i>This option is available if you use an IntelliJ IDEA project to run the inspections.</i> Click this button to reparse your IntelliJ IDEA project and import the build settings right from the project, for example the list of JDKs. <div style="background-color: #ffffcc; padding: 10px; margin-top: 10px;"><p> 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.</p></div>
Working directory	Enter a path to a Build Working Directory if it differs from the Build Checkout Directory . 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="background-color: #ffffcc; padding: 10px;">  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 <code>JAVA_HOME</code> 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. The default value is used for Linux and Windows operating systems:</p> <div style="border: 1px dashed #ccc; padding: 5px; margin-top: 10px;"> <pre>jre/lib/*.jar</pre> </div> <p>For Mac OS X, use the following lines:</p> <div style="border: 1px dashed #ccc; padding: 5px; margin-top: 10px;"> <pre>lib/*.jar ../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 home path	Use this field to specify the path to your custom JDK which should be 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 the agent computer, or from the <code>env.JAVA_HOME</code> property specified in the build agent configuration file (<code>buildAgent.properties</code>). If neither of these values is specified, TeamCity uses the Java home of the build agent process itself.
JVM command line parameters	<p>Specify the desired Java Virtual Machine parameters, for example the maximum heap size. These settings are passed to the JVM used to run your build.</p> <p>Example:</p> <div style="border: 1px dashed #ccc; padding: 5px; 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 [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
- 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.

Inspections (.NET)

The **Inspections (.NET)** runner allows you to use the benefits of [JetBrains ReSharper code quality analysis](#) feature right in TeamCity.

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

Refer to [ReSharper documentation](#) for more details.

This page contains reference information about the **Inspections (.Net)** Build Runner fields:

- Sources to Analyze

- Environment Requirements
- InspectCode Options
- Build Failure Conditions
- Build before analyze



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 Visual Studio Multi-Targeting feature. Agent requirement will be created for every checked item. .NET Frameworks client profiles are not supported as target frameworks

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 ReSharper Web Help and JetBrains .NET Tools Blog
Enable debug output	Check this option to include debug messages in the build log and publish the file with additional logs (dotnet-tools-inspectcode.log) as a hidden artifact.

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 [lpr \(deprecated\)](#) 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">• Test runner parameters are not supported• running of the Ant or Maven before tests start is not supported• alternative JRE is not supported
TestNG	, with limitations	<ul style="list-style-type: none">• Test runner parameters are not supported• running of the Ant or Maven before tests start is not supported• running of the tests from group is not supported• alternative JRE is not supported
Application run configuration	, with limitations	<ul style="list-style-type: none">• running of the Ant or Maven before tests start is not supported• altrenative JRE is not supported
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> <p> Specified path should be relative to the checkout directory.</p>

Detect global libraries and module-based JDK in the *.iml files	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> 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.</p>
Check/Reparse Project	Click to reparse the project and import build settings right from the IDEA project, for example the list of JDKs.  If you update your project settings in IntelliJ IDEA - add new jdk, libraries, don't forget to update build runner settings by clicking Check/Reparse Project .
Working directory	Enter a path to a Build Working Directory , if it differs from the Build Checkout Directory . 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 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	Use this field to specify JDK home for the project.  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.
JDK Jar File Patterns	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: <div style="border: 1px dashed #ccc; padding: 5px; width: fit-content;"> <pre>jre/lib/*.jar</pre> </div> For Mac OS X, use the following lines: <div style="border: 1px dashed #ccc; padding: 5px; width: fit-content;"> <pre>lib/*.jar ...Classes/*.jar</pre> </div>

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 home path	Use this field to specify the path to your custom JDK which should be 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 computer, or from <code>env.JAVA_HOME</code> property specified in the build agent configuration file (<code>buildAgent.properties</code>). If these both values are not specified, TeamCity uses Java home of the build agent process itself.
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

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 build that are configured in 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 <i>shared</i> (via "Share" checkbox in IntelliJ IDEA Run/Debug Configurations dialog) and checked in to the version control.

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:

Maven

Note that you can create a new Maven-based build configuration automatically from a specified `pom.xml`, 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.

Below you can find reference information about the **Maven2** Build Runner fields:

- [Maven Parameters](#)
- [Maven Home](#)
- [User Settings](#)
- [Java Parameters](#)
- [Local Artifact Repository Settings](#)
- [Incremental Building](#)
- [Code Coverage](#)

Maven Parameters

Option	Description
Goals	<p>In the Goals field, specify the sequence of space-separated Maven goals that you want TeamCity to execute. Some Maven goals can use version control systems, and, thus, they may become incompatible with some VCS checkout modes. If you want TeamCity to execute such a goal:</p> <ul style="list-style-type: none"> • Select "Automatically on agent" in the VCS checkout mode drop-down list on the Version Control Settings page. This makes the version control system available to the goal execution software.
Path to POM file	<p>Specify the path to the POM file relative to the build working directory. 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><subdirectory>/pom.xml</code> is used.</p> <div style="background-color: #e0f2e0; padding: 10px; margin-top: 10px;"> ✔ Alternatively, click to choose the file using the VCS repository browser (Currently the VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.) </div>
Additional Maven command line parameters	<p>Specify the list of command line parameters.</p> <div style="background-color: #ffffcc; padding: 10px; 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 Build Working Directory , if it differs from the build checkout directory .

Maven Home

In **Maven selection** field, choose the Maven version you want to use. By default, the path to Maven installation is taken from the M2_HOME environment variable, otherwise the bundled Maven 3 is used.

Alternatively, you can set it as `%MAVEN_HOME%` environment variable right on a build agent.

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 Maven Server-Side Settings .
<Custom>	Enter the path to an alternative user settings file. The path should be valid on agent and also on the server, see Maven Server-Side Settings .
Predefined settings	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 <i>Manage settings files</i> . You can upload Maven user settings files at any time using the Administration Integrations Maven Settings page. The uploaded files are stored under <code><TeamCity Data Directory>/config/_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.  Starting from version 8.1, 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 the Maven Settings tab. The files will be available in the project and its subprojects.

If Custom or Predefined settings are used, the path to the effective user settings file is available inside the maven process as the `teamcity.maven.userSettings.path` system property.

Java Parameters

JDK Home Path	The path to JDK Home is read from the <code>*JAVA_HOME*</code> environment variable or <code>*JDK home*</code> specified on the build agent if you leave this field empty. If these two values are not specified, TeamCity uses the JDK home on which the build agent process is started.
JVM command line parameters	Specify JVM command line parameters; for example, <i>maximum heap size</i> or parameters enabling <i>remote debugging</i> . These values are passed by the JVM used to run your build. For example: -Xmx512m -Xms256m

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.

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.

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. **Since TeamCity 8.0.5** Microsoft Build Tools 2013 are also supported.

Before setting up the 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	<p>Specify the path to the solution to be built relative to the Build Checkout Directory. For example:</p> <pre>vs-addin\addin\addin.sln</pre> <p> Alternatively, click to choose the file using the VCS repository browser (Currently the VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)</p>
Working directory	Specify the path to the build working directory .
MSBuild version	Select the MSBuild version: .NET Framework or Mono xbuild. Since TeamCity 8.1 , Microsoft Build Tools 2013 is also supported.

MSBuild ToolsVersion	Specify here the version of tools that will be used to compile (equivalent to the <code>/toolsversion:</code> commandline argument).
	<p> 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 http://msdn.microsoft.com/en-us/library/bb383796(VS.100).aspx</p>
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.
Command line parameters	Specify any additional parameters for <code>msbuild.exe</code>
Reduce test failure feedback time	Use the following 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.

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. <p> 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.</p>
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>mstest.exe</code> .

Code Coverage

Learn about [configuring code coverage options](#).

See also:

Administrator's Guide: [Configuring .NET Code Coverage](#)

MSTest

This page describes MSTest runner options, for details on MSTest support, please refer to the corresponding page.

Option	Description
Path to MSTest.exe	A path to <code>MSTest.exe</code> file. By default Build Agent will autodetect MSTest installation. You may use <code>%system.MSTest.8.0%</code> , <code>%system.MSTest.9.0%</code> or <code>%system.MSTest.10.0%</code> to refer to build agent auto-detected <code>MSTest.exe</code> .
List assembly files	A list of assemblies to run MSTests on. Will be mapped to <code>/testcontainer:file</code> argument.
MSTest run configuration file	Specify a MSTest run configuration file to use (<code>/runconfig:file</code>).
MSTest metadata	Enter a value for <code>/testmetadata:file</code> argument.
Testlist from metadata to run	Every line will be translated into <code>/testlist:line</code> argument.
Test	Names of the tests to run. This option will be translated to the series of <code>/test: arguments</code> . Check Add /unique commandline argument to add <code>/unique</code> argument to <code>MSTest.exe</code>
Results file	Enter a value for <code>/resultsfile:file</code> commandline argument.
Additional commandline parameters	Enter an additional commandline parameters for <code>MSTest.exe</code>

Please, note that tests run with MSTest are not reported on-the-fly.

For more details on configuring MSTests, please refer to the [MSTest.exe Command-Line Options](#)

Code Coverage

To learn about configuring code coverage options, please refer to the [Configuring .NET Code Coverage](#) page.

See also:

Administrator's Guide: [Configuring Unit Testing and Code Coverage | MSTest Support](#)

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 overridden 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	<p>Specify path relative to the Build Checkout Directory.</p>  <p>Alternatively, click  to choose the file using the VCS repository browser (Currently the VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)</p>
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.
Working directory	Specify the path to the Build Working Directory . By default, the build working directory is set to the same path as the Build Checkout Directory .
NAnt home	<p>Enter a path to the <code>NAnt.exe</code>.</p>  <p>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.</p>
Target framework	<p>Sets <code>-targetframework:</code> option to '<code>NAnt</code>' and generates appropriate agent requirements (<code>mono-2.0</code> target framework will require <code>Mono</code> system property, <code>net-2.0</code> — <code>DotNetFramework2.0</code> property, and so on). Selecting unsupported in TeamCity framework (<code>sscli-1.0</code>, <code>netcf-1.0</code>, <code>netcf-2.0</code>) won't impose any agent requirements.</p> <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 <code>exe</code> file if <code>.NET</code> framework was found, and through <code>mono</code> executable, if not.</p>
Command line parameters	<p>Specify any additional parameters for <code>NAnt.exe</code></p>  <p>TeamCity passes automatically to <code>NAnt</code> all defined system properties, so you do not need to specify all of the properties here via '<code>-D</code>' option.</p>
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 <i>blinking</i> tests)

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

TeamCity integrates with [NuGet](#) package manager and allows you to:

- Install and update NuGet packages: [NuGet Installer](#) build runner;
- Pack NuGet packages: [NuGet Pack](#) build runner;
- Publish packages to a feed of your choice: [NuGet Publish](#) build runner.



NuGet build runners are only supported on build agents running Windows OS.

You can use TeamCity as NuGet server as [described below](#).

Installing NuGet to TeamCity agents

To start working with NuGet integration, build agents require the NuGet.exe Command Line tool installed.

To do it in TeamCity:

1. Go to **Administration | NuGet Settings | NuGet.exe** tab.
2. Click **Fetch NuGet**.
3. In **Add NuGet**, select which NuGet versions you want to be installed on agents.
4. Select the default version.

You can also upload your own NuGet package containing NuGet.exe instead of downloading it from the public feed using **Upload NuGet**.



Note also that installing NuGet on agents results in upgrade of agents.

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 trigger a new build when a NuGet package is updated, use [NuGet Dependency Trigger](#).
- To create a package and publish it to the internal TeamCity NuGet Server, enable TeamCity as a NuGet Server (see the section below), use the [NuGet Pack](#) build step and properly configure [artifact paths](#).

Using TeamCity as NuGet Server

If for some reason you don't want to publish packages to public feed, e.g. you're producing packages that are intended to be used internally; you can use TeamCity as a native NuGet Server instead of setting up your own repository.



TeamCity running on any of the supported operating systems (Windows, Linux, Mac OS X) can be used as a NuGet Server.

To start using TeamCity as a NuGet Server, click **Enable** on the **Administration | NuGet Settings | NuGet Server** page. Two different links will be displayed on the same page: for public (with `guestAuth` prefix) and private (with `httpAuth` prefix) feed. If **Public Url** is not available, you need to enable the [Guest user login](#) in TeamCity on the **Administration | Global Settings** page.

For an example of set up see blog post: [Setting up TeamCity as a native NuGet Server](#).

When you have TeamCity NuGet server enabled:

- You don't need to use [NuGet Publish](#) build step (unless you want to publish packages on some public feed).
- You can work with built NuGet packages as with build artifacts: remember to specify [artifact paths](#) in the General Settings of your build configuration.
- You can add TeamCity NuGet server to your repositories in Visual Studio to avoid having to type in long URLs each time you want to read from a specific package repository (add NuGet repository and specify the public URL provided by TeamCity when enabling NuGet server).
- 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](#).



Note that the links given allow you to get the packages belonging to projects where you have permissions to access artifacts. This relates to both accessing the feed from Visual Studio and using console.

See also:

Administrator's Guide: [NuGet Installer](#) | [NuGet Publish](#) | [NuGet Pack](#) | [NuGet Dependency Trigger](#)

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.



NuGet Installer is only supported on build agents running Windows OS.

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 (if you installed NuGet beforehand), or specify a custom path to <code>NuGet.exe</code> .
Packages Sources	Specify the NuGet package sources, for example your own NuGet repository. If you are using TeamCity as a NuGet repository, specify <code>%teamcity.nuget.server%</code> here. If left blank, <code>nuget.org</code> is used to search for your packages.
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">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.Disable looking up packages from local machine cache: Equivalent to the <code>-NoCache</code> option of the <code>NuGet.exe install</code> command.
Update Packages	Update packages with help of NuGet update command: Uses the <code>NuGet.exe update</code> command to update all packages under solution. 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">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.Update via packages.config - Select to update packages via calls to <code>NuGet.exe update Packages.Config</code> for each <code>packages.config</code> file under the solution.
Update Options	<ul style="list-style-type: none">Include pre-release packages: Equivalent to the <code>-Prerelease</code> option of the <code>NuGet.exe update</code> commandPerform 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.

See [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](#)

NuGet Pack

NuGet Pack build runner allows to build a NuGet package from a given spec file. If you want to publish this package, add one more build step called **NuGet Publish**.



NuGet Pack is only supported on build agents running Windows OS.

To configure NuGet Pack:

1. Select NuGet version to use from the **NuGet.exe** drop-down list (if you have [installed NuGet beforehand](#)), or specify custom path to `NuGet.exe`.
 2. Specify your package parameters:
 - Enter path(s) to `csproj` or `nuspec` file(s). You can specify as many specification files here as you need. Wildcards are supported. If you specify here a `csproj` file, you won't have to redefine version number and copyright information in the spec file.
 - Specify version for the package. You can use TeamCity variable `%build.number%` here.
 - Specify base directory, where the files defined in the `nuspec` file are located (the directory against which the paths in `<files></files>` from `nuspec` are resolved, usually some `bin` directory). If left blank, TeamCity will use build checkout directory as base directory.
 3. In the "Output Directory" field, specify the path where generated NuGet package should be put. Optionally, you can clean this directory before packing. If you're using TeamCity as NuGet repository, select the **Publish created packages to build artifacts** check box to publish packages to TeamCity's NuGet server and be able to use them as regular TeamCity artifacts.
1. Set additional parameters, if needed:
 - Exclude files: specify one or more wildcard patterns to exclude when creating a package. Equivalent to `NuGet.exe -Exclude` argument.
 - Properties: Semicolon or new line separated list of package creation properties. For example to make a release build you define here `Configuration=Release`.
 - Options: Select whether the output files of the project should be in the **tool** folder. Select whether a package containing sources and symbols should be created. When specified with a `nuspec`, creates a regular NuGet package file and the corresponding symbols package (needed for publishing the sources to `Symbolsource`)
 - Set "Additional commandline arguments" to be passed to `NuGet.exe`.

See also:

Administrator's Guide: [NuGet Installer](#) | [NuGet Publish](#)

NuGet Publish

NuGet Publish build runner is intended to publish (`push`) your NuGet packages to a given feed (custom or default).



If you're using TeamCity as NuGet server, you don't need to add this build step. However the output of the **NuGet Pack** build step should be a build artifact: specify path to it in [General Settings](#) of your build configuration.



NuGet Publish is only supported on build agents running Windows OS.

To configure NuGet Publish:

1. Select 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. In the **Package Sources** field specify the destination to publish the package, by default it is `nuget.org`. If you have your own NuGet server, specify its address here.
3. Provide your API key. This field is mandatory.
4. List the packages you want to upload: specify each package individually or use wildcards.
5. If you want to upload your package but keep it invisible in the feed, select the "Only upload package but do not publish it to feed" check box. This works for NuGet versions 1.5 and older.

See also:

Administrator's Guide: [NuGet Installer](#) | [NuGet Pack](#)

NUnit

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.



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.**

NUnit Test Settings

NUnit runner	Select the NUnit version to be used to run the tests.
.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.
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.</p> <p>In the following example, TeamCity will search for the tests assemblies in all project directories and run these tests.</p> <div style="border: 1px dashed #ccc; padding: 5px; margin-top: 10px;">***.dll</div> <div style="background-color: #ffffcc; padding: 10px; margin-top: 10px;"> All these wildcards are specified relative to path that contains the solution file.</div>
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.</p> <p>In the following example, TeamCity will omit tests specified in this directory.</p> <div style="border: 1px dashed #ccc; padding: 5px; margin-top: 10px;">**\obj***.dll</div> <div style="background-color: #ffffcc; padding: 10px; margin-top: 10px;"> All these wildcards are specified relative to path that contains the solution file.</div>
NUnit categories include	Specify NUnit categories of tests that should be run. Multiple entries are comma-separated.
NUnit categories exclude	Specify NUnit categories that should be excluded from the tests to be run. Multiple entries are comma-separated.
Run process per assembly	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.

See also:

Administrator's Guide: [Configuring Unit Testing and Code Coverage | NUnit Support](#)

PowerShell

PowerShell build runner is designed specifically to run PowerShell scripts.

PowerShell Settings

Option	Description
Version	List of PowerShell versions, supported by TeamCity. Since 8.1 is passed to <code>powershell.exe</code> as <code>-Version</code> command line argument
Powershell run mode	Select the desired execution mode on a x64 machine.
Working directory	Specify the path to the build working directory.
Script	Select whether you want to enter the script right in TeamCity, or specify path to a script: <ul style="list-style-type: none"> Script file: Enter path to Powershell file. It has to be relative to checkout directory. Script source: Enter PowerShell script source. Note, that TeamCity references (build parameters and TeamCity system properties) will be replaced in the code.
Script execution mode	Specify PowerShell script execution mode. If you've selected 'Execute .ps1 script with "-File" argument' your script should be signed or you should make PowerShell allow execution of arbitrary .ps1 files. If execution policy doesn't allow to run your scripts, select 'Put script into PowerShell stdin' mode to avoid this issue.
Script arguments	Available if "Script execution mode" option is set to "Execute .ps1 script with "-File" argument". Specify here arguments to be passed into PowerShell script.
Additional command line parameters	Specify parameters to be passed to <code>powershell.exe</code> .

Interaction with TeamCity

Attention must be paid, when using PowerShell to interact with TeamCity through service messages. Powershell tends to wrap strings written to console with commands like `Write-Output`, `Write-Error` and similar (see [TW-15080](#)). To avoid this behavior, either `Write-Host` command should be used, or buffer length should be adjusted 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 {}
    }
}
```

Development Links

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.

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 setup 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 Build Checkout Directory . <div style="background-color: #e0f2e0; padding: 10px; margin-top: 10px;"> Alternatively, click to choose the file using the VCS repository browser (Currently the VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.) </div>
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 Build Checkout Directory .
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="background-color: #e0f2e0; padding: 10px; margin-top: 10px;"> <pre>ruby rake <Additional Rake command line parameters> <TeamCity Rake Runner options, e.g TESTOPTS> <tasks></pre> </div>

Ruby Interpreter

Option	Description
Use default Ruby	Use Ruby interpreter settings defined in the Ruby environment configurator build feature settings or the interpreter will be searched in the PATH.
Ruby interpreter path	The path to Ruby interpreter. The path cannot be empty. This field supports values of environment and system variables. For example: %env.I_AM_DEFINED_IN_BUILDAgent_CONFIGURATION%
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 Bundler requirements 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 Command Line step before the Rake runner step. Also, remember to setup the Ruby environment configurator build feature to automatically pass Ruby interpreter to the command line runner.
Debug	Check the Track invoke/execute stages option to enable showing <i>Invoke</i> stage data in the build log.

Tests Reporting

Option	Description
Attached reporters	If you want TeamCity to display the test results on a dedicated Tests tab of the Build Results page, select here the testing framework you use: Test::Unit, Test-Spec, Shoulda, RSpec or Cucumber.  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](#).

Visual Studio (sln)

This page contains reference information for the Visual Studio(sln) Build Runner that builds Microsoft Visual Studio 2005, 2008, 2010 and **since TeamCity 8.0.5**, 2013 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

General Build Runner Options

Option	Description
Solution file path	<p>Specify the path to the solution to be built relative to the Build Checkout Directory. For example:</p> <div style="border: 1px dashed #ccc; padding: 5px; margin-top: 10px;">vs-addin\addin\addin.sln</div> <p>Alternatively, click  to choose the file using the VCS repository browser (Currently the VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)</p>
Working directory	Specify the Build Working Directory . (optional)
Visual Studio	Select the Visual Studio version: 2005, 2008, 2010 .
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 Build Parameters page.

Visual Studio 2003

Visual Studio 2003 Build Runner supports building Microsoft Visual Studio 2003 .NET projects.



- 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	<p>A path to the solution to be built is relative to the build checkout directory. For example:</p> <div style="border: 1px dashed #ccc; padding: 5px; margin-top: 10px;"> <pre>vs-addin\addin\addin.sln</pre> </div> <div style="background-color: #e0f2e0; padding: 10px; margin-top: 10px;"> <p> Alternatively, click to choose the file using the VCS repository browser (Currently the VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)</p> </div>
Working directory	Specify the Build Working Directory .
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	<p>Click this link and specify the desired map in the text area. Use the following format:</p> <div style="border: 1px dashed #ccc; padding: 5px; margin-top: 10px;"> <pre>http://localhost:8111=myProjectPath/myProject</pre> </div> <p>where</p> <ul style="list-style-type: none"> • http://localhost:8111 is a host where the project will be uploaded • myProjectPath/myProject is the project root
Resolve URLs via WebDAV	<p>Click this radio button, if you want the URLs to be resolved via WebDav.</p> <div style="background-color: #ffcccc; padding: 10px; margin-top: 10px;"> <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="background-color: #e0f2e0; padding: 10px; margin-top: 10px;"> <p> TeamCity passes automatically to NAnt all defined system properties, so you do not need to specify all of the properties here via '<code>-D</code>' 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.</p> <p>In the following example, TeamCity will search for the tests assemblies in all project directories and run these tests.</p> <div style="border: 1px dashed #ccc; padding: 5px; margin-top: 10px;"> <pre>***.dll</pre> </div> <div style="background-color: #e0f2e0; padding: 10px; 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.</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 path that contains the solution file.</p>
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 <i>blinking</i> tests)

See also:

[Administrator's Guide: NUnit for MSBuild](#)

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](#)
- [Java Parameters](#)
- [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="background-color: #e0f2f1; padding: 10px; margin-top: 10px;"> i 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="background-color: #f8d7da; padding: 10px; margin-top: 10px;"> - 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="background-color: #FFFACD; padding: 10px; margin-top: 10px;"> ! If you update your project settings in IntelliJ IDEA - add new jdk, libraries, don't forget to update build runner settings by clicking Check/Reparse Project. </div>
Working directory	<p>Enter a path to a Build Working Directory, if it differs from the Build Checkout Directory. 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>	<p>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.</p>

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> <div style="background-color: #FFFACD; padding: 10px; margin-top: 10px;"> ! 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. </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> <hr/> <p>For Mac OS X, use the following lines:</p> <hr/>
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 home path	Use this field to specify the path to your custom JDK which should be 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 computer, or from <code>env.JAVA_HOME</code> property specified in the build agent configuration file (<code>buildAgent.properties</code>). If these both values are not specified, TeamCity uses Java home of the build agent process itself.
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: <hr/>

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 batchtest 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 data-bbox="311 291 1454 354">[-]moduleName:[testFileNamePattern]</pre> <p>where:</p> <ul style="list-style-type: none"> • [-]: If a pattern starts with minus character, the corresponding files will be excluded from the build process. • moduleName : this name can contain wildcards. • [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 Ant syntax 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: <pre data-bbox="393 572 1454 656">*test:**/*Test.java -*:**/ui/**/*.java</pre>
Search for tests	<p>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:</p> <ul style="list-style-type: none"> • Throughout all project sources: look for tests in both "sources" and "test" folders of your IDEA project. • In test sources only: look through the folders marked as tests root only.
Classpath in Tests	<p>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.</p>
Override classpath in tests	<p>If this option is checked, you can define test classpath from a single, explicitly specified module.</p>
Module name to use JDK and classpath from	<p>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.</p>
JUnit Fork mode	<p>Select the desired fork mode from the combo box:</p> <ul style="list-style-type: none"> • Do not fork: fork is disabled. • Fork per test: fork is enabled, and each test class runs in a separate JVM • Fork once: fork is enabled, and all test classes run in a single JVM
New classloader instance for each test	<p>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.</p>
Include Ant runtime	<p>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).</p>
JVM executable	<p>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).</p>
Stop build on error	<p>Check this option, if you want the build to stop if an error occurs during test run.</p>
JVM command line parameters for JUnit	<p>Specify JVM parameters to be passed to JUnit task.</p>

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> <p> 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.</p>
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 <i>blinking</i> 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.  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.
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.

Xcode Project

The *Xcode Project Build Runner* supports Xcode 3 (target-based build) and Xcode 4 (scheme-based build). **Since TeamCity 8.1** Xcode 5 is supported. The runner provides structured build log based on Xcode build stages, detects compilation errors, reports tests from the xcdebuild 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.

Xcode Project Runner Settings

Setting	Build	Description
Path to the project or workspace		The path to a (.xcodeproj) project file or a (.xcworkspace) workspace file, should be relative to the checkout directory. For Xcode 3 build, only the path to a project is supported.
Working directory		Specify the build working directory .
Path to Xcode		Specify the path to Xcode on the agent.
Build		Select either a target-based or scheme-based (for Xcode 4+ only) 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 you can 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 Apple documentation .
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)		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. It is not recommended to change this field unless you want to change the number or order of actions. i 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).
Run tests		Check this option if want to run tests after your project is built.
Additional command line parameters		Other command line parameters to be passed to the "xcodebuild" utility.

See also:

Concepts: Build Runner

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.

Prior to TeamCity 8.1, build features are configured on the **Build Steps** page of the build configuration settings.

Since TeamCity 8.1, build features are configured on the dedicated page of the build configuration settings.



You can disable a build feature temporarily or permanently at any time, even if it is inherited from a build configuration template.

The currently available build features are:

- [AssemblyInfo Patcher](#)
- [Automatic Merge](#)
- [Build Files Cleaner \(Swabra\)](#)
- [Free disk space](#)

- Performance Monitor
- Ruby Environment Configurator
- Shared Resources
- VCS Labeling
- XML Report Processing

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.

When configured, TeamCity searches for all `AssemblyInfo` files (`.cs`, `.vb` or `.cpp`) in their standard locations under checkout directory and replaces the parameter for the `AssemblyVersion` and `AssemblyFileVersion` attributes with the build number you have specified in the TeamCity web UI.

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.

At the end of the build the files are reverted to the initial state.

Automatic Merge

The **Automatic Merge** build feature tracks builds in branches matched by a given filter and merges them into a specified destination branch if the build satisfies a certain merge condition. It is supported for Git and Mercurial VCS roots for build configurations with enabled **feature branches**.

Automatic Merge Settings

Check [Adding Build Features](#) for notes on how to add a build feature.

Option	Description
Watch builds in branches	A filter 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 logical name of the destination branch the sources will be merged to. A corresponding VCS branch must be present in a repository. Parameter references are supported here.
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 the related [TeamCity blog post](#).

Build Files Cleaner (Swabra)

Bundled Swabra plugin allows 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 checkout directory for newly created, modified and deleted files and enable file locking processes detection.

The checkout directory state is saved into a file in the caches directory named `<checkout_directory_name_hash>.snapshot` using the DiskDir format. The checkout directory to snapshot name map is saved into `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 should 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 clean checkout. 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	Specify newline-separated set of + -:path rules to define what files and folders should be involved in files collection process (by default entire checkout directory is monitored). The path can be relative (based from build's checkout directory) or absolute and can include Ant-like wildcards. If no +: or -: prefix is specified, a rule is treated as "include". Specifying a directory affects its entire content and sub-directories. Rules on any path should come in order from more abstract to more concrete, e.g. use -:*/dir/* to exclude all <code>dir</code> folders and their content, or -:some/dir, +:some/dir/inner to exclude <code>some/dir</code> folder and all its content except <code>inner</code> subfolder and its content. <div style="background-color: #ffffcc; padding: 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 for locking processes detection <code>handle.exe</code> is required on agents.
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.

Downloading Handle

You can download `handle.exe` from the [Administration | Tools](#) page.

Select **Sysinternals handle.exe** from the list of tools, specify the **URL for downloading handle.exe** or download it manually and specify the path on local machine, click **Continue** and TeamCity will automatically download or upload `handle.exe` and send it to Windows agents.

`handle.exe` is present on agents only after the upgrade process.

You may also download `handle.exe`, extract it on the agent and set up the `handle.exe.path` system property manually.

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](#).

Free disk space

The [Free disk space](#) build feature allows ensuring the build gets enough disk free space.

Before the build and before each build preparation stage, the agent will check the currently available free disk space. If the amount is less than specified, it will try to clean data of other builds before proceeding.

The data cleaned includes:

- the checkout directories that were marked for [deletion](#);
- the cache of previously downloaded artifacts (that were downloaded to the agent via TeamCity artifact dependencies)
- contents of other build's checkout directories in the reversed most recently used order.

The disk space check is performed for two locations: the agent's temp directory and the build checkout directory.

By default, each build has the required free space set to 3Gb. You can specify a custom free disk space value.

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](#).

Other ways to set the free disk space value

Besides, for compatibility reasons the free disk space value can be specified via properties. However, it is advised to use the [build feature](#) 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`

See also:

Administrator's Guide: TeamCity Server Disk Space Watcher

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. When enabled, each build has an additional tab called **PerfMon** on the build results page, where this statistics is presented as a graph. You can also click on points in the chart to see the corresponding part of the build log.

For example, from the picture below it is clear that at some point the CPU and Disk usage is very low. This lasts for about 20 minutes. It seems that the tests executing at this time need some investigation, probably, most of the time they are blocked on some lock or wait for some event:



The Performance monitor supports Windows, Linux, Solaris and MacOS X operating systems. 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.

Ruby Environment Configurator

Ruby environment configurator build feature passes Ruby interpreter to all build steps. The build feature adds selected Ruby interpreter and gems bin directories to system PATH environment variable and configures other necessary environment variables in case of **RVM** interpreter. E.g. in **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 **Rake** build runner you need to add **Command Line** build step which launches custom script like:

```
gem install rake --no-ri --no-rdoc
gem install bundler --no-ri --no-rdoc
```

Ruby Environment Configurator Settings

Option	Description
Ruby interpreter path	<p>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 dashed #ccc; padding: 5px; margin-top: 10px;"> <code>%env.I_AM_DEFINED_IN_BUILDAgent_CONFIGURATION%</code> </div>
RVM interpreter	<p>Specify here the RVM interpreter name and optionally a gemset configured on a build agent. Note, that interpreter name cannot be empty. If gemset isn't specified the default one will be used.</p> <div style="background-color: #e0f2f1; padding: 10px; margin-top: 10px;"> <p>i This option can be used if you don't want use <code>.rvmrc</code> settings, for instance to run tests on different ruby interpreters instead of hardcoded in <code>.rvmrc</code> file.</p> </div>
RVM with <code>.rvmrc</code> file	<p>Since TeamCity 7.1 Specify here the path to a <code>.rvmrc</code> file which should be relative to the checkout directory. If the file is specified, TeamCity will fetch environment variables using rvm-shell and will pass it to all build steps.</p>
Fail build if Ruby interpreter wasn't found	<p>Check the option to fail build if Ruby environment configurator cannot pass Ruby interpreter to step execution environment because the interpreter wasn't found on agent.</p>

See also:

Administrator's Guide: [Configuring Build Steps](#) | [Command Line](#) | [Rake](#)

Shared Resources

The **Shared Resources** build feature allows limiting the number of 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](#).

Adding and Editing Shared Resources

You can add, edit shared resources, and explore their details on the **Shared Resources** tab of the project configuration page. Shared resources defined at a project level will be available in all its subprojects and build configurations.

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

After shared resources have been created, you need to define which build configuration(s) will use them by adding this build feature to the build configuration settings.

To define locks for the available resources, click **Add 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.

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 (`system.locks.readLock.<lockName>` or `system.locks.writeLock.<lockName>`), e.g. the parameter name can be: `teamcity.locks.readLock.databaseUrl`.

Development Links

See the Shared Resources plugin page at [TeamCity Shared Resources](#).

See also:

[JetBrains TV TeamCity Shared Resources Screencast](#)

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 build results Changes tab.

On this page:

- Automatic VCS labeling
 - Manual VCS labeling

- Subversion Labeling Rules
- Labeling Rule Examples

Automatic VCS labeling

You can set TC 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. Therefore, a labeling failure is not a standard [notification event](#). However, 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 build results Changes tab.

Labeling is configured for a build configuration/template.

Prior to TeamCity 8.1, VCS labeling is configured on the [VCS Settings page](#) of the build configuration settings.

Since TeamCity 8.1, 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](#).

Since TeamCity 8.1, 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:

In **TeamCity 8.0**, navigate to the [Changes tab](#) of the build results page and click the **Label this build sources** link.

In **TeamCity 8.1** and later, 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).

XML Report Processing

The *XML Report processing* build feature allows to use the 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 plugin
- NUnit-Console XML reports
- MSTest TRX reports (for MSTest 2005/2008/2010)
- Google Test XML reports

and the following code inspection tools:

- FindBugs (code inspections only)
- 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. 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.

- 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](#)

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](#) and specify there all the required parameters.



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.**

Please, refer to [NUnit build runner page](#).

Alternative approaches

However, if for some reason it is not applicable, 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.
TeamCity NUnit Addin supports the following versions: **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.**
- 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	✓	✗	✓	✓
<code><nunit2></code> NAnt task	✓	✓/✗*	✓	✓
<code><NUnit></code> MSBuild task	✓	✓/✗*	✓	✓
<code><NUnitTeamCity></code> 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 | 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 TeamCity build agent environment, the results are reported to the TeamCity agent.



- If you need to access the path to TeamCity NUnit launcher from some process, you can add an environment variable `%system.teamcity.dotnet.nunitlauncher%`.
- Values surrounded with "%" within custom scripts in Commandline runner TeamCity treats as TeamCity references.

You can pass to the TeamCity NUnit Test Launcher the following command line options:

```
 ${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> . 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 .
/category-include:<list>	The list of categories separated by ';' (optional).
/category-exclude:<list>	The list of categories separated by ';' (optional).
/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.

Examples

The following examples assume that the `teamcity.dotnet.nunitlauncher` property is set as system property on the [Build 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  
Assembly2.dll
```

NUnit for NAnt Build Runner

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 test 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 `nunit2` 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> 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.
<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

Working with NUnit Task in MSBuild Build

This section assumes, that you already have a MSBuild build script with 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.

TeamCity provides custom [NUnit](#) task compatible with [NUnit](#) task from [MSBuild Community tasks](#) project. If you've configured [NUnit](#) tests in your MSBuild build script via [NUnit](#) task, TeamCity will automatically replace the original task with its own, and start command line TeamCity [NUnit](#) test launcher in order to be able to report test results. TeamCity's [NUnit](#) task version is compatible with the [MSBuild Community Task](#) and will issue a warning, if TeamCity does not support an attribute listed in the build script. These warnings are only for your information and will not affect the building process.



In order for this task to work, the `teamcity_dotnet_nunitlauncher` system property has to be accessible. 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.

The [NUnit](#) task uses the following syntax:

```
<UsingTask TaskName="NUnit" AssemblyFile="$(teamcity_dotnet_nunitlauncher_msbuild_task)" />
```

```
<NUnit Assemblies="@{assemblies_to_test}" />
```

Example (part of the MSBuild build script):

```

<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask TaskName="NUnit" AssemblyFile="$(teamcity_dotnet_nunitlauncher_msbuild_task)" />

  <Target Name="SayHello">
    <NUnit Assemblies="!!!*put here item group of assemblies to run tests on*!!!" />
  </Target>
</Project>

```



- 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 TeamCity test runner to support third-party NUnit addins, please, refer to the [NUnit Addins Support](#) section for the details.

Using NUnitTeamCity task in MSBuild Build Script

TeamCity provides custom `NUnitTeamCity` task compatible with `NUnit` task from [MSBuild Community tasks](#) project. If you'll provide `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 `NUnitTeamCity` task, perform the following steps:

- 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.
- 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 MSBuild Community tasks project.
ExcludeCategory	As used in the <code>NUnit</code> task from MSBuild Community tasks project.
NUnitVersion	Version of NUnit to be used to run the tests. <div style="background-color: #e0f2f1; padding: 10px; margin-top: 10px;"> Info 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. </div> For example, <code>NUnit-2.2.10</code> .
Addins	List of third-party NUnit addins to be used. For more information on using NUnit addins, refer to NUnit Addins Support 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.

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 Solution Runner for Microsoft Visual Studio 2005 and 2008 solution files. 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 TeamCity independent build script, consider the following trick:

```
<NUnitTeamCity ... Condition=" '$(TEAMCITY_VERSION)' != '' "/>
```

MSBuild Property `TEAMCITY_VERSION` is added to msbuild when started from TeamCity.

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 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="

" />
```

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

To support NUnit addins for the MSBuild runner, specify the `Addins` property for the `NUnitTeamCity` task with the following format:

```
Addins="

"
```

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

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 Test Reporting plugin, or manually configure reporting tests by means of service messages.



TeamCity NUnit Addin supports the following versions: **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.**

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 `addons` 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 addons to application probe paths. Add the following code into the `config/runtime` element:

```
<assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <probing privatePath="addons"/>
</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

TeamCity provides support for MSTest 2005/2008/2010 testing framework via parsing of the MSTest results file (.trx file).

Due to specifics of MSTest tool, TeamCity does **not** support on-the-fly test reporting for MSTest. All test results are imported **after** tests run has finished.

There are two ways to report test results to TeamCity:

- Add [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

Build agent will add `%system.MSTest.8.0%`, `%system.MSTest.9.0%` or `%system.MSTest.10.0%` system property (see [Build Agent Configuration](#) for detail) in cases MSTest.exe from Microsoft Visual Studio 2005/2008/2010 is found on the build agent running system.

See also:

Concepts: Testing Frameworks
Administrator's Guide: [NUnit Support](#)

Configuring .NET Code Coverage

NUnit Test Launcher bundles support for .NET code coverage using NCover, PartCover and dotCover coverage engines. Details on configuring code coverage can be found on the corresponding pages:

- [NCover](#)
- [JetBrains dotCover](#)
- [PartCover](#)
- [Manually Configuring Reporting Coverage](#)

Coverage support limitations

Coverage configuration via TeamCity UI is only supported for the cases when the tests are run using TeamCity-managed test launchers.

Specifically these covers:

NAnt runner: <nunit2> NAnt task

MSBuild runner: <NUnit> and <NUnitTeamCity> MSBuild tasks

Any runner: [TeamCity NUnit Test Launcher](#)

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](#).

See also:

Administrator's Guide: [NUnit Support](#)

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](#), [NUnitTeamCity 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"> Do not enter the arguments that can be configured in the web UI. Do not specify the output path for the generated reports. It is configured automatically by TeamCity.
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 NCoverExplorer documentation .
Sorting	Select the preferred sorting option. For the details, refer to NCoverExplorer documentation .
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:

- While creating/editing Build Configuration, go to the Build Steps page.
- Add one of the build steps that support NCover ([.NET Process Runner](#), [MSBuild](#), [MSpec](#), [MSTest](#), [NAnt](#), [NUnit](#)), configure unit tests.
- Select **NCover (3.x)** in **.NET coverage tool**.
- 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](#).

JetBrains dotCover

TeamCity comes bundled with the console runner of [JetBrains dotCover](#). Just by enabling the configuration option, you can 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 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.

dotCover Settings

Path to dotCover Home	Leave this field blank to use the bundled dotCover. Alternatively, specify the path to the dotCover installed on a build agent.
Filters	Specify assemblies to profile - one per line - using the following syntax: +:assembly=*;type=*;method=***. Use -:assembly to exclude an assembly from code coverage. The asterisk wildcard (*) is supported here.
Attribute Filters	If you don't 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> or <code>-:module=myassembly;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.

Note that dotCover coverage engine reports statement coverage instead of line coverage.

Bundled dotCover Versions

This section provides information on the versions of dotCover bundled with different TeamCity versions.

TeamCity Version	dotCover Version
TeamCity 8.1	dotCover 2.6
TeamCity 8.0.6 (and later 8.0.x)	dotCover 2.6
TeamCity 8.0.5	dotCover 2.5
TeamCity 8.0	dotCover 2.2
TeamCity 7.1.3 (and later 7.1.x)	dotCover 2.2
TeamCity 7.1.2	dotCover 2.1
TeamCity 7.1	dotCover 2.0
TeamCity 7.0.1 (and later 7.0.x)	dotCover 1.2
TeamCity 7.0	dotCover 1.1
TeamCity 6.0	dotCover 1.0

See also:

[Administrator's Guide: 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 system property , 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 <code>[*]*</code> to include all assemblies.
Exclude Assemblies	Explicitly specify the assemblies to be excluded from coverage statistics. If you have specified <code>[*]*</code> to profile all assemblies, type <code>[JetBrains]*</code> here to exclude TeamCity NUnit test runner sources.
Report XSLT	Write new-line delimited xslt transformation rules in the following format: <code>file.xslt=>generatedFileName.html</code> . You can use the default PartCover xslt as <code>file.xslt</code> , 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	Full path to DotCover home folder to override bundled dotCover.

For NCover 3.x:

key	description	sample value
ncover3_home	Full path to NCover installation folder.	Path to NCover3 installation directory
ncover3_reporter_args	Arguments for 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 Maven plugin; 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 and [EMMA](#) open-source toolkit. **Since TeamCity 8.1**, [JaCoCo](#) coverage is also supported.

In this section:

- 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: org.apache.*.
Classes to exclude from instrumentation	Use newline-separated patterns for fully qualified class names to be excluded from the coverage. A pattern should start with a valid package name and can contain a wildcard, for example: org.apache.*. 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.e(c|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> <div style="background-color: #f0e6d2; padding: 10px;">  Warning 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 http://emma.sourceforge.net/reference_single/reference.html#prop-ref.tables for details). </div>
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 http://emma.sourceforge.net/reference_single/reference.html#prop-ref.tables.</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:

Concepts: [Build Runner](#) | [Code Coverage](#)

Administrator's Guide: [Configuring Java Code Coverage](#) | [IntelliJ IDEA](#)

JaCoCo

Since TeamCity 8.1, 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](#).



Make sure your tests run in the `fork=true` mode. Otherwise the coverage data may not be properly collected.

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:

- In the **Code Coverage** section, select **JaCoCo** as a coverage tool in the **Choose coverage runner** drop-down.
- 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 <code>+ -:[path]</code> to scan for classfiles to be analyzed. Libraries and test classfiles are not needed to be listed unless their coverage is wanted.	<code>+:src/main/java/**</code>

Classes to instrument	Newline-delimited set of classname patterns in the form + -[path]. Allows filtering out unwanted classes listed in "Classfile directories or jars" field. Useful in case test classes are compiled.	-:*Test*
------------------------------	---	----------

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.

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

TestNG TestNG 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.

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:

Concepts: [Build Runner](#)
Extending TeamCity: [Risk Tests Reordering in Custom Test Runner](#)

Build Failure Conditions

In TeamCity you can adjust the conditions when a build should be marked as failed.

Common build failure conditions

To do so, in the **Fail build if** area specify how TeamCity should fail builds:

- **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.
- **it runs longer than ... minutes:** Check this option and enter a value in minutes to enable time control on the build. If the specified amount of time is exceeded, the build is automatically canceled. This option helps to deal with builds that hang and maintains agent efficiency.
- **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.

Advanced build failure conditions

Starting from TeamCity 7.0 you can instruct TeamCity to mark a build as failed if it has become "worse" by some of the metrics generated by the build, for example, code coverage, or artifacts size, etc. For instance, you can mark build as failed if code coverage or code duplicates number is worse than in the previous build. Another advanced build failure condition allows to mark build as fail when a certain line is met in build log. To add such build failure condition to your build configuration click the [Add build failure condition](#) link and select it 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 will fail the build upon exceeding them. For example, you can instruct TeamCity to mark build as failed, if code coverage or code duplicates number is worse than in the previous build.

In general there are two ways to configure this build fail condition:

- A *build metric* exceeds or is less than the specified threshold. For example: **Fail build if build duration (secs) is more than 3000.**
- A *build metric* has become "worse" comparing to specific build by specified value. For example: **Fail build if build duration (secs) is more than 300 compared to Last successful build.** In this case a build will fail if it runs 300 seconds longer than the last successful build.

To compare a *build metric* to, you can select 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), build duration (secs), number of classes, number of code duplicates, 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 tests, percent of block-level coverage, percent of class-level coverage, percent of line-level coverage, percent of method-level coverage, test duration (secs).

Moreover you can add your own build metric. To do so, you need to modify TeamCity's configuration file `<TeamCity Data Directory>/config/main-config.xml` and add the following section there:

```
<build-metrics>
<statisticValue key="myMetric" description="build metric for number of files"/>
</build-metrics>
```

So, if your build will publish value `myMetric`, you can use it as a criteria 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 build failure. Lines are matched fair without time and block name prefixes which precede each message in build log representation.

To configure this build failure condition you need to specify:

- a string which presence/absence in build log is an indicator of build failure,
- a failure message to be displayed on build results page when build fails due to this condition.

Note that as well as exact text you can specify a [Java Regular Expression](#) regexp pattern to be looked for in build log.

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.

Build trigger is a rule which initiates a new build on certain events. The build is put into the [build queue](#) and is run when there are idle agents that can run it.

While creating/editing a build configuration, you can configure triggers on 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.

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 previous build failed after specified time delay.
- [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 packages update detected in 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

- [Trigger a build on changes in snapshot dependencies](#)
- [Per-check-in Triggering](#)
- [Quiet Period Settings](#)
- [VCS Trigger Rules](#)
 - [Trigger Rules Example](#)
- [Branch Filter](#)

VCS triggers automatically start a new build each time TeamCity detects new changes in the configured VCS roots.

TeamCity periodically (according to the checking for changes interval of a VCS root) polls VCS roots of the build configuration for changes. Newly detected changes appear as Pending Changes of a build configuration.

A new VCS trigger with default settings triggers a build each time new changes are detected.

However, you can adjust a VCS trigger to your needs by means of [Quiet Period Settings](#) and [Build Trigger Rules](#).

Trigger a build on changes in snapshot dependencies

By default, if you have a [build chain](#) (i.e. a number of build configurations interconnected by [snapshot dependencies](#)), automatic triggering of builds is performed in one direction only: if a build with snapshot dependencies is triggered, all the builds it depends on are triggered too, but **not vice versa**. ([More about snapshot dependencies](#)).

But it is possible to configure a dependent build to start by enabling the **Trigger a build on changes in snapshot dependencies** option.

For example, the *Test* build configuration snapshot-depends on the *Compile* build configuration. When a *Test* build is triggered, a *Compile* build is triggered too. But if *Compile* is triggered, nothing happens to *Test*. If you want a *Test* build to be triggered on changes in *Compile*, enable **Trigger a build on changes in snapshot dependencies** in the VCS Trigger options of the *Test* build configuration. In this setup, no VCS trigger is required for the *Compile* Build configuration. See also an example at the [Build Dependencies](#) page.

Per-check-in Triggering

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. Moreover, 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 in sorting out whose change has broken a build, or caused new test failure.

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.

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 actual quiet period will not be less than maximum checking for changes interval among build configuration VCS roots. Because TeamCity must ensure that changes were collected at least once during the quiet period.

Quiet period can be set to the default value (which can be changed globally at the [Administration | Global Settings](#) page), or custom value can be specified.



Note, that when a build is triggered by a trigger with 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](#).

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 VCS root settings and specifying [Checkout Rules](#).

To limit the changes that trigger the build, use 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 "-").

The general syntax for a single rule is:

```
+|-:[user=VCS_username;][root=VCS_root_id;][comment=VCS_comment_regexp]:Ant_like_wildcard
```

Where:

- **Ant_like_wildcard** - A [wildcard](#) to match the changed file path. Only "" and "" patterns are supported, "?" pattern is *not supported. The file paths in the rule can be relative (resulting paths on the agent will be matched) or absolute (started with '/), VCS paths relative to VCS root are matched).
- **VCS_username** - if specified, limits the rule only to the changes made by a user with 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 VCS comment. Use [Java Regular Expression](#) pattern for matching text in a comment (see examples below). The rule matches if the comment text contains matched text

portion; to match entire text, include ^ and \$ special characters.

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 rules.



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.

Trigger Rules Example

```
+:  
-:*.html  
-:user=techwriter;root=Internal SVN:/misc/doc/*.xml  
-:lib/**  
-:comment=minor:**  
-:comment=^oops$:**
```

Here,

- "-:*.html" excludes all .html files from triggering a build.
- "-:user=techwriter;root=Internal SVN:/misc/doc/*.xml" excludes builds being triggered by .xml files checked in by user "techwriter" 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](#) principles ^ and \$ in pattern stand for string beginning and ending)

Branch Filter

Branch filter setting limits a set of logical branch names according to specified rules. Branch filter has the following format:

```
+:logical branch name  
-:logical branch name
```

Where **logical branch name** is a part of branch name matched by branch specification (i.e. displayed for a build in TeamCity UI), see [Working With Feature Branches](#). Wildcard character (*) can also be used.

Examples:

Only default branch is accepted:

```
+:<default>
```

All branches except default are accepted:

```
+:*  
-:<default>
```

Only branches with with feature- prefix are accepted:

```
+:feature-
```

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 for branch remote-run and you push changes to both 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.

On this page:

- [Schedule Trigger options](#)
- [Examples](#)

- Brief description of the cron format used
- VCS Settings
 - VCS Trigger Rules
 - Trigger Rules Example
 - Branch Filter

Schedule Trigger options

Besides triggering builds **daily** or **weekly**, 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.

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 Settings

You can restrict schedule trigger to start builds only if there are pending changes in your version control by selecting the corresponding option.

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 VCS root settings and specifying [Checkout Rules](#).

To limit the changes that trigger the build, use 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 "-").

The general syntax for a single rule is:

Where:

- **Ant_like_wildcard** - A [wildcard](#) to match the changed file path. Only "**"** and "**"** patterns are supported, "?" pattern is *not supported. The file paths in the rule can be relative (resulting paths on the agent will be matched) or absolute (started with '/', VCS paths relative to VCS root are matched).
- **VCS_username** - if specified, limits the rule only to the changes made by a user with 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 VCS comment. Use [Java Regular Expression](#) pattern for matching text in a comment (see examples below). The rule matches if the comment text contains matched text portion; to match entire text, include ^ and \$ special characters.

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 rules.



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.

Trigger Rules Example

Here,

- "-:*.html" excludes all .html files from triggering a build.
- "-:user=techwriter;root=Internal SVN:/misc/doc/*.xml" excludes builds being triggered by .xml files checked in by user "techwriter" 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](#) principles ^ and \$ in pattern stand for string beginning and ending)

Branch Filter

Branch filter setting limits a set of logical branch names according to specified rules. Branch filter has the following format:

Where **logical branch name** is a part of branch name matched by branch specification (i.e. displayed for a build in TeamCity UI), see [Working With Feature Branches](#). Wildcard character ("*") can also be used.

Examples:

Only default branch is accepted:

All branches except default are accepted:

Only branches with with feature- prefix are accepted:

Configuring Maven Triggers

The **Build Triggering** page allows you to add the following Maven dependency triggers:

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

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 identifier of a group the desired Maven artifact belongs to.
Artifact Id	Specify the artifact's identifier.
Version or Version range	Specify version or version range of the artifact. The version range syntax is described in the table 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 classifier of an artifact.
Maven repository URL	Specify URL to the Maven repository. Note, that this parameter is optional. If the URL is not specified, then: <ul style="list-style-type: none">For a Maven project the repository URL is determined from the POM and the server-side Maven SettingsFor a non-Maven project the repository URL is determined from the server-side Maven Settings only
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.

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

NuGet Dependency Trigger allows to start a new build if there is a NuGet packages update detected in NuGet repository.

Prerequisites:

- .NET 4.0 should be installed on TeamCity server

To configure NuGet Dependency Trigger you need to:

- select NuGet version to use from the **NuGet.exe** drop-down list (if you have installed NuGet beforehand), or specify custom path to `NuGet.exe`;
- specify NuGet package source, if it's different from `nuget.org`;
- and enter package Id to check for updates.

Optionally, you can specify package version range to check for. If not specified, TeamCity will check for latest version.

Starting with TeamCity 7.1 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

Finish build trigger triggers a build of current build configuration if a build of selected build configuration is finished. If **Trigger after successful build only** checkbox is enabled, a build is triggered only after successful build of the selected configuration.



In most of the cases finish build trigger should be used with snapshot dependencies, i.e. current build configuration where trigger is defined should have direct or indirect snapshot dependency on build configuration selected in the trigger. If there is no snapshot dependency the following limitations will occur:

- it is likely that build of build configuration being triggered will not have the same revisions as finished build even if both configurations have the same VCS settings
- if build configuration with finish build trigger has artifact dependency on last finished build of build configuration specified in trigger settings, there is no guarantee that artifacts of a build which caused build triggering will be used, because while triggered build sits in the build queue, another build can finish
- build triggered by finish build trigger will always be triggered in default branch even if finished build has some other branch

All these limitations do not apply if build configuration with "Finish build trigger" has snapshot dependency to selected build configuration. In this case the trigger will run build on the same revisions and will attach build to the chain. It will also use consistent artifacts if they are produced by dependencies.

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

- Configuring Artifact Dependencies Using Web UI
- Configuring Artifact Dependencies Using Ant Build Script

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.
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">When selecting the build configuration, take your clean-up policy settings 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.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.
Build number	<i>This field appears if you have selected build with specific build number in the Get artifacts from list.</i> Specify here the exact build number of the artifact.
Build tag	<i>This field appears if you have selected last finished build with specified tag in the Get artifacts from list.</i> 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	Allows setting a branch 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, 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, 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=>dlls` will extract *.dll from all archives matching the `release-*.*.zip` pattern to the `dlls` directory.
- `a.zip!**=>destination` will unpack the entire archive saving the path information.
- `a.zip!a/b/c/**/*.*.dll=>dlls` will extract all .dll files from `a/b/c` and its subdirectories into the `dlls` 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="${basedir}/ivyconf.xml" />
<!--<ivy:cachelocal />-->
<ivy:retrieve pattern="${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:cachelocal />` 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"
  password="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`.

The properties `teamcity.auth.userId/teamcity.auth.password` store automatically generated build-unique values only intended to download artifacts within the build script. The values are valid only during the time the build is running. 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 will not be cleaned up by the `clean-up` process much like pinned builds.

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.

To add a snapshot dependency, on the **Dependencies** page of the build configuration settings, click **Add new snapshot dependency** and specify the following options:

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	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 Suitable Builds below. 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.
	 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.
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 this build if dependency has failed	If this option is enabled and the dependency build fails, the build of this build configuration will still be run. The build status will be set to "failed". If the option is not set, the build on the build configuration will fail with the "failed to start" error without running.
Run build on the same agent	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. If a build of B is already in the build queue, then TeamCity will not let any other build run on the agent before the build of B.

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 "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 another snapshot dependencies path 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). 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: **Checkout by label** set to **Client** instead of **Client mapping**
- Starteam: checkout mode option set to **view label** or **promotion date**

See also:

Concepts: [Dependent Build](#)

Build Dependencies Setup

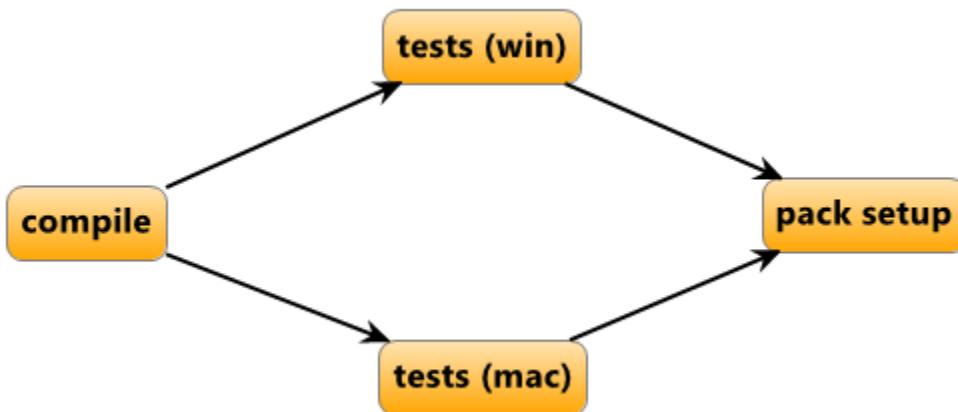
This page is intended to give you the general idea on how dependencies work in TeamCity based on an example. For the dependencies description, please see [Dependent Build](#).

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 Mac before you get the production build. The best workflow for this simple case would be to:

1. Compile your project
2. Run tests under Windows and Mac simultaneously on the same sources
3. Build a release version, again, 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** compilation, which means they should wait till compilation is ready.

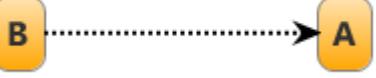
In this section:

- [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 even if dependency has failed](#)
 - [Run build on the same agent](#)
 - [Trigger on changes in snapshot dependencies](#)
 - [Parameters in dependent builds](#)
- [Miscellaneous Notes on Using Dependencies](#)

Basics

Generally known as "build pipeline", in TeamCity a similar concept is referred to as "[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.
	Snapshot dependency between 2 build configurations. Note, that the arrow shows the sequence of triggering build configurations, the build chain 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 in the TeamCity UI, you can find visual representation of build chains which are always displayed this way - according to the build chain flow.
	Artifact dependency. The arrow shows the artifacts flow, the dependency is configured in the opposite direction.

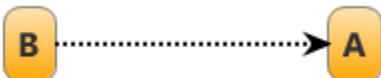
As you noticed, there are 2 types of dependencies in TeamCity: **artifact** dependencies and **snapshot** dependencies. In two words, the first one allows to use the output of one build in another, while the second one can trigger builds from several build configurations in a specific order, but on the 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.

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 correspond to the same moment).

When you have a number of build configurations 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 are run correctly under 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 much time to run, so you have to extract them into a separate build configuration, but you also need to make sure the same sources snapshot is used.

Build Chains in TeamCity UI

Once you have snapshot dependencies defined and at least one build chain was triggered, a new "Build Chains" tab appears among project tabs and among build configuration tabs, providing a visual representation of all related build chains and a way to re-run any chain step manually, using the same set of sources pulled originally.

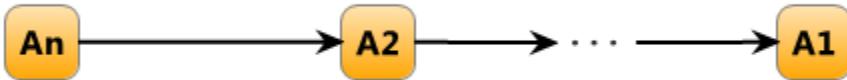


[Learn more](#)

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:



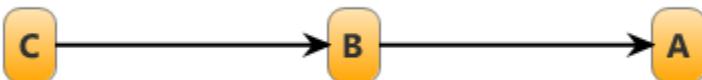
Here are the main rules:

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.

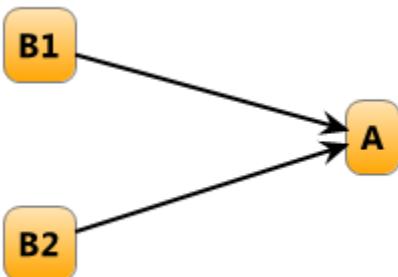
 Note, that if the build configurations connected with a snapshot dependency share the same set of VCS roots, all builds will run on the same sources. Otherwise, if the VCS roots are different, changes in the VCS will correspond to the same moment in time.

3. Once build C has successfully finished, build B starts, and so on. If build C failed, TeamCity won't further execute builds from the chain.

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 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 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 VCS.

Run build even if dependency has failed

When this option is enabled, a build of A will run after build B is finished, even if B failed.

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.

Trigger on changes in snapshot dependencies

Another option that alters triggering behavior within a build chain you can find in the [VCS build trigger options](#). It allows to trigger the whole build chain even if changes are detected in some further build configuration, not in the root.

Let's take a build chain from the first example: Pack setup--depends on-->Tests--depends on-->Compile.

Normally, the whole build chain is triggered when TeamCity detects changes in Pack setup, changes in Compile do not trigger the whole chain - only Compile is run. If you want the whole chain to be triggered on VCS change in Compile, add a VCS trigger with "Trigger on changes in snapshot dependencies" to your Pack setup configuration.

This won't change the order in which builds are executed in any way. This will only trigger the whole build chain, if there's a change in any of snapshot dependencies.

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 on 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
(mercurial) Wording	11 files ▾
(mercurial) TW-17252	▾ handle mercurial caches correctly
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, "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.

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):

- in a specific build (via [Run Custom Build](#) dialog)
- Build Configuration settings (the **Build 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 found in the project and its subprojects.
- Agent (<Agent home>/conf/buildAgent.properties file on 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.

See more in the corresponding sections: [Defining and Using Build Parameters in Build Configuration](#) and [Project and Agent Level Build Parameters](#).

See also:

[Administrator's Guide: Configuring Agent Requirements](#) | [Defining and Using Build Parameters in Build Configuration](#) | [Project and Agent Level Build Parameters](#) | [Predefined Build Parameters](#)

Defining and Using Build Parameters in Build Configuration

To learn about build parameters in TeamCity, refer to the [Configuring Build Parameters](#) page.

In this section:

- [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](#)

Defining Build Parameters in Build Configuration

On the **Build 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 Predefined Server Build Properties
VCS root and checkout rules settings	any of the properties that are passed into the build
VCS label pattern	system.build.number and Server Build Predefined Properties
Artifact dependency settings	only Predefined Server Build Properties

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.

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 [Build 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 Gradle use teamcity["<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 the build script.

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:

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.

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 Property Name	Environment Variable Name	Description
<code>teamcity.version</code>	<code>TEAMCITY_VERSION</code>	The version of TeamCity server. This property can be used to determine the build is run within TeamCity.
<code>teamcity.projectName</code>	<code>TEAMCITY_PROJECT_NAME</code>	The name of the project the current build belongs to.
<code>teamcity.buildConfName</code>	<code>TEAMCITY_BUILDCONF_NAME</code>	The name of the Build Configuration the current build belongs to.
<code>build.is.personal</code>	<code>BUILD_IS_PERSONAL</code>	Is set to <code>true</code> if the build is a personal one . Is not defined otherwise.
<code>build.number</code>	<code>BUILD_NUMBER</code>	The build number assigned to the build by TeamCity using the build number format. The property is assigned based on the build number format .
<code>teamcity.build.id</code>	<code>none</code>	The internal unique id used by TeamCity to reference builds.
<code>teamcity.auth.userId</code>	<code>none</code>	A generated username that can be used to download artifacts of other build configurations. Valid only during the build.
<code>teamcity.auth.password</code>	<code>none</code>	A generated password that can be used to download artifacts of other build configurations. Valid only during the build.
<code>build.vcs.number.<VCS root ID></code>	<code>BUILD_VCS_NUMBER_<VCS root ID></code>	<p>The latest VCS revision included in the build for the root identified. See Configuring VCS Roots for the <code><VCS root ID></code> 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.</p> <p> 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)</p> <p>In versions of TeamCity prior to 4.0, a different format for the VCS revision number when specified in a build number pattern was used: <code>{build.vcs.number.N}</code> where <code>N</code> is the VCS root order number in the build configuration. If you still need this to work, you can launch TeamCity with a special internal option:</p> <div style="border: 1px dashed blue; padding: 5px; margin-top: 10px;"><code>teamcity.buildVcsNumberCompatibilityMode=true</code></div>

Configuration Parameters

These are the parameters that other properties can reference (only if defined on the [Build 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
- VCS Properties
- Branch-Related Parameters
- Other Parameters

Dependencies Properties

These are properties provided by the builds the current build depends on (via a snapshot or an artifact dependency).

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 [server build](#) property of the build configuration with the given ID.

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
<code>teamcity.build.triggeredBy</code>	Since TeamCity 8.1, a human-friendly description of how the build was triggered
<code>teamcity.build.triggeredBy.username</code>	Since TeamCity 8.1, 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, `os.name` or `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>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>agent.work.dir</code>	The path of Agent Work Directory .
<code>agent.home.dir</code>	The path of Agent Home Directory .
<code>os.name</code>	The corresponding JVM property (see JDK help for properties description)
<code>os.arch</code>	The corresponding JVM property
<code>os.version</code>	The corresponding JVM property
<code>user.country</code>	The corresponding JVM property
<code>user.home</code>	The corresponding JVM property
<code>user.timezone</code>	The corresponding JVM property
<code>user.name</code>	The corresponding JVM property
<code>user.language</code>	The corresponding JVM property
<code>user.variant</code>	The corresponding JVM property
<code>file.encoding</code>	The corresponding JVM property
<code>file.separator</code>	The corresponding JVM property
<code>path.separator</code>	The corresponding JVM property
<code>DotNetFramework<version>[_x86 x64]</code>	This property is defined if the corresponding version(s) of .NET Framework is installed. (Supported versions are 1.1, 2.0, 3.5, 4.0)
<code>DotNetFramework<version>[_x86 x64]_Path</code>	This property value is set to the corresponding framework version(s) path(s)
<code>DotNetFrameworkSDK<version>[_x86 x64]</code>	This property is defined if the corresponding version(s) of .NET Framework SDK is installed. (Supported versions are 1.1, 2.0)
<code>DotNetFrameworkSDK<version>[_x86 x64]_Path</code>	This property value is the path of the corresponding framework SDK version.
<code>WindowsSDK<version></code>	This property is defined if the corresponding version of Windows SDK is installed. (Supported versions are 6.0, 6.0A, 7.0 , 7.0A, 7.1)
<code>VS[2003 2005 2008 2010]</code>	This property is defined if the corresponding version(s) of Visual Studio is installed

VS[2003 2005 2008 2010]_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 (sln).



- 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, the JavaDowser plugin looks for the installed JDK and JRE and when it finds them, it defines environment variables as described below.

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 found, the **JRE** variables are defined when the JRE found but the JDK is not found.

The `_x64` variables point to 64-bit java only; the variables without the `_x64` suffix may points 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 additional, 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 points 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
```

JavaDowser defines environment variables only if they are not already defined in the environment. So, if a started agent already has some of the mentioned above variables defined, the are not redefined by JavaDowser.

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>	none	Checkout directory used for the build.
<code>teamcity.build.workingDir</code>	none	Working directory 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>	none	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>	TEAMCITY_BUILD_PROPERTIES_FILE	Full name (including path) of the file containing all the system.* properties passed to the build. "system." prefix stripped off. The file uses Java properties file format (for example, special symbols are backslash-escaped).
<code>teamcity.build.changedFiles.file</code>	none	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.

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

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.

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 `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. With the build parameter's specification you can make your parameters more readable in the **Run Custom Build** dialog. Currently you can present parameters in following forms:

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	<p>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 Build 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.</p> <p>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.</p>

- simple text field with the ability to validate its value using regular expression;
- check box;
- select control;
- password field.

To add specification to a build parameter, click **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.
- **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, even on simply clicking the **Run** button; if *normal* is selected, the parameter will be shown as usual.

Depending on the specification's "type", there are additional settings.

Text	Pattern : In this field specify a Java-style regular expression to validate the field value.
Check box	Check box name : Title of the check box to be displayed in Run Custom Build dialog. Checked value/Unchecked value : Specify values for the parameter to have depending on the check box's state.
Select	Items : Specify a newline-separated list of items. Use following syntax <code>label => value</code> or <code>value</code> . Check Allow multiple if you want to enable multiple selection.

Manually Configuring Parameter Specification

Alternatively, you can manually configure a specification using specially formatted string with 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

Since TeamCity 8.1, you can view/edit typed parameters specification via REST API.

See also:

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 Configuring Build Parameters section.
Parameter Name	Specify the mandatory property or environment variable name.
Condition	Select condition from the drop-down list. <div style="background-color: #e0f2ff; padding: 10px;"><p> Some notes on how conditions work:</p><ul style="list-style-type: none">• 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.• 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.• 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.• is more than, is not more than, is less than, is not less than: These conditions only work with numbers.• matches, does not match: This condition will be true if the specified property matches/does not match the specified Regular Expression pattern.• 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 < 1.1g.</div>
Value	Is shown for some conditions that require value, for example: equals

Moreover, you can use the [Frequently used requirements](#) to quickly add a popular requirement.

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

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).

Let us consider an example of creating a meta-runner. More examples of meta-runners are available on GitHub: see [Meta-runner Power Pack for TeamCity 8](#).

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 **Build 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.



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.

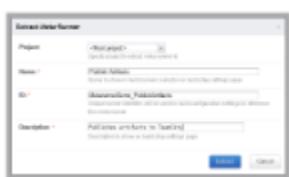
Verifying Build Configuration Works Properly

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 **Extract Meta-Runner** button in the build configuration settings sidebar:



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.



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:



The current meta-runner usages can be seen at the project **Meta-Runners** tab:



When a meta-runner is extracted from the web interface, 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 in the `TeamCity Data Directory\config\projects\<project ID>\pluginData\metaRunners` folder. The file should be named as follows: `<runner id>.xml`, where `<runner id>` is the ID of this build runner. The server will detect this definition and will load it on the fly.

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.

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

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](#))

Delete Build Configuration

When you delete a build configuration, TeamCity will remove its .xml configuration file. After the deletion, there is a default 24-hour timeout 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.

Working with Feature Branches

Feature Branches in distributed version control systems (DVCS) like Git and Mercurial allow you to work on a feature independently from the repository and commit all the changes for the feature onto the branch, merging the changes back 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.

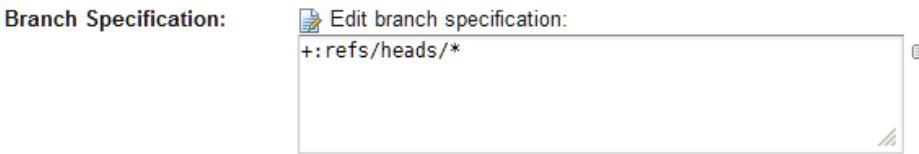
TeamCity support for feature branches is continuously increasing, starting from partial support in version 7.0 by [Branch Remote Run Trigger](#), that automatically starts a new personal build each time TeamCity detects changes in particular branches of the VCS roots of the build configuration; moving on to TeamCity 7.1, allowing you to automate the process and ensuring visibility of branches all over the interface. In TeamCity 8.0, the feature branches support is taken to the next level [with a number of improvements](#).

- Configuring branches
- Default branch
- Logical branch name
- Builds
- Changes
- Active branches
- Tests
- Triggers
- Dependencies
- Notifications
- Build configuration status
- Multiple VCS roots
- Build parameters
- Clean-up

Configuring branches

To start working with DVCS branches, you need to tell TeamCity which of them need to be monitored. This is done via the [branch specification](#) field of the VCS root which accepts a list of names or patterns of branch names to watch.

The syntax of the branch specification field is similar to [VCS checkout rules](#):



Newline-delimited set or rules in the form of +|-:branch name (with optional * placeholder)

For further details on branches specification, refer to the [Git](#) and [Mercurial](#) VCS roots description.

Everything that is matched by the wildcard will be shown as a branch name in the TeamCity interface (also known as [logical branch name](#)). For example, `+:refs/heads/*` will match `refs/heads/feature1` branch, but in the TeamCity interface you'll see `feature1` only as a branch name.

If you want shortened branch labels in builds, you can use extended syntax of branch specification like this:

```
+refs/heads/release-(7.0)
+refs/heads/release-(7.1)
```

In this case, TeamCity will use label 7.0 for builds from the `refs/heads/release-7.0` branch and 7.1 for builds from `refs/heads/release-7.1`, and so on.

Specification supports comments as lines beginning with #.

To use brackets in the branch name, you need to escape them. To do that, specify an escaping symbol as the first line in the specification. Let's say you want to track the `release-(7.1)` branch. The following branch specification does that:

```
#! escape: \
+release-\(7.1\)
```

The first line in this spec defines the escape symbol to use.

Note that you can also use parameters in the branch specification.

Once you've done 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 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.

Default branch

When configuring a Git or Mercurial VCS root, 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.

Logical branch name

A logical branch name is a branch name shown in the user interface for the builds and on build configuration level. It is calculated by applying a branch specification to the branch name from version control. For example, if the branch specification is defined like this:

```
+refs/heads/*
```

then a part matched by * is a logical branch name. For the default branch <default> can be used.

Builds

Builds from branches are easily recognizable in the TeamCity UI, because they are marked with a special label:

Youtrack branches

master	#309	Success	No artifacts	No changes
develop	#321	Success	No artifacts	anna.zhdan (1)
explain	#303	Success	No artifacts	Sergey Bankevi... (1)

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.



If the **Show builds** and **Show graph** options are enabled in the change log, TeamCity will display build markers on the graph.

Active branches

In a build configuration with configured branches, the Overview page shows active branches.

A branch 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).

Since TeamCity 7.1.1 you can change the parameters. This can be done 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.

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.

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 the other build configurations, when a build in a branch is triggered, all builds from the chain will be marked with this branch too.

Starting from TeamCity 8.0, 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 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/devel/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](#).

See also:

[Administrator's Guide: Git \(JetBrains\) | Mercurial](#)

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
 - Parameters
 - Comment
- 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](#) 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.
- **all enabled compatible agents** — select 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

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 should be taken.

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.

- **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 builds run with the changes earlier than the latest available is marked as a [History Build](#).



If you have branches in your build configuration (or in snapshot dependencies of this build configuration), you can choose a branch to be used for the custom build in this dialog.

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

Add an optional comment to the build.



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

Ordering 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.

- Manually Reordering Builds in Queue
- 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.

To move a build configuration to the top position, click the arrow button next to the build sequence number .

Managing Build Priorities

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 configurations using **Del** box, then select **Remove selected builds from the queue** from the **Actions** menu. 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.

{anchor:queueSizeLimit}

Limiting Maximum Size of Build Queue

Since TeamCity 8.1.2, 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.

When a test is muted, it is **still run** in the future builds, but its failure does not fail the build (by "at least one test failed" build failure condition). The test can be unmuted manually on specific date or after 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 build green when there are failing but muted tests. Please ensure that build does not fail because of other build failure conditions (like "Fail if build process exit code is not zero") in case the only errors encountered were tests failures. See also related issue [TW-16784](#).



The screenshot shows a build history entry for build #6.0.1717.0. The status bar indicates "Tests passed: 2075, ignored: 63, muted: 3". A mouse cursor is hovering over the "muted" count. Below the status bar, there's a "View build results" button. To the left, there's a "Win32 Notify" icon and the text "Last built: 9 minutes ago".

How to mute tests

Only users with the **Mute/unmute problems in project** permission can perform it. By default, these are **Project administrator** and **System administrator**

You can mute a test failure from:

- **Projects** page
- Project overview page
- Build Configuration overview page
- Current Problems tab



On the build results page you can select several test failures (or all) to be muted or unmuted:

! 30 tests failed (★ 30 new)

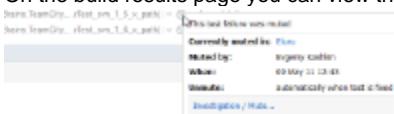
[Collapse All](#) | [Expand All](#) | Bulk operations: [Investigation / Mute...](#) | [Cancel](#)

▼ <Root> (25)

- ★ [OptionParser should read options from file when --options](#) |
- ★ [OptionParser should read spaced and mul...ns from file whe](#)
- ★ [OptionParser should support c option](#) |
- ★ [OptionParser should support queens colour option](#) |
- ★ [OptionParser should support us color option](#) |
- ★ [OptionParser when attempting a focussed...c should suppo](#)
- ★ [SpecParser should find context and desc... and example fo](#)
- ...

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 mute status:



- From the Project Home page you can navigate to the **Muted Tests** tab to view all the test failure muted in all build configurations within project.

This feature is useful when some tests fail for some known reason, but it is currently not possible to fix them. For example, 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 required functionality is not yet written (TDD). In these cases you can mute such failure and avoid unnecessary disturbance of other developers.

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 "last successful" dependencies.
- For a running personal build - mark current failures non-relevant to allow pre-tested commit to pass (if user has permission to do this).

"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 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. To learn more about the charts, refer to [Statistic Charts](#). This page describes how to modify pre-defined project-level charts, and 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](#)
- [Adding Custom Project- and Build Configuration-level Charts](#)

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/<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).

Adding Custom Project- and Build Configuration-level Charts

Please refer to the [Custom Chart](#) page for details.

See also:

Concepts: [Code Coverage](#) | [Code Inspection](#) | [Code Duplicates](#)
User's Guide: [Statistic Charts](#)
Extending TeamCity: [Build Script Interaction with TeamCity](#)

Archiving Projects

If a project is not in active development state, you can *archive* it from its settings page. When *archived*:

- All project's build configurations are automatically paused.
- Automatic checking for changes in 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 list of build configurations for remote run.

By default, permissions to archive projects are given to project and system administrators.

If you need to unarchive a project, you can do it from the [Administration | Projects | Archived projects](#) tab where all archived projects are displayed.

See also:

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

Managing Licenses

In this section:

- [Licensing Policy](#)
- [Third-Party License Agreements](#)

Licensing Policy

This page covers:

- Editions
 - Number of Build Configurations
 - Number of Agents
- Managing Licenses
- Valid TeamCity Versions
- License Expiration
- Enterprise Edition License Types
- 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

You can review TeamCity license agreement on the [official web site](#) or in the footer of the installed TeamCity server web UI.

New licenses can be purchased 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](#).

Editions

There are two editions of TeamCity: **Professional** and **Enterprise**.

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 20, and since TeamCity 8.0 it can be extended by 10 with each agent license key added. You can install several servers with 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.

Each TeamCity edition comes bundled with 3 build agents. Each additional **build agent** above the bundled 3 requires a new build agent license key in both editions.

The editions are equal in all the features except for the maximum number of build configurations allowed.

The same TeamCity distribution and installation is 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.

Number of Build Configurations

The Professional edition allows 20 build configurations per server. Since TeamCity 8.0, each entered build agent license key gives you 10 more build configurations in addition to one more agent. All build configurations are counted (i.e. including those in archived projects).

The Enterprise edition has no limit on the number of build configurations.

Number of Agents

Each TeamCity edition comes bundled with 3 build agents. These 3 agents are bound to the TeamCity server installation and not to the server license key. More build agents can be added by purchasing additional agent license keys.

Generally, a server license key does not include any agent licenses. The 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.

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 Evaluation (limited time) license to run a server for testing/non production purposes.

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.

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 an [upgrade](#).

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 on enterprise 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 exceed the limits imposed by the valid licenses, the server stops to start any builds and displays a warning message to all users in the web browser.

Enterprise Edition License Types

The Enterprise edition requires one of the following types of licenses:

- **Commercial**
 - **Perpetual** — no expiration date
 - **Evaluation** — has an expiration date and provides an unlimited number of agents and build configurations. To obtain the evaluation license, please 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.

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 is an alphabetical list of third-party libraries distributed with TeamCity:

Product	License
Acegi Security	Apache
Apache Ant	Apache
Apache Commons libraries	Apache
Apache HttpComponents	Apache
Apache Lucene	Apache
Behaviour	BSD

Byteman	GNU LGPL
CassiniDev	Ms-PL
Core4j	Apache 2.0
cglib	Apache
CVS client library	CDDL
CyberNekoHTML Parser	Apache-style
DHTML Tip Message	DHTML Tip Message
EhCache	Apache
Expat XML Parser Toolkit	MIT
FormattedDataSet API	BSD
FreeMarker	BSD-style
Ganymed SSH-2 for Java	BSD-style
google-gson	Apache
Guava: Google Core Libraries	Apache
Highlight.js	BSD-like
HSQLDB	BSD
Ivy	Apache
Jackson	Apache
Jakarta	Apache
Jakarta-ORO	Apache
JAMon	BSD-like
JAXB reference implementation	CDDL v1.0
Java Mail	CDDL v1.0
Java Native Access	LGPL
Java Service Wrapper, version 3.2.3	Java Service Wrapper 3.2.3 License
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
JUnit	CPL
J2SSH Maverick	J2SSH Maverick

Log4j	Apache
Log4net	Apache
Maven	Apache
Microsoft.Web.Infrastructure	MVC-3-EULA
Mono.Cecil	MIT/X11
NanoContainer	NanoContainer
NUnit	NUnit
OData4j	Apache
opencsv	Apache
pack:tag	LGPL
Paul Johnson's MD5	BSD
PicoContainer	PicoContainer
PocketHTTP	Mozilla
PocketXML-RPC	Mozilla
PostgreSEL Data Base Management System	BSD
Prototype	MIT
pty4j	EPL
Raphaël	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
Tomcat	Apache
Tom Wu's jsbn	BSD
Trove4j	GNU LGPL
Underscore.js	MIT
user-agent-utils, version 1.12	user-agent-utils
Waffle	EPL
WebActivator	MS-PL
Winp, version 1.7 (patched)	MIT
Xerces2 Java Parser	Apache
XML Pull Parser	IU Extreme! Lab
XML-RPC.NET	MIT X11

XStream	XStream
Missing Link Ant Tasks	Apache

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) 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).

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. All rights reserved.

Maverick Copyright © 2001 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>).

Shai is Copyright Cenqua Pty Ltd.

SilverStripe Unobtrusive Javascript Tree Control Copyright © 2006-7, SilverStripe Limited - 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.

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 © 2006-2007, 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:

- Mapping External Links in Comments
- External Changes Viewer
- Integrating TeamCity with Issue Tracker

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">$2</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 Atlassian Fisheye.

To enable external viewer for changes, you should 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.

Detailed example of the configuration file including description of available formats, variables, and other parameters can be found in `change-viewers.properties.dist` file in `<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 on 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 :



- TeamCity file diff page:



Integrating TeamCity with Issue Tracker

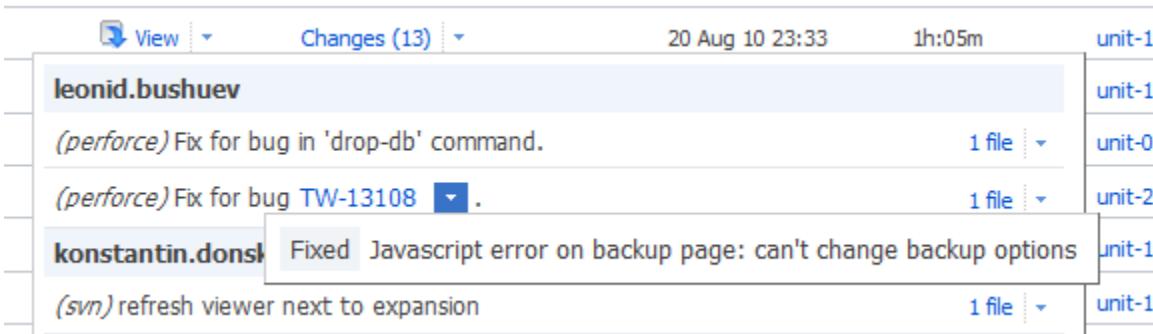
In this section:

- Dedicated Support for Issue Trackers
- Recommendations on Using Issue Tracker Integration
- Enabling Issue Tracker Integration
- Integrating TeamCity with Other Issue Trackers

Dedicated Support for Issue Trackers

TeamCity supports Jira, Bugzilla and YouTrack issue trackers out of the box. Refer to the [Supported Platforms and Environments](#) page for the list of supported versions.

When integration is configured, TeamCity automatically transforms an issue ID (=issue key in JIRA) mentioned in VCS comment into a link to the corresponding issue and allows to see basic issue details on hover over an icon displayed near the issue ID.

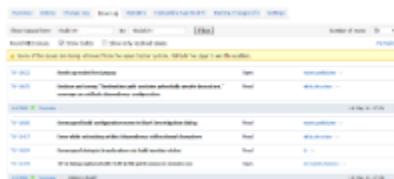


The screenshot shows a TeamCity build results page. At the top, there's a navigation bar with 'View' and 'Changes (13)'. Below it, the build summary shows '20 Aug 10 23:33' and '1h:05m'. To the right, there are four vertical bars labeled 'unit-1', 'unit-1', 'unit-0', and 'unit-2'. The main area displays a list of changes:

- leonid.bushuev**
(perforce) Fix for bug in 'drop-db' command.
1 file
- (perforce)** Fix for bug [TW-13108](#).
1 file
- konstantin.donsk** Fixed Javascript error on backup page: can't change backup options
1 file
- (svn)** refresh viewer next to expansion
1 file

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 particular range of builds and view which issues were mentioned in comments, and their state.



The screenshot shows the 'Issues' tab of a build configuration's home page. It lists several issues with their status and resolution time:

Issue	Status	Resolution time
TW-13108	Open	20 Aug 10 23:33
TW-13108	Resolved	20 Aug 10 23:33
TW-13108	Resolved	20 Aug 10 23:33
TW-13108	Resolved	20 Aug 10 23:33
TW-13108	Resolved	20 Aug 10 23:33
TW-13108	Resolved	20 Aug 10 23:33
TW-13108	Resolved	20 Aug 10 23:33
TW-13108	Resolved	20 Aug 10 23:33

Recommendations on Using Issue Tracker Integration

To get best results out of issue tracker integration we recommend to:

- When committing changes to your version control **always mention issue id (issue key)** related to the fix in the comment to commit.
- Resolve issues when they are fixed (time of resolve does not really matter).
- Use [Issue Log](#) of a build configuration to get issues related to builds; turn "Show only resolved issues" option on to show only issues fixed in the builds.

Enabling Issue Tracker Integration

To enable integration, you need to create connection to your issue tracker on the [Administration | Issue Tracker](#) page. The described below settings are common for all issue trackers:

Connection type	Select type of your issue tracker from the list.
Display name	Symbolic name that will be displayed in TeamCity UI for the issue tracker.
Server URL	Issue tracker URL
Username/Password	Username/password to log in to the issue tracker, if it requires authorizing.

In addition to these general settings you also need to specify which strings should be recognized by TeamCity and converted to links to issues in your issue tracker. For details, please refer to corresponding section:

- YouTrack
- JIRA
- Bugzilla



Requirements:

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:

- TeamCity server should have 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 should have enough 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.

Integrating TeamCity with Other Issue Trackers

To integrate TeamCity with an issue tracker other than Jira/YouTrack/Bugzilla, you can configure TeamCity to turn any issue tracker issue ID references 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 should be recognized as references to issues in your tracker.

For Bugzilla, you need to specify **Issue Id Pattern**: a [Java Regular Expression](#) pattern to find issue ID in the text. The matched text (or the first group if there are groups defined) is used as issue number. Most common case seems to be `#(\d+)` - this will extract 1234 as issue ID from text "Fix for #1234".

Requirements

If username and password are specified, you should have Bugzilla XML-RPC interface switched on. This is not required if you use anonymous access to Bugzilla without username and password.

Known Issues

There are several known issues in Bugzilla regarding XMLs generated for the issues, which makes it hard to communicate to. However it usually can be fixed by tweaking 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 `bugzilla.dtd` file, instead of URL. To fix that you need to set 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 you need to make the `type_id` attribute `#IMPLIED` (optional) in the `bugzilla.dtd` file. The issue and the workaround described in detail [here](#).

See also:

[Concepts: Supported Issue Trackers](#)

[Administrator's Guide: Integrating TeamCity with Issue Tracker](#)

JIRA

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 specify space-separated list of [Project keys](#).

For example, if a project key is **WEB**, then when an issue key like **WEB-101** is mentioned in a VCS comment, it'll be resolved to a link to corresponding issue.

Requirements

For the Jira integration to work, you should have Jira XML-RPC interface switched on (see instructions in [Jira documentation](#)).

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 strings should be recognized as references to issues in your tracker.

For YouTrack, you need to specify space-separated list of [Project Ids](#). For example, if a project id is **TW**, then when an issue id like **TW-18802** is mentioned in a VCS comment, it'll be resolved to a link to 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.

However, to be able to use these features you need to [configure YouTrack](#).

See also:

Concepts: [Supported Issue Trackers](#)

Administrator's Guide: [Integrating TeamCity with Issue Tracker](#)

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:

- [Managing Users](#)
 - [Creating New User](#)
 - [Editing User Account](#)
 - [Assigning Roles to Users](#)
- [Managing User Groups](#)
 - [Creating New Group](#)
 - [Editing Group Settings](#)
 - [Adding Multiple Users to Group](#)

Managing Users

Creating New User

The [Administration | Users](#) page provides the **Create user account** option.

When creating a user account, 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 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.

General tab

The tab provides several panes allowing you to modify various user account settings:

The *General* pane 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.

The *Authentication Settings* pane allows editing usernames for different authentication modules such as LDAP and Windows Domain.

The *Version Control Username Settings* pane allows viewing and editing the default usernames for different VCS used by the current user. The names set here will be used to:

- show builds with changes committed by a user with such VCS username on the [My Changes page](#) page, called *Changes since TeamCity 8.1*,
- 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](#).

Watched Builds and Notifications displays the [notification rules](#) configured for this user account.

Groups tab

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 at the [Server Configuration](#) page.

Use this tab to view the roles assigned to the user directly and inherited from groups. The roles assigned directly can be modified/removed here.

Notification Rules

Please, refer to [Subscribing to Notifications](#) for details.

Assigning Roles to Users



To be able to grant roles to users on per-project basis, enable per-project permissions on the [Administration|Global Settings](#) page.

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.

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.

Managing User Groups

Creating New Group

Open the **Administration | Groups** page and click **Create new group**.

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 list of users, roles and permissions and notification settings.



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 and Groups** page, select the users, click the **Add to groups** button, and check the groups where these users should be added. 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 modify the existing roles and create new ones in the TeamCity Web UI using the [Administration | Roles](#) link (in the User Management section of Settings).

At this page you can:

- Create new roles.
- Delete existing roles.
- Add/delete permissions from existing roles.
- Include/exclude one role permissions.

Note, that 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 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). Each user can [select the events to receive notifications](#). The notification messages can be customized globally on per-server basis.

Notifications can also be received via Atom/RSS syndication feeds, but since feed use "pull" model for receiving notifications instead of "push", some of the approaches are different for the feeds.

Notifications Lifecycle

TeamCity supports a set of events that can generate user notifications (such as build failure, investigation state changes, etc). On event occurrence, for each notificator type, TeamCity processes notification settings for all the users to define users that the notification should be sent to.

When the set of users is determined, TeamCity fills the notification model (the objects relevant to the notification 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 output values (e.g. notification message text). The output values are then used by the notificator to send the message. Each notificator 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 per-user basis.

The output values defined by the template are then used by the notificator to send notification to the selected users.

Customizing Notifications Templates

Notification Templates Location

Each of the bundled notificators has a directory under `<TeamCity data directory>/config/_notifications/` which stores FreeMarker (.ftl) templates. There are also .dist files that store default templates. Each notification type evaluates a template file with corresponding name. The template files can be modified while the server is running. By default server checks for changes in the files each 60 seconds, but this can be changed by setting `teamcity.notification.template.update.interval` internal property to the desired number of seconds.

If there an error occurs during 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>.

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** - (optional) HTML text of the email message to send. It will be included together with plain text part of the message
- **headers** - (optional) Raw list of additional headers to include into 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

The Atom/RSS feeds template differs from the others. For the details, please refer to the [dedicated section](#).

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 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)

See also [TW-8621](#) on including build log messages into the template.

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 **os.name** property and set the **starts with** condition and the **linux** value.



On the **Agent Requirements** page, click the **Frequently used requirements** link to add the regularly used requirements.

See also:

Concepts: [Build Agent](#) | [Agent Requirements](#) | [Run Configuration Policy](#)

Administrator's Guide: [Triggering a Custom Build](#)

Patterns For Accessing Build Artifacts

If you need to access the artifacts in your builds, consider using TeamCity's built-in Artifact Dependency feature.

This section covers URL patterns that you may use to download build artifacts from outside of TeamCity.

Please consider using [REST API](#) which provides more rich build selection facilities and allows for artifacts listing.

The other URLs described below are mostly preserved for backward-compatibility with previous TeamCity versions and for some specific functionality.

You may also download artifacts from TeamCity using [Ivy](#) dependency manager.

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 build id, use:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_ID:id/ARTIFACT_PATH
```

To download artifacts by build number, use:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_NUMBER/ARTIFACT_PATH
```

To download artifacts from last build with 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 artifact on TeamCity server. This path may contain a `{build.number}` pattern which will be replaced with build number of the build whose artifact is retrieved.
By default, archive with all artifacts does not include `hidden artifact`. To include them, add "`?showAll=true`" to 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 `branch=<branch_name>` parameter to the corresponding URL.

Obtaining Artifacts from an Archive

TeamCity allows to obtain 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 an zip/jar/tar.gz archive on TeamCity server.

Following archive types are supported (case insensitive):

- `.zip`
- `.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.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 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}` as a shortcut to current build number in the artifact file name.

For example:

```
http://teamcity.yourdomain.com/repository/download/MyConfExtId/.lastFinished/TeamCity-{build.number}.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](#).

TeamCity supports running .NET builds using MSBuild and NAnt build runners under Mono framework as well as under .NET Frameworks.

Tests reporting tasks are also supported under Mono.

Mono Platform Detection

When a build agent starts it detects 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 agent-side, the following properties are set:

- **Mono** — path to `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 custom directory), you can make these properties available for 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. Mono version is read from `HKLM\SOFTWARE\Novell\Mono\DefaultCLR`
2. Frameworks paths are extracted from `HKLM\SOFTWARE\Novell\Mono\ %MonoVersion%`

3. Platform architecture is detected by analyzing `mono.exe`

Mac OS X Specifics

1. Framework is detected automatically from `/Library/Frameworks/Mono.framework/Versions`
2. The highest version is selected
3. Frameworks path are extracted from `/Library/Frameworks/Mono.framework/Versions/%MonoVersion%/lib/mono`
4. 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. Frameworks paths are extracted from "`pkg-config --variable=prefix mono`" and "`pkg-config --variable=libdir mono`"
3. Platform arch is detected by analyzing `PREFIX/bin/mono` executable.

You can force Mono to be detected from custom location by adding `PREFIX/bin` directory to the beginning of the `PATH` and updating `PKG_CONFIG_PATH` (described in [pkg-config\(1\)](#)) with `PREFIX/lib/pkgconfig`

Supported Build Runners

Both **NAnt** and **MSBuild** runners support using Mono framework to run a build (MSBuild as `xbuild` in mono).

See also:

[Administrator's Guide: NAnt | MSBuild](#)

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 Maven model display on the "Maven" build configuration tab.

During the process, TeamCity uses usual Maven logic for finding the `settings.xml` files with several differences (see below).

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 variable and `maven.home` JVM option are used - those set for the TeamCity server process),

For the Maven *user-level* settings, TeamCity uses settings defined in the **User settings selection** section of the Maven build step of the build configuration (if there are several, settings from the first Maven step are used).

Here is the explanation for the **User settings selection** options:

If the `<Default>` value is selected, TeamCity searches the following locations for the `settings.xml` file (listed in order of priority):

1. `<TeamCity Data Directory>/system/pluginData/maven/settings.xml`
2. `<User Home>/ .m2/settings.xml` (Home directory of the user which TeamCity server process runs under is used)

If the `<Custom>` value is selected, the file should be available both on the server and all the agents where the build will be run.

If one of pre-uploaded settings is selected, TeamCity automatically uses the specified file content both on the server and agents.



Since TeamCity 8.1, Maven settings are defined on the project level: the **Project Settings** page|the **Maven Settings** tab. The settings will be available in the current project and its subprojects. The settings are stored in the `<TeamCity Data Directory>/config/projects/%projectId%/pluginData/mavenSettings` directory. To override the inherited settings, in the 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](#).

See also:

Tracking User Actions

TeamCity logs user actions into the Audit log, which is available at the [Administration | Audit](#) page. Here you can find 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.

Since project and build configuration settings are stored on the disk in plain xml, 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.

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 Platfrom-based IDEs. These include IntelliJ IDEA, RubyMine, PhpStorm, WebStorm and others.

See a separate [section](#) on the list of supported versions.

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 form 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...** from the main menu.
2. Open **Plugins** section under **IDE Settings** to invoke the **Plugins** dialog.
3. On the **Plugins** dialog, open the **Available** tab or click **Install JetBrains plugin...** to view the list of available plugins.
4. Select the TeamCity plugin, click the **Install** button.
5. Restart the IDE.
6. Log in into TeamCity server from the plugin
7. Invoke **Update** command in **TeamCity** menu to install the plugin version matching the server version.

Installing the Plugin Manually

The plugin for IntelliJ platform can be downloaded from the **TeamCity Tool** 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. Under the IntelliJ Platform Plugin section in the **TeamCity Tools** area, click the **download** link, and save the archive.
3. Make sure that IDE is not running and unzip the archive into the IDE user plugins [directory](#).

Plugins directory for IntelliJ IDEA is located in:

- Windows: C:\Documents and Settings\<username>\.IntelliJIdea<vers.>\config\plugins
- OS X: \$HOME/Library/Application Support/IntelliJIDEA<vers.>
- Linux/Unix: \$HOME/.IntelliJIdea<vers.>/config/plugins

Plugins directory for RubyMine is located in:

- Windows: C:\Documents and Settings\<username>\.RubyMine<vers.>\config\plugins
- OS X: \$HOME/Library/Application Support/RubyMine<vers.>
- Linux/Unix: \$HOME/.RubyMine<vers.>/config/plugins

All additional information on how to work with 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"

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 build log,
- viewing failed tests of a particular build,
- navigating to TeamCity web interface,
- starting investigation of a build failure,
- viewing server-provided code coverage results run on TeamCity using IDEA or EMMA code coverage engine: "<Main Menu>/TeamCity/Code Coverage Data...".
- comparing personal patch content with workspace resources,
- viewing compilation errors,
- downloading patch to IDE from the TeamCity server,
- shelving changes,
- re-running tests failed on the TeamCity agent locally,
- support for P4Eclipse up to 2012.3 and Eclipse EGIT 0.9 - 3.2.0

Installing the Plugin

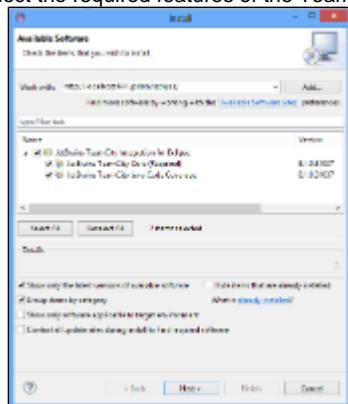
The TeamCity Eclipse plugin version must correspond to the version of the TeamCity server it connects to. Connections to TeamCity servers with different versions are generally not supported.

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, [1.10.x update site](#), [1.8.x update site](#), [1.6.x update site](#).
- **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 1.5 or newer:** Eclipse must be run under JDK 1.5 or newer 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. On the **General** tab, locate the **TeamCity Tools** section.
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.

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 TeamCity server (see [more](#)) to MS Visual Studio (requires dotCover of supported version installed in Visual Studio),
- viewing recently committed changes and personal builds with their build status in MyChanges tool window,
- opening build failure details in MS Visual Studio from the TeamCity web UI,
- viewing failed tests' details for a build,
- re-running tests failed in the TeamCity build locally via [ReSharper](#) test runner (requires ReSharper 5.0 installed),
- navigation from IDE to build results web page,
- re-applying changes sent in Remote Run or Pre-tested commit to the working directory.

For detailed instructions on how to use the add-in, refer to the TeamCity Help section, embedded in Microsoft Visual Studio's Help System.



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 current Visual Studio project.

Installing the Add-in

TeamCity VS add-in version should correspond to the version of the TeamCity server it connects to. Connections to TeamCity servers with different versions are generally not supported.

Download the add-in available in the **TeamCity Tools** section of the **My Settings&Tools** page (to open the page click the arrow next to your username in the top right corner of the TeamCity web UI and select **My Settings&Tools**) and follow the installation wizard. Please, close all running instances of Visual Studio before starting add-in installation (initial or upgrade).

Requirements

Please see [Supported Platforms and Environments](#) page for the system requirements for integrations with different version control systems or coverage tools.

See also:

Related blog posts: [TeamCity plugin for Visual Studio](#)@[TeamCity blog](#)

Windows Tray Notifier

The Windows Tray Notifier is a utility which allows monitoring the status of the specific build configurations in the system tray via popup alerts and status icons.

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.

For general instructions on using Windows Tray Notifier, please refer to the [Working with Windows Tray Notifier](#) page.

See also:

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

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

Installing Tools: [Working with Windows Tray Notifier](#)

Working with Windows Tray Notifier

To launch Windows Tray Notifier, run the **Start > Programs > Windows Tray Notifier** menu.

When the application started, you need to connect and log in to your server:

1. Specify the server's URL



2. Specify your credentials to log in:



When Windows Tray Notifier is launched, the status icon in 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
	Some team member has started investigation of 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 **Notify when the first 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.

Quick View Window

When you click the Notifier icon, a **Quick View** window opens:



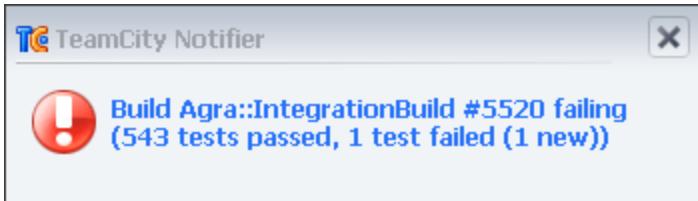
Click the *results* or *changes* links to navigate to the **Build Results** and **Changes** pages in TeamCity web interface, respectively, and investigate the desired issues deeper.

If you right-click the 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.

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 more details and investigate the desired issues.

See also:

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

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

[How To: Watching Several TeamCity Servers With 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, 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
- Accessing Server by HTTP
- Including Third-Party Reports in the Build Results

Plugin Development

- Plugin Types in TeamCity
- Bundled Development Package
- Developing Plugins Using Maven
- Open API Changes
- Plugins Packaging
- Server-side Object Model
- Agent-side Object Model
- Extensions
- Web UI Extensions
- Plugin Settings
- Development Environment

- 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

[Open API Javadoc](#)

[Open API Javadoc \(ver. 5.0.x\)](#)

[Open API Javadoc \(ver. 4.5.x\)](#)

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



If you use MSBuild build runner, you can use [MSBuild Service Tasks](#).

In this section:

- Service Messages
 - Common Properties
 - Message Creation Timestamp
 - Message FlowId
 - Blocks of Service Messages
 - Reporting Messages For Build Log
 - Reporting Compilation Messages
 - Reporting Tests
 - Reporting .NET Code Coverage Results
 - 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
- `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 used to pass commands/build information to TeamCity server from the build script. In order to be processed by TeamCity they should be printed into standard output stream of the build (otherwise, if the output is not in the service message syntax, it should appear in the build log). A single service message should not contain a newline character inside it, it should not span across multiple lines.

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=value'WSPpropertyName_IDOWSP=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****propertyName**OWSP=**value**'_blocks

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	Should be escaped 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' ...]
```

Timestamp format is "yyyy-MM-dd'T'HH:mm:ss.SSSZ" or "yyyy-MM-dd'T'HH:mm:ss.SSS" (or "yyyy-MM-dd'T'HH:mm:ss.fffzzz" for .NET `DateTime`), 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' ...]
```

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' ...]
```

Blocks of Service Messages

Blocks are used to group several messages in the build log.

Block opening:

```
##teamcity[blockOpened name='<blockName>']
```

Block closing:

```
##teamcity[blockClosed name='<blockName>']
```



Please note that when you close the block all inner blocks become closed automatically.

Reporting Messages For Build Log

You can report messages for 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" checkbox is checked on build configuration "Build Failure Conditions" page. For example:

```
##teamcity[message text='Exception text' errorDetails='stack trace' status='ERROR']
```

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 compilation error.

Reporting Tests

To use TeamCity's on-the-fly test reporting, 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.

Here is the list of supported test service messages:

Test suite messages:

Test suites are used to group tests. TeamCity display tests grouped by suite on the Tests tab of the build results and in other places.

```
##teamcity[testSuiteStarted name='suite.name']
<individual test messages go here>
##teamcity[testSuiteFinished name='suite.name']
```

All the individual test messages should appear between `testSuiteStarted` and `testSuiteFinished` (in that order) with the same `name` attributes.

Suites may also be nested.

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 `testFailed` message is not present, the test is regarded as successful.

duration (optional numeric attribute) - sets the test duration 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).
- Currently, the test-related service messages **cannot** be output with Ant's `echo` task until `flowId` attribute is specified.

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 should be reported under the same name in every build). e.g. it's highly unrecommended to include absolute paths in the reported test names.

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 `testIgnored` message can be reported without 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']
```

`testStdOut` and `testStdErr` service messages report the test's standard and error output to be displayed in TeamCity UI. There should be no more than single `testStdOut` and single `testStdErr` message per test.

Alternative, but less reliable approach is to use `captureStandardOutput` attribute of `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 test with the name "*testname*" has failed. Only single `testFailed` message should appear for a given test name.
`message` contains a textual representation of the error.
`details` contains detailed information on the test failure, typically a message and an exception stacktrace.
`actual` and `expected` attributes should only be used together with `type='comparisonFailure'` and can be used for reporting 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='suite.name']
##teamcity[testSuiteStarted name='nested.suite']
##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='nested.suite']
##teamcity[testSuiteFinished name='suite.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.

Publishing Artifacts while the Build is Still in Progress

You can publish build artifacts, while the build is still running, right after the artifacts are built.
For this you need to output the following line:

```
##teamcity[publishArtifacts '<path>']
```

And the files matching the `<path>` will be uploaded and visible as artifacts of the running build. `<path>` should adhere to the same rules as [Build Artifact](#) specification on the Build Configuration settings.

The message should be printed after all the files are ready and no file is locked for reading. Artifacts uploading happens in background and can take time. Please make sure the matching files are not deleted till the end of the build. (e.g. you might put them in a directory that is cleaned on next build start, `temp` directory or use `Swabra` to clean them after the build.)



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 corresponding build and on the build results page.

To log single progress message use:

```
##teamcity[progressMessage '<message>']
```

This progress message will be shown until another progress message occurs or until next target starts (in case of Ant builds).

If you wish to show progress message for a part of a build only you can use:

```
##teamcity[progressStart '<message>']
...some build activity...
##teamcity[progressFinish '<message>']
```



The same message should be used for both `progressStart` and `progressFinish`. This allows nesting of progress blocks. Also note that in case of Ant builds progress messages will be replaced if Ant target starts.

Reporting Build Problems

To fail a build directly from the build script, a build problem should be reported. Build problems appear on the build results page and also affect the build status text.

To add build problem to a build use the service message:

```
##teamcity[buildProblem description='<description>' identity='<identity>']
```

where:

- `description` - (mandatory) a human-readable plain text describing the build problem. By default `description` appears in the build status text and in the list of build's problems. Limited to 4000 symbols, will be truncated if exceeds.
- `identity` - (optional) a unique problem instance id. Different problems should have different id, same problems - same id. 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. If omitted, `identity` is calculated based on `description` text.

Reporting Build Status

TeamCity allows changing the **build status text** from the build script. Unlike [progress messages](#), this change persists even after 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 have been used for changing build status to failed. However since TeamCity 7.1 `buildProblem` service message should 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 build is running and will also affect 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 dedicated service message in your build script, you can dynamically update some build parameters right from a build step, so that following build steps will run with modified set of build parameters.

```
##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.](#)

The changed build parameters will also be available in the following build steps and in the dependent builds as `%dep.*%` properties.

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>']
```

The key should not be equal to any of [predefined keys](#).
The value should be a positive integer value.

Disabling Service Messages Processing

If you need for some reason to disable searching for service messages in output, you can disable 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 support flowId attribute and will ignore only the messages with the same flowId.

Importing XML Reports

In addition to UI [Build Feature](#), XML reporting can be configured from within the build script with the help of `importData` service message. Also, the message supports importing of previously collected code coverage and code inspection/duplicates reports.

The service message format is:

```
##teamcity[importData type='typeID' path='<path to the xml file>']
```

where `typeID` can be one of the following (see also [XML Report Processing](#)):

typeID	Description
Testing frameworks	
junit	JUnit Ant task XML reports
surefire	Maven Surefire XML reports

nunit	NUnit-Console XML reports
mstest	MSTest XML reports
gtest	Google Test XML reports
Code inspection	
checkstyle	Checkstyle inspections XML reports
findBugs ²⁾	FindBugs inspections XML reports
jslint	JSLint XML reports
ReSharperInspectCode ¹⁾	ReSharper inspectCode.exe XML reports
FxCop ¹⁾	FxCop inspection XML reports
pmd	PMD inspections XML reports
Code duplication	
pmdCpd	PMD Copy/Paste Detector (CPD) XML reports
DotNetDupFinder ¹⁾	ReSharper dupfinder.exe XML reports
Code coverage	
dotNetCoverage ^{1) 3)}	XML reports generated by dotcover, partcover, ncover or ncover3

Notes:

¹⁾ only supports specific file in the "path" attribute

²⁾ also requires findBugsHome attribute specified pointing to the home directory of installed FindBugs tool.

³⁾ also requires tool='<tool name>' service message attribute, where <tool name> is one of: dotcover, partcover, ncover or ncover3.

- If not specially noted, the report types support Ant-like wildcards in the path attribute.
- verbose='true' attribute will enable detailed logging into the build log.
- parseOutOfDate='true' attribute will process all the files matching the path. Otherwise, only those updated during the build (is determined by last modification timestamp) are processed.
- 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 which specify errors and warnings limits, exceeding which will cause the build failure.



- After importData message is received, TeamCity agent starts to monitor specified paths on 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 step finish, the agent ensures all the present reports are processed before beginning of the next step. This behavior is different from that of [XML 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 several directories or parse several types of the report, send the corresponding service messages one after another.

teamcity-info.xml

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 should contain 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 integer value.

The key here relates to the key of `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 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 valid TeamCity server username and password to authenticate using basic HTTP authentication. The user should have appropriate permissions 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 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 GET** 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&agent
```

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&agent
```

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&modif
```

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&branchName=master
```

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 build artifacts to publish 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 as described [here](#) if you are running TeamCity 8.1 or above; for versions earlier than 8.1, refer to [this section](#).

Starting from TeamCity 8.1, 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.
You can override the inherited Report tab by creating a new report tab with the same name as the inherited one in the subproject.
- **Project-level:** appears under the Project 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 page| [Report Tabs](#) page, click **Create new report tab** and proceed with the following options:

Option	Description
Report tab type	Type of the report tab (Build or Project-level)
Tab Title	Specify a unique title of the report tab that will be displayed in the web UI.
Get artifacts from	This field is available for project-specific report tabs only Specify the build whose artifacts will be shown on the tab. Select whether the report should be taken from last successful, pinned, finished build or build with specified build number or last build with a specified tag.
Start page	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. To use a file from an archive, use the <code>path-to-archive!relative-path</code> syntax, e.g. <code>javadoc.zip!index.html</code> . See the list of supported archives .

Prior to TeamCity 8.1, the behavior is slightly different:

There are two types of report tabs available:

- **Build-level:** appears on the [build results](#) page for each build that produced the artifact with the specified name. These are configured server-wide on [Administration | Integrations | Report Tabs](#) page.
- **Project-level:** appears under the Project page for a particular project only if a build within the project produces the specified reports artifact. These are configured on [Project administration > Report Tabs](#) page.

To configure a report tab, go to the [Integrations | Report Tabs](#) page (for the build-level tab) or project settings (for a project-level tab), click **Create new report tab** and proceed with the following options:

Option	Description
Tab Title	Specify a unique title of the report tab that will be displayed in the web UI.
Get artifacts from	This field is available for project-specific report tabs only Specify the build whose artifacts will be shown on the tab. Select whether the report should be taken from last successful, pinned, finished build or build with specified build number or last build with a specified tag.
Start page	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. To use a file from an archive, use the <code>path-to-archive!relative-path</code> syntax, e.g. <code>javadoc.zip!index.html</code> . See the list of supported archives .

See also:

Custom Chart

In addition to statistic charts generated automatically by TeamCity, 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.

On this page:

- Displaying Custom Chart in TeamCity Web UI
 - Tags Reference
 - Chart Dimensions
 - Chart Axis Settings
 - Default Statistics Values Provided by TeamCity
 - Custom Build Metrics

Displaying Custom Chart in TeamCity Web UI

To make TeamCity display a custom chart in the web UI, you need to update the dedicated configuration file:

- For Project-level chart: `<TeamCity Data Directory>/config/projects/<ProjectID>/pluginData/plugin-settings.xml`
- For Build Configuration-level chart: `<TeamCity Data Directory>/config/main-config.xml`

You can edit these files while the server is running, they will be automatically reloaded.

A statistics chart is added using the `graph` tag. See the examples below:

Custom project-level charts in `plugin-settings.xml`

```

<settings>
  <custom-graphs> <!-- This tag is required only in plugin-settings.xml -->
    <graph title="Duration comparison" hideFilters="showFailed" seriesTitle="Some key"
format="duration">
      <valueType key="BuildDuration" title="duration1" buildTypeId="my_first_configuration_id"/>
      <valueType key="BuildDuration" title="duration2" buildTypeId="my_second_configuration_id"/>
      <valueType key="customKey" title="Custom data" color="#ee0055" /> <!-- Will use data from build
configuration my_second_configuration_id -->
    </graph>
  </custom-graphs>
</settings>

```

Custom build configuration-level charts in `main-config.xml`

```

<server ...>
  <!-- Some other stuff -->
  <graph title="Passed Test Count" seriesTitle="Configuration">
    <valueType key="PassedTestCount" title="This configuration" />
    <valueType key="PassedTestCount" title="Passed Test Count" buildTypeId="bt32" /> <!-- This is
explicit reference to build configuration -->
  </graph>
  <graph title="Tests against Coverage">
    <valueType key="PassedTestCount" title="Tests" color="#00ff00" />
    <valueType key="CodeCoverageL" title="Line coverage" color="#ff0000" />
  </graph>
  <graph title="Custom data" seriesTitle="Metric name" format="size">
    <valueType key="key1" title="Metric 1" />
    <valueType key="key2" title="Metric 1" />
    <valueType key="BuildDuration" title="Duration" />
  </graph>
</server>

```

Note that when adding custom charts on the project level, the intermediate `custom-graphs` tag is required.

Tags Reference

<graph> : describes a single chart. It should contain one or more `valueType` subtags, which describe series of data shown in the chart.

Attribute	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"> • <code>showFailed</code> — include results from failed builds by default. • <code>averaged</code> — by default, show averaged values on the chart.
<code>hideFilters</code>	The list of comma-separated filter names that will not be shown next to the chart: <ul style="list-style-type: none"> • <code>all</code> — hide all filters. • <code>series</code> — hide series filter (you won't be able to show only data from specific <code>valueType</code> specified for the chart.) • <code>range</code> — hide the date range filter. • <code>showFailed</code> — hide the checkbox which allows to include data for failed builds. • <code>averaged</code> — hide the checkbox which allows to view averaged values. • <code>Defaults</code> — empty (all filters are shown).
<code>format</code>	The format of the y-axis values. Supported formats are: <ul style="list-style-type: none"> • <code>duration</code>, data should be in milliseconds; • <code>percent</code>, data should be in percents (from 0 to 100); • <code>percentby1</code>, the format will show data between 0 and 1 as percents (from 0 to 100); • <code>size</code>, data should be in bytes. If no format is specified, the numeric format is used.

<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.

Attribute	Description
key	A name of the valueType (or series). It can be predefined by TeamCity, like <code>BuildDuration</code> or <code>ArtifactsSize</code> (see below Default Statistics Values Provided by TeamCity 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 <key>.
buildTypeId	This field allows you to explicitly specify a build configuration to use the data from for the given valueType. 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"> if the <code>buildTypeId</code> is set within the <code>valueType</code>, the data is taken from this build configuration even if it belongs to a different project. if the <code>buildTypeId</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. if the <code>buildTypeId</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. Such charts can be configured only in <code>main-config.xml</code>.
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 HTML Colors reference and HTML Color Names reference . If not specified, an automatic color will be assigned based on the series title.

Chart Dimensions

You can set the custom chart width/height in pixels using the `width` and `height` properties within the XML `properties` tag:

```
<graph ...>
  <properties>
    <property name="width" value="300"/>
    <property name="height" value="100"/>
  </properties>
</graph>
```

Chart Axis Settings

You can also customize the default axis settings for a chart via properties added withing the XML `properties` tag:

```
<graph title="Test count" seriesTitle="Test group">
  <properties>
    <property name="axis.y.type" value="logarithmic"/>
    <property name="axis.y.includeZero" value="false"/>
    <property name="axis.y.max" value="10000"/>
  </properties>
  <valueType key="FailedTestCount" title="Failed" color="red"/>
  <valueType key="IgnoredTestCount" title="Ignored" color="grey"/>
  <valueType key="PassedTestCount" title="Passed" color="green"/>
</graph>
```

Supported properties:

Name	Description
<code>axis.y.type</code>	Logarithmic for the logarithmic Y axis scale, linear for the standard scale. The default is linear .
<code>axis.y.includeZero</code>	Whether the zero value is included on the Y axis (true) or not (false). The default is true .
<code>axis.y.min</code>	An integer value to start the Y axis from.
<code>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.

The **Build Results|Parameters|Reported statistic values** page shows a statistics chart next to each of the values reported by the build. Click the *View Trend* icon  to see the chart.

Key	Description	Unit
ArtifactsSize	The sum of all artifact file sizes in the artifact directory	Bytes
BuildArtifactsPublishingTime	The duration of the artifact publishing step in the build	Milliseconds
BuildCheckoutTime	The duration of the source checkout step	Milliseconds
BuildDuration	The build duration (all build stages)	Milliseconds
CodeCoverageB	Block-level code coverage	%
CodeCoverageC	Class-level code coverage	%
CodeCoverageL	Line-level code coverage	%
CodeCoverageM	Method-level code coverage	%
CodeCoverageAbsLCovered	The number of covered lines	int
CodeCoverageAbsMCovered	The number of covered methods	int
CodeCoverageAbsCCovered	The number of covered classes	int
CodeCoverageAbsLTotal	The total number of lines	int
CodeCoverageAbsMTotal	The total number of methods	int
CodeCoverageAbsCTotal	The total number of classes	int
DuplicatorStats	The number of code duplicates found	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
PassedTestCount	The number of successfully passed tests 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 using the [teamcity-info.xml](#) file.

Note that custom value keys should be unique and should not interfere with value keys predefined by TeamCity.

Since TeamCity 8.1, charts are displayed for configured custom statistic values on the Build Parameters tab without any additional configuration.

See also:

Concepts: [Code Coverage](#) | [Code Inspection](#) | [Code Duplicates](#)
User's Guide: [Statistic Charts](#)
Extending TeamCity: [Build Script Interaction with TeamCity](#)

Developing TeamCity Plugins

TeamCity functionality can be significantly extended by a custom plugin. TeamCity plugins are written in Java (Kotlin, Groovy and JRuby can also be used), runs within the TeamCity application and has 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](#)
- open-source plugins: [bundled](#) or [third-party](#)

If you cannot find enough information or have a question regarding API please do not hesitate to post your question into [TeamCity Plugins forum](#). Please use search before posting to find out if alike question was already answered in the forums.

Please refer to corresponding section for further details.

- [Plugin Types in TeamCity](#)
- [Bundled Development Package](#)
- [Developing Plugins Using Maven](#)
- [Open API Changes](#)
- [Plugins Packaging](#)
- [Server-side Object Model](#)
- [Agent-side Object Model](#)
- [Extensions](#)
- [Web UI Extensions](#)
- [Plugin Settings](#)
- [Development Environment](#)
- [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](#)

Plugin Quick Start

For starting with a plugin using Maven please see [Developing Plugins Using Maven](#).

There are also several approaches to create plugins provided by third parties or existing out of the main TeamCity development line:

- [Gradle plugin to build TeamCity plugins](#)
- [Maven Archetype for TeamCity server plugin](#)
- [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

See also a [post](#) on the very initial steps for setting up plugin development environment.

Plugin Types in TeamCity

TeamCity mainly consists of two parts:

1. The server that gathers information while builds are running
2. Agents that run builds and send information to server

Consequently, depending on where the code runs, there are

- server-side plugins
- agent-side plugins.

Aside from 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, for example, it is possible to format stack trace on the web the way you need or modify build status text, [read more](#).

Plugins can also modify 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](#).

Bundled Development Package

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:

- `common-api.jar`
- `agent-api.jar`

Correspondingly for the server-side plugin, you need:

- `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 `build.properties` file and set value for `path.variable.teamcitydistribution` property to 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's Tomcat 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.

Developing Plugins Using Maven

Since TeamCity 8.0 you can easily develop TeamCity plugins with Maven.

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://repository.jetbrains.com/all>). Add this fragment to the <repositories> section of your pom file to access it:

```
<repository>
  <id>jetbrains-all</id>
  <url>http://repository.jetbrains.com/all</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>8.0</version>
  <scope>provided</scope>
</dependency>
```

The agent-side API:

```
<dependency>
  <groupId>org.jetbrains.teamcity</groupId>
  <artifactId>agent-api</artifactId>
  <version>8.0</version>
  <scope>provided</scope>
</dependency>
```

 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>8.0</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)

Here is the Maven command that will generate a project for a server-side-only plugin depending on 8.0 TeamCity version:

```
mvn archetype:generate -DarchetypeRepository=http://repository.jetbrains.com/all  
-DarchetypeArtifactId=teamcity-server-plugin -DarchetypeGroupId=org.jetbrains.teamcity.archetypes  
-DarchetypeVersion=8.0
```

Here is the Maven command that will generate a project that contains both, the server and agent parts of a plugin and depends on 8.0 TeamCity version:

```
mvn archetype:generate -DarchetypeRepository=http://repository.jetbrains.com/all  
-DarchetypeArtifactId=teamcity-plugin -DarchetypeGroupId=org.jetbrains.teamcity.archetypes  
-DarchetypeVersion=8.0
```

Here is the Maven command that will generate a sample project on 8.0 TeamCity version:

```
mvn archetype:generate -DarchetypeRepository=http://repository.jetbrains.com/all  
-DarchetypeArtifactId=teamcity-sample-plugin -DarchetypeGroupId=org.jetbrains.teamcity.archetypes  
-DarchetypeVersion=8.0
```

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, ready to be installed to TeamCity.

Open API Changes

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.agentBuildContext`. 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.Notificator.notifyResponsibleAssigned(SBuildType, Set<SUser>),`
`j.b.notification.Notificator.notifyResponsibleChanged(TestNameResponsibilityEntry,`
`TestNameResponsibilityEntry, SProject, Set<SUser>),`
`j.b.notification.Notificator.notifyResponsibleAssigned(TestNameResponsibilityEntry,`
`TestNameResponsibilityEntry, SProject, Set<SUser>)`

Changes prior to 4.5.5

Not documented

Plugins Packaging

This page is intended for plugin developers and explains how to package TeamCity plugins. See [Installing Additional Plugins](#) for instructions on installing plugins.

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](#)
- [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`.



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 `teamcity-plugin.xml` file, see the [section 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 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 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:schemas-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>
    <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>
```

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

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

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#GetConfiguration\(\)](#) 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

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 to customize 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 to apply custom formatting to test stacktrace to be shown in the UI.
jetbrains.buildServer.serverSide.buildDistribution.StartBuildPrecondition	4.5	Allows to define preconditions for build starting on an agent, that is, you can instruct TeamCity to delay build till some condition is met.

<code>jetbrains.buildServer.serverSide.GeneralDataCleaner</code>	2.0	This extension is called when cleanup process is going to finish, plugins can clean their data with help of this extension.
<code>jetbrains.buildServer.serverSide.DataCleaner</code>	2.0	This extension is called when 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.
<code>jetbrains.buildServer.serverSide.ParametersPreprocessor</code>	3.0	Allows to modify build parameters right before they are sent to an agent.
<code>jetbrains.buildServer.serverSide.parameters.BuildParametersProvider</code>	5.0	Allows to add additional parameters available for a build. It differs from <code>ParametersPreprocessor</code> in a way that parameters added by <code>BuildParametersProvider</code> will be available in popup showing available parameters, and will be considered when requirements are calculated.
<code>jetbrains.buildServer.serverSide.parameters.ParameterDescriptionProvider</code>	5.0	Provides a human readable description for a parameter, see also <code>BuildParametersProvider</code> .
<code>jetbrains.buildServer.messages.serviceMessages.ServiceMessageTranslator</code>	4.0	Translator for specific type of service messages.
<code>jetbrains.buildServer.usageStatistics.UsageStatisticsProvider</code>	6.0	Provides a custom usage statistics.

See also:

[Extending TeamCity: Developing TeamCity Plugins | Web UI Extensions](#)

Web UI Extensions

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
jetbrains.buildServer.web.openapi.PlaceId	A list of page place identifiers / extension points
jetbrains.buildServer.web.openapi.PagePlace	A single page place associated with PlaceId , allows to add / remove extensions
jetbrains.buildServer.web.openapi.PageExtension	Page extension interface
jetbrains.buildServer.web.openapi.SimplePageExtension	Base class for page extensions
jetbrains.buildServer.web.openapi.CustomTab	Custom tab extension interface
jetbrains.buildServer.web.openapi.PagePlaces	Maintains a collection of page places and allows to locate PagePlace by PlaceId
jetbrains.buildServer.web.openapi.WebControllerManager	Maintains a collection of custom controllers, allows to register custom controllers
jetbrains.buildServer.controllers BaseController	Base class for controllers

Plugin Settings

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

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](#)

Typical Plugins

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

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.

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.

Logging to 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.ant.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.

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.

Risk Tests Reordering in Custom Test Runner

In TeamCity, you can instruct the system to [run risk group tests before any others](#).

To implement risk group tests reordering feature for your own custom test runner, TeamCity provides following special properties:

- **teamcity.tests.runRiskGroupTestsFirst** — this property value contains groups of tests to run before others. Accordingly there are two groups: **recentlyFailed** and **newAndModified**. If more than one group specified they are separated with comma. This property is provided only if corresponding settings are selected on build runner page.
- **teamcity.tests.recentlyFailedTests.file** — this property value contains full path to a file with recently failed tests. The property is provided only if **recentlyFailed** group is selected. The file contains tests separated by new line character. For Java like tests full class names are stored in the file (without test method name). In other cases full name of the test will be stored in the file as it was reported by the tests runner.
- **teamcity.build.changedFiles.file** — this property is useful if you want to support running of new and modified tests in your tests runner. This property contains full path to a file with 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 lines separated by new line. Each line corresponds to one file and has the following format:

```
<relative file path>:<change type>:<revision>
```

where:

- **<relative file path>** is a 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 repository. If file is a part of change list started via remote run then string **<personal>** will be written instead of file revision.

- **teamcity.build.checkoutDir** — this property contains path to a build checkout directory. It is useful if you need to convert relative paths to modified files to absolute ones.



TeamCity will pass **teamcity.tests.runRiskGroupTestsFirst**, **teamcity.tests.recentlyFailedTests.file** and **teamcity.build.changedFiles.file** properties to the build process, but if process starts additional JVM or other processes, these properties won't be passed to them automatically.

For example, if you are using Ant runner, you will have access to these properties from the Ant build.xml. But if your build.xml starts 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.

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 `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.IssueProvider` - 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.IIssueProvider` and `jetbrains.buildServer.issueTracker.IIssueProviderFactory`)
- 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/youttrack/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 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

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:

- Git plug-in
- Mercurial plug-in

For more information on TeamCity plugins, please refer to [TeamCity Plugins section](#).

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](#).

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.VcsSupportCore#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:



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 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.

Server-Side Caching

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 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, 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`, the 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.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:

```

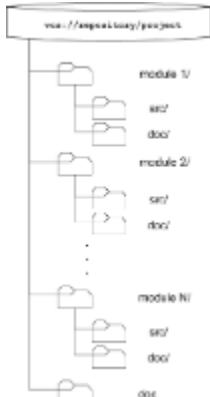
public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion, String
currentVersion, CheckoutRules checkoutRules)
throws VcsException {
final MyConnection conn = obtainConnection(root); // get a connection to the repository
return VcsSupportUtil.collectBuildChanges(root, fromVersion, 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:



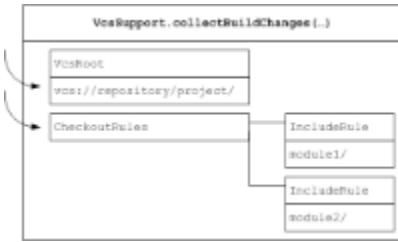
We want to monitor changes only in `module1` and `module2`. Therefore we've configured the following checkout rules:

```

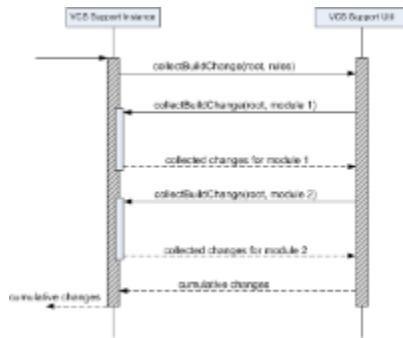
\+:module1
\+:module2

```

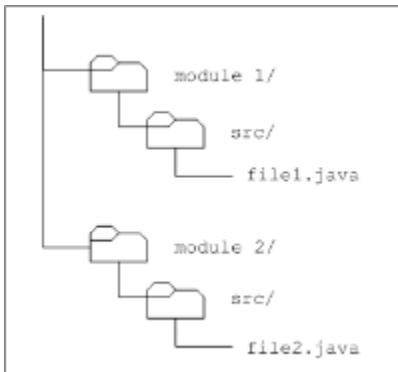
When `collectBuildChanges(...)` is invoked it will receive a `VcsRoot` instance that corresponds to `vcs://repository/project/ ||` and a `\{CheckoutRules` instance with two `IncludeRules` — one for "module1" and the other for "module2".



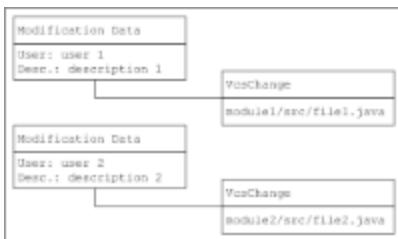
If `collectBuildChanges(...)` utilizes `VcsSupportUtil.collectBuildChanges(...)` it transforms the invocation into two separate calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. If you have implemented `CollectChangesByIncludeRule` in the way described in the listing above you will have 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.

Registering In TeamCity

During the server startup all VCS plugins are required to register themselves in the VCS Manager ([jetbrains.buildServer.vcs.VcsManager](#)). 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 to be used in "auth-config.xml"
        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 to be used in "auth-config.xml"
        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.Notifier` interface and register implementation in the `jetbrains.buildServer.notification.NotifierRegistry`.

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 [Placeld.NOTIFIER_SETTINGS_FRAGMENT](#) and [Placeld.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>
- <http://code.google.com/p/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 [Custom Chart](#).

Quick Start

- TODO: how to use a plugin sample project, how to package and deploy a plugin (links?)
- An easy way to add custom statistics is to just insert a jsp fragment into WebPlace using a helper bean:

```
<bean id="myLogoFragment" class="jetbrains.buildServer.web.openapi.SimpleWebExtension"
init-method="register">
    <property name="name" value="myLogoFragment"></property>
    <property name="place" value="BUILD_CONF_STATISTICS_FRAGMENT"></property>
    <property name="jspPath" value="/myLogoFragment.jsp"/>
</bean>
```

- To insert statistics chart into a jsp page:

```
<%@taglib prefix="stats" tagdir="/WEB-INF/tags/chart"%>
<stats:buildChart id="g1" valueType="BuildDuration"/>
```

- To add a custom build metric, extend `BuildValueTypeBase` to just define your build metric calculation method, appearance, and key. After that you can reference this metric by its key in statistics chart/report tags.

More Details

BuildType Statistics tab extension point

```
//WebControllerManager webControllerManager
webControllerManager.addPageExtension(WebPlace.BUILD_INFO_FRAGMENT, this);
```

Customizing chart appearance

- width, height — chart image size
- hideFilters — suppress filter controls

Adding custom metrics

1. Implement `jetbrains.buildServer.serverSide.statistics.ValueType`, extend `BuildFinishAwareValueTypeBase` or `CompositeVTB` for convenience
2. Register it using `jetbrains.buildServer.serverSide.statistics.ValueProviderRegistry.registerValueProvider`

Custom build metrics details

1. Implement `jetbrains.buildServer.serverSide.statistics.build.BuildFinishAware` in your `ValueType` to be notified of build finished event.
2. Calculate your metric.
3. Employ `jetbrains.buildServer.serverSide.statistics.build.BuildDataStorage.publishValue` to publish your value.
4. Employ `jetbrains.buildServer.serverSide.statistics.build.BuildDataStorage.getDataSet` to retrieve selected data.

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 `Placeld.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\uzzz

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!

REST API

On this page:

- General information
 - REST Authentication
 - Superuser access
 - REST API Versions
 - URL Structure
 - Locator
 - Examples
 - Supported HTTP Methods
 - Response Formats
 - Logging
 - CORS Support
 - TeamCity Data Entities Requests
 - Projects and Build Configuration/Templates Lists
 - Build Configuration Locator
 - Project Settings
 - VCS Roots
 - Build Configuration And Template Settings
 - Build Requests
 - Build Locator
 - Queued Builds
 - Triggering a Build
 - Build node examples
 - Build Tags
 - Build Pinning
 - Build Canceling/Stopping
 - Artifacts
 - Authentication
 - Other Build Requests
 - Build fields
 - Statistics
 - Tests
 - Investigations
 - Agents
 - Agent Pools
 - Assigning Projects to Agent Pools
- Users
- Other
 - Data Backup

- Typed Parameters Specification
- Build Status Icon
- CCTray
- Request Examples
 - Request Sending Tool
 - Creating a new project
 - Making user a system administrator

General information

REST API is a [plugin](#) bundled **since TeamCity 5.0.**

To use a 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.

If your server web UI is accessible via the <http://teamcity:8111/> URL, use <http://teamcity:8111/httpAuth/app/rest/application.wadl> \ to get the full list of supported requests and names of parameters (for basic HTTP authentication).

REST Authentication

You can authenticate yourself for the REST API in the following ways:

- Using basic HTTP authentication. Provide a valid TeamCity username and password with the request and include "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): <http://teamcity:8111/guestAuth/app/rest/builds>

There is also a [workaround](#) for not sending credentials with every request.

If you perform a request from within a TeamCity build, consider using teamcity.auth.userId/teamcity.auth.password system properties as credentials (within TeamCity settings you can reference them like %system.teamcity.auth.userId% and %system.teamcity.auth.password%). Server URI is available as %teamcity.serverUrl% within a build.

Superuser access

If you add the `rest.use.authToken=true` internal property, any user can perform superuser operation if the authToken is passed in the URL parameter. The authToken will be logged into logs/teamcity-rest.log log. You will still need to supply valid user credentials to use this approach.

REST API Versions

Under the <http://teamcity:8111/app/rest/> URL the latest version is available.

Under the <http://teamcity:8111/app/rest/<version>> URL, earlier versions CAN be available. Our general policy is to supply TeamCity with at least ONE previous version.

In TeamCity 8.0.x you can use "6.0" or "7.0" instead of <version> to get [earlier versions](#) of the protocol.

URL Structure

The general structure of the URL in the TeamCity API is `teamcityserver:port/<authType>/app/rest/<entity>`, where

- `teamcityserver` and `port` define the server name and the port used by TeamCity
- `<authType>` is the [authentication type](#) to be used
- `app` means that the request will be directed to the TeamCity application
- `rest` means REST API
- `<entity>` identifies the required entity. Requests that respond with lists (e.g. .../projects, .../buildTypes, .../builds, .../changes) serve partial items with only the most important item fields and list `href` }s of the items within the list. To get the full item data, use the URL constructed with the value of the {{ "href" }} item attribute.

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 symbols ",)"
- dimension, allowing to filter entities using multiple criteria: `<dimension1>:<value1>,<dimension2>:<value2>`

Refer to each entity description for the supported locators.

Note: If the value is to contain the "," symbol, it should be enclosed into parentheses: "<value>".

Examples

`http://teamcity:8111/httpAuth/app/rest/projects` gets you the list of projects
`http://teamcity:8111/httpAuth/app/rest/projects/id:RESTAPIPlugin`(example id is used) gets you the full data for the REST API Plugin project.
`http://teamcity:8111/httpAuth/app/rest/buildTypes/id:bt284/builds?status=SUCCESS&tag=EAP` - (example ids are used) to get builds
`http://teamcity:8111/httpAuth/app/rest/builds/?locator=<buildLocator>` - to get builds by build locator.
`http://teamcity:8111/httpAuth/app/rest/changes?buildType=id:bt133&sinceChange=id:24234` - to get all the changes in the build configuration since the change identified by the id.

Supported HTTP Methods

- GET: retrieves the requested data
- 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.
- PUT: based on the existence of the entity, creates or updates the entity in the request
- DELETE: removes the requested data

Response Formats

The TeamCity REST APIs returns HTTP responses in the following formats:

Format	Response Type	Requested via
plain text	single-value responses	text/plain in the HTTP Accept header
xml	complex value responses	application/xml in the HTTP Accept header
json	complex value responses	application/json in the HTTP Accept header

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 Log4J "jetbrains.buildServer.server.rest" category).

CORS Support

TeamCity REST can be configured to allow [cross-origin requests](#).

If you want to allow requests from a page loaded from a specific domain, add the domain to comma-separated `internal` property `rest.cors.origins`.

e.g.

```
| rest.cors.origins=http://myinternalwebpage.org.com:8080,https://myinternalwebpage.org.com
```

If that does not work, enable [debug logging](#) and investigate the log.

TeamCity Data Entities Requests

Projects and Build Configuration/Templates Lists

List of projects: GET `http://teamcity:8111/httpAuth/app/rest/projects`

Project details: GET `http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>`
`<projectLocator>` can be `id:<internal_project_id>` or `name:<project%20name>`

List of Build Configurations: GET `http://teamcity:8111/httpAuth/app/rest/buildTypes`

List of Build Configurations of a project: GET

`http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>/buildTypes`

List of templates for a particular project: `http://teamcity:8111/httpAuth/app/projects/<projectLocator>/templates`.

List of all the templates on the server(**Since TeamCity 8.1**):

<http://teamcity:8111/httpAuth/app/rest/buildTypes?locator=templateFlag:true>

Build Configuration/Template details: GET <http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>>

Build Configuration Locator

Most frequently used values for "<buildTypeLocator>" are id:<buildConfigurationOrTemplate_id> and name:<Build%20Configuration%20name>.

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

Project Settings

Get project details: GET <http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>>

Delete a project: DELETE <http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>>

Create a new empty project: POST plain text (name) to <http://teamcity:8111/httpAuth/app/rest/projects/>

Since TeamCity 8.0: 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/httpAuth/app/rest/projects>. Also see an example.

Edit project parameters: GET/DELETE/PUT

http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>/parameters/<parameter_name> (accepts/produces text/plain)

Project name/description/archived status: GET/PUT

http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>/<field_name> (accepts/produces text/plain) where <field_name> is one of "name", "description", "archived".

Since TeamCity 8.0

Project's parent project: GET/PUT XML <http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>/parentProject>

VCS Roots

List all VCS roots: GET <http://teamcity:8111/httpAuth/app/rest/vcs-roots>

Get details of a VCS root/delete a VCS root: GET/DELETE

<http://teamcity:8111/httpAuth/app/rest/vcs-roots/<vcsRootLocator>>

Where <vcsRootLocator> is "id:<internal VCS root id>"

Create a new VCS root: POST VCS root XML (the one like retrieved for a GET request for VCS root details) to

<http://teamcity:8111/httpAuth/app/rest/vcs-roots>

Also supported:

GET/PUT http://teamcity:8111/httpAuth/app/rest/vcs-roots/<vcsRootLocator>/properties/<property_name>

GET/PUT http://teamcity:8111/httpAuth/app/rest/vcs-roots/<vcsRootLocator>/<field_name> where <field_name> is one of the following: name, shared, project (post project locator to "project" to associate a VCS root with a specific project). **Before TeamCity 8.0** project used to be a "projectId".

Build Configuration And Template Settings

Get build configuration details: GET <http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>> (check details about <buildTypeLocator>)

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.

Get/set paused build configuration state: GET/PUT

<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/paused> (put "true" or "false" text as text/plain)

Build configuration settings: GET/DELETE/PUT

http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/settings/<setting_name>

Build configuration parameters: GET/DELETE/PUT

http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/parameters/<parameter_name> (accepts/produces text/plain)

Build configuration steps: GET/DELETE

http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/steps/<step_id>
Create build configuration step: POST <http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/steps> The XML posted is the same as retrieved by GET request to .../steps/<step_id>

Features, triggers, agent requirements, artifact and snapshot dependencies follow the same pattern as steps with URLs like:

<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/features/<id>>
<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/triggers/<id>>
<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/agent-requirements/<id>>
<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/artifact-dependencies/<id>>
<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/snapshot-dependencies/<id>>

Build configuration VCS roots: GET/DELETE

<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/vcs-root-entries/<id>>

Attach VCS root to a build configuration: POST

<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/vcs-root-entries> The XML posted is the same as retrieved by GET request to

<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/vcs-root-entries/<id>>

Create a new empty build configuration: POST plain text (name) to

<http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>/buildTypes>

Copy a build configuration: POST XML <newBuildTypeDescription name='Conf Name' sourceBuildTypeLocator='id:bt42' copyAllAssociatedSettings='true' shareVCSRoots='false' /> to

<http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>/buildTypes>

Read, detach and attach a build configuration from/to a template: GET/DELETE/PUT

<http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/template> (PUT accepts template locator with "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/buildNumberCounter --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/buildNumberPattern --data  
<new format> --header "Content-Type: text/plain"
```

Build Requests

List builds: GET <http://teamcity:8111/httpAuth/app/rest/builds/?locator=<buildLocator>>

Get details of a specific build: GET <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>> (also supports DELETE to delete a build)

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.

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:

`buildType:<buildTypeLocator>` - only the builds of the specified build configuration

`tags:<tags>` - ","(comma) - a delimited list of build tags (only builds containing all the specified tags are returned)

`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 a personal flag.

`canceled:<true/false/any>` - limit builds by a canceled flag.

`running:<true/false/any>` - limit builds by a running flag.

`pinned:<true/false/any>` - limit builds by a 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>)". If not specified, only builds from the **default**

branch are returned.

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 in the same format as dates returned by REST API (e.g. "20130305T170030+0400").

project:<project locator> - limit the list to the builds of the specified project (belonging to any build type directly or indirectly under the project)

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. If none of the latest N builds match the other specified criteria of the build locator, 404 response is returned.

Note: If build configuration utilizes feature branches then, by default, only builds from default branch are returned. To retrieve all builds, add the following locator: branch:default:any. The whole path will look like this: /httpAuth/app/rest/builds/?locator=buildType:One_Git,branch:default:any

Queued Builds

The features listed in the section are available **since TeamCity 8.1**:

GET <http://teamcity:8111/httpAuth/app/rest/buildQueue>

Supported locators:

- project:<locator>
- buildType:<locator>

Get details of a queued build:

GET <http://teamcity:8111/httpAuth/app/rest/buildQueue/taskId: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/httpAuth/app/rest/buildQueue/taskId:XXX/compatibleAgents>

Examples:

List queued builds per project:

GET <http://teamcity:8111/httpAuth/app/rest/buildQueue?locator=project:<locator>>

List queued builds per build configuration:

GET <http://teamcity:8111/httpAuth/app/rest/buildQueue?locator=buildType:<locator>>

Triggering a Build

This is only available **since TeamCity 8.1**. For earlier versions see [Accessing Server by HTTP#Triggering a Build From Script](#).

To start a build, send POST request to <http://teamcity:8111/httpAuth/app/rest/buildQueue> with the "build" node 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.

Build node examples

Basic build for a build configuration:

```
<build>
    <buildType id="buildConfigID"/>
</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>

```

Build on a specified change, forced rebuild of all dependencies and clean sources before the build, moved to the build queue top on triggering.
(Please note that the change is set via the change's internal modification id, not revision. The id can be seen in the change node listed by the REST API or in the URL of the change detail UI page):

```

<build>
  <triggeringOptions cleanSources="true" rebuildAllDependencies="true" queueAtTop="true" />
  <buildType id="buildConfID"/>
  <lastChanges>
    <change id="modificationId"/>
  </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/httpAuth/app/rest/builds/<buildLocator>/tags/>
 Replace tags: PUT <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/tags/> (put the same XML of JSON as returned by GET)
 Add tags: POST <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/tags/> (post the same XML of 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/httpAuth/app/rest/builds/<buildLocator>/pin/> (returns "true" or "false" text)
 Pin: PUT <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/pin/> (the text in the request data is added as a comment for the action)
 Unpin: DELETE <http://teamcity:8111/httpAuth/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

POST the <buildCancelRequest comment='CommentText' readdlntoQueue='true' /> item to the URL of a running or queued build:

▼ Example: click to expand

```

curl -v -u user:password --request POST "http://localhost:7000/app/rest/buildQueue/<buildLocator>" --data
"<buildCancelRequest comment='readdlntoQueue='true' />" --header "Content-Type: application/xml"

```

Note: Readding of builds into the queue is supported for running builds only.

Expose cancelled build details:

See the canceledInfo element of the build item (available via GET
<http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>>)

Artifacts

Since TeamCity 8.0

GET <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/artifacts/content/<artifact relative name>>
 (returns the content of a build artifact)

Media-Type: application/octet-stream or a more specific media type (determined from artifact name)
Possible error: 400 if the specified path references a directory

GET <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/artifacts/metadata/<artifact relative name>> (returns information about a build artifact)
Media-Type: application/xml or application/json

GET <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/artifacts/children/<artifact relative name>> (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

<artifact relative name> supports referencing files under archives using "!" delimiter after the archive name.

Examples:

GET

<http://teamcity:8111/httpAuth/app/rest/builds/id:100/artifacts/children/my-great-tool-0.1.jar!-/META-INF/MANIFEST.MF>

GET

<http://teamcity:8111/httpAuth/app/rest/builds/id:100/artifacts/metadata/my-great-tool-0.1.jar!-/lib/commons-logging-1.1.3.jar>

GET

<http://teamcity:8111/httpAuth/app/rest/builds/id:100/artifacts/content/my-great-tool-0.1.jar!-/lib/commons-logging-1.1.3.jar>

Authentication

If you download the 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

Build fields

Get single build's field: GET http://teamcity:8111/httpAuth/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 build statistics: GET <http://teamcity:8111/httpAuth/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/httpAuth/app/rest/builds/<buildLocator>/statistics/<value_name>

Tests

Since TeamCity 8.1

List tests:

GET <http://localhost:8111/app/rest/testOccurrences?locator=<locator dimension>:<value>>

Supported locators:

- build
- test
- current:true

Examples:

List all build's tests: GET <http://localhost:8111/app/rest/testOccurrences?locator=build:<build locator>>

Get individual test history:

GET <http://localhost:8111/app/rest/testOccurrences?locator=test:<id>>

Supported test locators:

- "id:<internal test id>" available as a part of the URL on the test history page
- "name:<full test name>"

Investigations

Since TeamCity 8.1

List investigations in the Root project and its subprojects:

<http://localhost:8111/app/rest/investigations>

Supported locators:

- test: (id:TEST_NAME_ID)
- test: (name:FULL_TEST_NAME)
- assignee: (<user locator>)
- buildType:(id:XXXX)

Examples:

Get investigations for a specific test:

[http://localhost:8111/app/rest/investigations?locator=test:\(id:TEST_NAME_ID\)](http://localhost:8111/app/rest/investigations?locator=test:(id:TEST_NAME_ID))

[http://localhost:8111/app/rest/investigations?locator=test:\(name:FULL_TEST_NAME\)](http://localhost:8111/app/rest/investigations?locator=test:(name:FULL_TEST_NAME))

Get investigations assigned to a user: [http://localhost:8111/app/rest/investigations?locator=assignee:\(<user locator>\)](http://localhost:8111/app/rest/investigations?locator=assignee:(<user locator>))

Get investigations for a build configuration: [http://localhost:8111/app/rest/investigations?locator=buildType:\(id:XXXX\)](http://localhost:8111/app/rest/investigations?locator=buildType:(id:XXXX))

Agents

List of agents: GET <http://teamcity:8111/httpAuth/app/rest/agents>

List of connected agents: GET <http://teamcity:8111/httpAuth/app/rest/agents?includeDisconnected=false>

List of authorized agents: GET <http://teamcity:8111/httpAuth/app/rest/agents?includeUnauthorized=false>

Agent's single field: GET/PUT <http://teamcity:8111/httpAuth/app/rest/agents/<agentLocator>/<field name>>

See also an [example](#) for agent enabling/disabling

Since TeamCity 8.1

Delete a build agent:

<DELETE http://teamcity:8111/httpAuth/app/rest/agents/<agentLocator>>

Agent Pools

Since TeamCity 8.1

Get/modify/remove agent pools:

<GET/PUT/DELETE http://teamcity:8111/httpAuth/app/rest/projects/XXX/agentPools/ID>

Add an agent pool:

POST the agentPool name='PoolName' element to <http://teamcity:8111/httpAuth/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

Since TeamCity 8.1

Add a project to a pool:

POST the plain text (name) 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/httpAuth/app/rest/users>

Get specific user details: GET <http://teamcity:8111/httpAuth/app/rest/users/<userLocator>>

Create a user: POST <http://teamcity:8111/httpAuth/app/rest/users>

Update specific user: PUT <http://teamcity:8111/httpAuth/app/rest/users/<userLocator>>

For 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/httpAuth/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/httpAuth/app/rest/users/<userLocator>/<field name>>

User's single property: GET/DELETE/PUT

<http://teamcity:8111/httpAuth/app/rest/users/<userLocator>/properties/<property name>>

Other

Data Backup

Start backup: POST

<http://teamcity:8111/httpAuth/app/rest/server/backup?includeConfigs=true&includeDatabase=true&includeBuildLogs=true>
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/httpAuth/app/rest/server/backup>

Typed Parameters Specification

Since TeamCity 8.1

List typed parameters:

- for a project: <http://teamcity:8111/httpAuth/app/rest/projects/<locator>/parameters>
 - for a build configuration: <http://teamcity:8111/httpAuth/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/httpAuth/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/httpAuth/app/rest/buildTypes/<locator>/parameters/>. Secure data parameters, i.e `type=password`, are listed, but the values are not displayed.

Build Status Icon

Icon that represents build status: GET <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/statusIcon>
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.

If the returned image contains "no permission" 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

CCTray

CCTray-compatible XML is available via <http://teamcity:8111/httpAuth/app/rest/cctray/projects.xml>.

Without authentication (only build configurations available for guest user):

<http://teamcity:8111/guestAuth/app/rest/cctray/projects.xml>.

Since TeamCity 8.1, 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/httpAuth/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. Enable superuser in REST

create a file `<TeamCity Data Directory>\config\internal.properties` with the content:

```
| rest.use.authToken=true
```

(add the line if the file already exists)

2. Get authToken

restart the TeamCity server and look into `<TeamCity home>\logs\teamcity-rest.log` for a line:

```
| Authentication token for superuser generated: 'XXX-YYY-...-ZZZ'.
```

Copy this "XXX-YYY-...-ZZZ" string. The string is unique for each server restart

3. Issue the request

Get `curl` command line tool and use a command line:

```
| curl -v --request PUT  
| http://USER:PASSWORD@teamcity:8111/httpAuth/app/rest/users/username:USERNAME/roles/SYSTEM_ADMIN/g/?authToken=XXX-Y
```

where

"USER" and "PASSWORD" - the credentials of a valid TeamCity user (that you can log in with)

"teamcity:8111" - the TeamCity server URL

"USERNAME" - the username of the user to be made the system administrator

"XXX-YYY-...-ZZZ" - the authentication token retrieved earlier

How To...

In this section:

- [Integrate with an Issue Tracker](#)
- [Install Multiple Agents on the Same Machine](#)
- [Watch Several TeamCity Servers with Windows Tray Notifier](#)
- [Move TeamCity Installation to a New Machine](#)
- [Move TeamCity Agent](#)
- [Share the Build number for Builds in a Chain Build](#)
- [Change Server Port](#)
- [Make Temporary Build Files Erased between the Builds](#)
- [Retrieve Administrator Password](#)
- [Clear Build Queue if It Has Too Many Builds due to a Configuration Error](#)
- [Estimate Hardware Requirements for TeamCity](#)
- [Estimating the Number of Required Build Agents](#)
- [Setup TeamCity in Replication/Clustering Environment](#)
- [Move TeamCity Projects from One Server to Another](#)
- [Automatically create or change TeamCity build configuration settings](#)
- [Attach Cucumber Reporter to Ant Build](#)
- [Get Last Successful Build Number](#)
- [Create a Copy of TeamCity Server with All Data](#)
- [Test-drive Newer TeamCity Version before Upgrade](#)
- [Choose OS/Platform for TeamCity Server](#)
- [Set up Deployment for My Application in TeamCity](#)
- [Use an External Tool that My Build Relies on](#)
- [Integrate with Build and Reporting Tools](#)
- [TeamCity Security Notes](#)
- [Restore Just Deleted Project](#)
- [Set Up TeamCity behind a Proxifying Server](#)
- [Transfer 3 Default Agents to Another Server](#)
- [Configure Newly Installed MySQL Server
 - InnoDB database engine
 - max_connections
 - innodb_buffer_pool_size
 - innodb_file_per_table
 - log files on different disk](#)
- [Import coverage results in TeamCity](#)
- [Debug a Build on a Specific Agent](#)
- [Debug a Part of the Build \(a build step\)](#)

Integrate with an Issue Tracker

TeamCity comes with dedicated support for YouTrack, Jira and Bugzilla.

For any other tracker, you can turn any issue tracker issue ID references in change comments into links. Please see [Mapping External Links in Comments](#) for configuration instructions.

Install Multiple Agents on the Same Machine

See the [corresponding section](#) under agent installation documentation.

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 server's URL via UI.

```
JetBrains.TrayNotifier.exe /allowMultiple /server:http://myTeamCityServer
```

See also [details](#) in the issue tracker.

Move TeamCity Installation to a New Machine

If you need to move existing TeamCity installation to a new hardware or clean OS, it is recommended to follow [instructions on copying](#) the server from one machine to another and then [switch](#) from the old server to a new one. If you are sure you do not need to preserve old server, you can perform move operations instead of copying in those instructions.

You can use 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.

A usual advice is 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 new TeamCity instance to use the same data, please ensure you shutdown and disable old TeamCity instance before starting a new one.

Generally it is recommended to use a domain name to access the server (in agent configuration and when users access 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 be automatically using the new server.

However, if you need to use another server domain address, you will need:

- Switch agents to new URL (requires updating `serverUrl` property in `buildAgent.properties` on each agent).
- Upon new server startup do not forget to update [Server URL](#) on [Administration | Global Settings](#) page.
- Notify all TeamCity users to use 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 [Do not run new build if there is a suitable one](#) snapshot dependency option.

[Read more](#) about dependency properties.

Please watch/comment the issue related to sharing a build number [TW-7745](#).

Change Server Port

See [corresponding section](#) in server installation instructions.

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.

Retrieve Administrator Password

On the first start TeamCity displays Administrator Setup page. TeamCity installation should always have a user with System Administrator role.

If there is no user account with System Administrator role in the current authentication scheme, you can use http://<your_TeamCity_server>/setupAdmin.html URL to setup administrator account.

If there is an administrator account already, the page is not available and you need to remember the administrator account credentials.

The simplest way to change administrator password is to log in as [super user](#) which gives access to profile page of any user.

Prior to TeamCity 8.0 other options were applicable:

If you forgot Administrator password and use internal database, you can reset the password using the [instructions](#).

Otherwise you can use [REST API](#) to add System Administrator role to any existing user.

And here is an [instruction](#) to patch roles directly in the database provided by a user.

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.

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 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 build agent on the same machine as the TeamCity server, the recommended approach is to use a separate machine (though, it may be virtual) for each build agent. If you chose to install several agents [on the same machine](#), please consider possible CPU, disk, memory or network bottlenecks that might occur. [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 [external database](#) at the same machine with the server, consider 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, server disk system might need upgrade both by size and speed characteristics.

Database Note:

When using the server extensively, database performance starts to play greater role.

For reliability and performance reasons you should use external database.

Please see [notes](#) on choosing external database.

Database size requirements naturally vary based on the amount of data stored. An active server database usage can be estimated at several gigabytes of data. Say, 2 Gb per year.

Overview on the TeamCity hardware resources usage:

- CPU: TeamCity utilizes multiple cores of the CPU, so increasing number of cores makes sense. It is probably not necessary to dedicate more than 8 cores to TeamCity server.
- 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. (e.g. please pay attention to this 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 multicore 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 -XX:MaxPermSize=120m` JVM settings.

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 -XX:MaxPermSize=300m` x64 JVM settings.

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. 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 core7i 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 can work well with up to 200 build agents (200 concurrently running builds). If you need more agents/parallel builds, it is recommended to setup several separate TeamCity instances and distribute 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 [blog post](#).

Estimating 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.

Setup TeamCity in Replication/Clustering Environment

TeamCity does not provide specific support for replication/redundancy/high availability or clustering solutions.

However it supports active - failover (hot standby) approach: the data that 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.

When setting up TeamCity in a replication environment please note that TeamCity uses both database and file storage to save data. 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 disk and the database which TeamCity uses should remain in a consistent state and thus should be replicated together.

Only single TeamCity server instance should use database and data directory at any time.

Please also ensure that the distribution of the failover/backup server is of exactly the same version as the main server.

See also information on [switching](#) from one server to another.

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.

Move TeamCity Projects from One Server to Another

At this time there is no dedicated feature to move projects or build configuration from one TeamCity server to another. Please plan your TeamCity deployment in advance with this in mind. Related supported ability is [creating a copy](#) of existing server.

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](#).

For the time being it is **not** possible to transfer builds information (history, artifacts, etc.)

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 [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.

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 are two possibilities:

- 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. More over 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/r
<env key="TEAMCITY_RAKE_RUNNER_MODE" value="buildserver"/>
    </java>
</target>
```

Please, check `RUBYLIB` path separator. (';' for Windows, ':' for Linux, or '\${path.separator}' in ant for safety)

If you are launching Cucumber tests using Rake build language TC will add all necessary cmdline parameters and env. variables automatically.
P.S: 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](#)

Create a Copy of TeamCity Server with All Data

One of the ways to create a copy of the server is to create a [backup](#), then install a new TeamCity server of the same version that you already run, ensure you have appropriate environment configured (see notes below), ensure that the server uses own [TeamCity Data Directory](#) and own [database](#) and then [restore the backup](#).

This way the new server won't get build artifacts and some other less important data. If you need them, you will need to copy appropriate directories (e.g. "artifacts") from [.BuildServer/system](#) from the original to the copied server.

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 TeamCity binaries (TeamCity home directory) into the new place (`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 existing version up and running.
4. transfer relevant environment if it was specially modified for existing TeamCity installation. This might include:
 - if you run TeamCity with OS startup (e.g. Windows service), make sure all the same configuration is performed on the new machine
 - use the same [TeamCity process launching options](#), specifically check/copy environment variables starting with `TEAMCITY_`
 - use appropriate OS user account for running TeamCity server process with appropriately configured settings, global and file system permissions
 - transfer OS security settings if required
 - ensure any files/settings that were configured in TeamCity web UI are accessible; put necessary libraries/files inside TeamCity installation if they were put there earlier)
5. copy [TeamCity Data Directory](#). If you do not need the full copy, refer to the items below for optional items.
 - `.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/messages` (optional) if you want build logs (including tests failure details) preserved on the new server
 - `.BuildServer/system/artifacts` (optional) if you want build artifacts 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 and build triggers
 - `.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).
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 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)

Note: 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.

1. ensure the new server is configured to use another data directory and the database then the original server
At this point you should be ready to run the copy TeamCity server.
2. run new TeamCity server
3. upon new server startup do not forget to update [Server URL](#) on [Administration | Global Settings](#) page. You will also probably need to disable Email and Jabber notifiers or change their settings to prevent new server from sending out notifications
4. if you need the services on the copied server check that email, jabber and VCS servers are accessible from the new installation.
5. install new agents (or select some from the existing ones) and configure them to connect to the new server (using new server URL)

See also the notes on [moving the server](#) from one machine to another.

Licensing issues

You cannot use a single TeamCity license on two running servers at the same time, so to run a copy of TeamCity server you will need another license. Copies of the server created for redundancy/backup purposes can use the same licenses as they only should be running one at a time. You can get 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](#).

If you plan to run the second server at the same time as the main one regularly, you need to purchase separate licenses for the second server.

Test-drive Newer TeamCity Version before Upgrade

It's advised to try new TeamCity version before upgrading your production server. 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.

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 (e.g. integration with Microsoft TFS and VSS will work only under MS Windows)

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.

Set up Deployment for My Application in TeamCity

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) You can also 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 if you need the deployment to be triggered automatically (e.g. to deploy last successful of 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.
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

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:

- Check in the tool into the version control and use relative paths.
- 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.
- Install and register the tool in TeamCity:
 1. Install the tool on all the agents that will run the build.
 2. Add `env.` or `system.` property into `buildAgent.properties` file (or add environment variable to the system).
 3. Add agent requirement for the property in the build configuration.
 4. Use the property in the build script.
- Add environment preparation stage into the build script to get the tool from elsewhere.

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.

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, 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.

TeamCity Security Notes

These notes are provided only for your reference 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 not vulnerable to different types of attacks.

However, the general assumption and recommended setup is to deploy TeamCity in a trusted environment with no possibility to be accessed by malicious users.

Here are some notes on different security-related aspects:

- man-in-the-middle concerns
 - between TeamCity server and user's web browser: It is advised to [use HTTPS](#) for the TeamCity server. During login, TeamCity transmits user login password in an encrypted form with moderate encryption level.
 - between TeamCity agent and TeamCity server: see [the section](#).
 - between TeamCity server and other external servers (version control, issue tracker, etc.): the general rules apply as for a client (TeamCity server in the case) connecting to the external server, see guidelines for the server in question.
- user that has access to TeamCity web UI: the specific information accessible to the user is defined via TeamCity [user roles](#).
- users who can change code that is used in the builds run by TeamCity: the users have the same permissions as the system user under which TeamCity agent is running. Can access and change source code of other projects built on the same agent, modify TeamCity agent code, etc. It is advised to run TeamCity agents under users with only [necessary set of permissions](#) and use [agent pools](#) feature to insure that projects requiring different set of access are not built on the same agents.
- users with System Administrator TeamCity role: It is assumed that the users also have access to the computer on which TeamCity server is running under the user account used to run the server process. Thus, some operations like server file system browsing can be accessible by the users.
- 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 scrambled under [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.
- TeamCity agent computer administrators: same as "users who can change code that is used in the builds run by TeamCity".
- Other:
 - TeamCity web application vulnerabilities: TeamCity development team makes 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 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.

Restore Just Deleted Project

TeamCity moves settings files of deleted projects under [TeamCity Data Directory/config/_trash](#) directory.

To restore project you should find its directory on the server and move it one level up. Also you should remove suffix _projectN from the directory name.

You can do this while server is running, it should pick up restored project automatically.

Please note that TeamCity preserves builds history and other data for deleted projects/build configurations for 24 hours since the deletion time. The data is removed during the next cleanup after 24 hours timeout elapses.

Set Up TeamCity behind a Proxying Server

An internal TeamCity server should work under the same context as it is visible from outside by an external address.

Provided:

TeamCity server is installed at URL: <http://teamcity.local:8111>
It is visible to the outside world as URL: <http://teamcity.public:400>

Then use:

for Apache

```
| ProxyPass /tc http://teamcity.local:8111/tc  
| ProxyPassReverse /tc http://teamcity.local:8111/tc
```

for Nginx

```
server {  
    listen      400;  
    server_name teamcity.public;  
  
    location /tc {  
        proxy_pass          http://teamcity.local:8111/tc;  
        proxy_set_header   X-Forwarded-For $remote_addr;  
        proxy_set_header   Host $server_name:$server_port;  
    }  
}
```

The settings above ensure requests to <http://teamcity.public:400> are redirected to <http://teamcity.local:8111> and the redirect URLs sent back to the clients are correctly mapped by the proxy server.

Some other Nginx settings must be changed as well:

```
http {  
    proxy_read_timeout     1200;  
    proxy_connect_timeout  240;  
    client_max_body_size  0;  
    ...  
}
```

Where `client_max_body_size` controls the maximum size of an HTTP request. It is set to 0 to allow uploading big artifacts to TeamCity.

The TeamCity server must know the original remote address of the client. This is especially important for agents, because the server tries to establish connection with an agent to check whether the agent is behind a firewall or not. For this you need to add the following into the Tomcat main `<Host>` node of the `conf\server.xml` file (see also [doc](#)):

```
<Valve  
    className="org.apache.catalina.valves.RemoteIpValve"  
    remoteIpHeader="x-forwarded-for"  
    protocolHeader="x-forwarded-proto"  
    internalProxies="192\.168\.0\.1"  
/>
```

Where `internalProxies` must be replaced with the IP address of nginx or Apache proxy server.

If you need to use different protocols for the public and local address (e.g. make TeamCity visible to the outside world as <https://teamcity.public:400>) or your proxy server does not support redirect URL rewriting, please use the following approach:

Setup a proxying server to redirect all requests to `teamcity.public:400` to a dedicated port on a TeamCity server (8111 in the example below) and edit `<TeamCity home>\conf\server.xml` to change the existing or add a new Connector node:

```
<Connector port="8111" protocol="HTTP/1.1"
           maxThreads="200" connectionTimeout="60000"
           redirectPort="400" useBodyEncodingForURI="true"
           proxyName="teamcity.public"
           proxyPort="400"
           secure="false"
           scheme="http"
           />
```

For HTTPS case, use `secure="true"` and `scheme="https"` attributes.

This is also described in the [comment](#).

Transfer 3 Default Agents to Another Server

This is not possible.

Each TeamCity server (Professional and Enterprise) allows to use 3 agents without any licenses.

It's a "function" of a server: users do not pay for these agents, there is no license key for them, nor are they bound to the server license key.

So, these 3 agents cannot be transferred to another server as they are "bound" to the server instance.

Each TeamCity server allows to connect agents up to number of agent license keys +3

See [more](#) on licensing.

Configure Newly Installed MySQL Server

If MySQL server is going to be used with TeamCity you should review and probably change some of its 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

Too small value in `innodb_buffer_pool_size` can affect performance significantly:

```

# 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.

innodb_file_per_table

For better performance you can enable 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.

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](#).

Import coverage results in TeamCity

TeamCity comes bundled with IntelliJ IDEA/Emma and, [since TeamCity 8.1](#), 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 published artifact name is `coverage.zip` and there is `index.html` file in it, report tab will be shown automatically.
- 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.

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.

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 [e-mail](#).

See also:

[Troubleshooting: Known Issues | Reporting Issues](#)

Common Problems

- Most frequently used documentation sections
- Build fails or behaves differently in TeamCity but not locally
- 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 ...
 - Database character set/collation-related problems
 - Character set/collation mismatch

- TeamCity displays ???? instead of national symbols
- Character set/collation-related problems
- Protocol violation error (Oracle only)
- Common Maven issues
- "Critical error in configuration file" errors

Most frequently used documentation sections

[Configuring server memory settings](#)
[Reporting server slowness issues](#)

[Back to top](#)

Build fails or behaves differently in TeamCity but not locally

If a build fails or otherwise misbehaves in TeamCity but you believe it should not, please check that the build runs fine 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.

If the TeamCity build agent is installed as a Windows service, try running the TeamCity agent 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 setup 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
- 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. Of 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.

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.

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.

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. For the database-specific information, see [PostgreSQL](#), [MySQL](#) and [MS SQL](#). If you are using MySQL or MS SQL, we recommend using **the case-sensitive collation** to avoid issues with agents on Unix-like OS.
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](#)

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](#)

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
 - Early start of the service before other resources are initialized
- 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
 - Subversion Repositories With NTLM Authorization
 - Very slow checkout on agent for Subversion in TeamCity 7.0.x
 - 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

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.

 You may want to 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).



It is recommended to run automated GUI and browser tests on a virtual machine isolated from corporate network resources, since a computer with a running desktop permanently logged into a user session might be considered a network security threat.

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](#)

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 with 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 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 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. Please try running with antivirus software uninstalled before reporting the issue to JetBrains. 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).

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:

1. use port 80 on the system so you might not be able to use TeamCity server using default 80 port.
2. corrupt layout of pages displayed in Internet Explorer. Internet Explorer Skype plugin is to blame. ([TW-13052](#)).

[Back to top](#)

Subversion issues

Subversion Repositories With NTLM Authorization

If TeamCity has problems connecting to SVN repository, while using NTLM Authentication, add an option to the `server JVM options` and to the `agent options` (if you use agent-side checkout):

`-Dsvnkit.http.methods=Basic,NTLM`

See a related [forum post](#) on this issue.

TeamCity default setting for `svnkit.http.methods` property is **Digest,Basic,Negotiate,NTLM**, so this problem should not arise regularly. However, this can also mean that more secure NTLM authentication is not used by default. To enforce using NTLM, add JVM option

`-Dsvnkit.http.methods=NTLM,Negotiate,Digest,Basic`.



Enforced NTLM authentication with TeamCity may be [unstable](#). For instance, you may notice authentication errors after each 5 days of running. One of the ways to workaround this, according to our users, is to use local machine account between TeamCity and VisualSVN server (instead of domain account)

See also a related [issue](#) on HTTP Negotiate configuration.

Very slow checkout on agent for Subversion in TeamCity 7.0.x

There are two known cases:

1. Using 1.7 working copy for checkout - please upgrade to TeamCity 7.0.2, where the problem should be fixed
2. Using pre-1.7 working copy - please start your build agent under Java 1.7. More details in this [issue](#)

svn: E175002: Received fatal alert: bad_record_mac

Please add system property `-Dsvnkit.http.sslProtocols=SSLv3` 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 JDK available for your platform (e.g. Oracle JDK [download](#)).

There were fixes in JDK 1.5 and 1.6 to address various wrong timezone reporting issues.

[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.lck` 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 seeps when downloading artifacts from TeamCity, try checking the speed on the server machine, downloading from localhost.

If the speed is OK for localhost, the issue can be in the network configuration or OS/hardware settings when combined with TeamCity(Tomcat) settings.

You can try the following approach:

in `<TeamCity home>\conf\server.xml`, change default part (port number may be different)

```
<Connector port="8111" protocol="HTTP/1.1"
           connectionTimeout="60000"
           redirectPort="8543"
           useBodyEncodingForURI="true"
        />
```

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 helps, please let us know via [email](#).

If it does not, please revert the settings.

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;}
```

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

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:

- 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
- 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
- Sending Information to the Developers

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 need to also 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 which 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](#). Please include all the relevant details of investigation, including the CPU/IO load information, what specifically is slow and what is not, etc.

For large amounts of data please use [our FTP](#).

Server Thread Dump

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 **View server thread dump** link to open the thread dump in a new browser window or **Save Thread Dump** button to save it to the <TeamCity home>/logs directory (where you can later download the files from "Server Logs").

If the web UI is not responsive, try the [direct URL](#) using the actual URL of your TeamCity server.

If the UI is not accessible, you can take a server thread dump manually using the approaches described [below](#).

You can also add "teamcity.diagnostics.requestTime.threshold.ms=30000" [internal property](#) to dump a thread dump automatically whenever there is a user-originated web request taking longer than 30 seconds into a directory `threadDumps-<date>` under TeamCity logs.

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** in the console window (this will not work for agent, since its console belongs to the launcher process).
- Alternatively, run `jstack <pid_of_java_process>` in the bin directory of the JVM used to by the application.
- To take an agent thread dump or if your server is running as a service, use the TeamCity-bundled 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 [server](#) or the [agent](#) process must be run from a console with elevated permissions (using Run as Administrator).

Under Linux

- run `jstack <pid_of_java_process>` (using jstack from the same JVM as used by the application) or `kill -3 <pid_of_java_process>`. In the latter case output will appear in <TeamCity server home>/logs/catalina.out or <TeamCity agent home>/logs/error.log.

See also [Server Performance](#) section below.

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). Since **TeamCity 8.1.2**, 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 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 you use x64 JVM, please consider using 32 bit JVM, as it will require less memory ([instructions for server](#)).
 - Try to increase the memory for the process via `'-Xmx'` JVM option, like `-Xmx1200m`. If the error message is "`java.lang.OutOfMemoryError: PermGen space`", increase the value in `-XX:MaxPermSize=270m` JVM option.
- The option needs to be passed to the process with problems:
- if it is the building process itself, use "JVM Command Line Parameters" settings in the build runner. e.g. Inspections builds may specifically need to increase the parameter;
 - refer to the corresponding documentation for TeamCity [server](#) or [agent](#). See also [Installing and Configuring the TeamCity Server#memory](#);
- If the TeamCity server is to blame and increasing memory size does not help, please report the case for us to investigate. For this, while the server is high on memory, take several server thread dumps as described [above](#), get the memory dump (see below), archive the results and [send them](#) to us for further analysis. If you have increased Xmx setting, please reduce it to the usual one before getting the dump (smaller snapshots are easier to analyze and easier to upload):
 - to get a memory dump (hprof file) automatically when an OutOfMemory error occurs, add the following [server JVM option](#) (works for JDK 1.5.0_07+): `-XX:+HeapDumpOnOutOfMemoryError`. When OOM error occurs next time, `java_xxx.hprof` file will be created in the process startup directory (`<TeamCity home>/bin` or `<TeamCity Agent home>/bin`);
 - 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 the memory dump manually is to run TeamCity server with JDK 1.6+ and use `jmap` standard JVM util. e.g. `jmap -dump:file=<file_on_disk_to_save_dump_into>.hprof <pid_of_your_TeamCity_server_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 locking the files (on Linux use `lsof`, on Windows you can use `handle` or `TCPView` for listing sockets) and the files list
3. If the number is not large, check the OS and 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 is looking suspicious and the locking process is a TeamCity one (TeamCity agent or server with no other web applications running), 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 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#Started Build Agent is not available on the server to run builds](#)

Logging events

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 debug logging level ('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 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 have 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 `<appender name="ROLL.VCS" />` node to increase 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 logging preset to "debug-SVN" in the administration web UI and then retrieve logs/teamcity-vcs.log and logs/teamcity-svn.log log files.

An alternative manual approach is also necessary for agent-side logging.

First, please enable generic VCS debug logging, as described [above](#).

Uncomment SVN-related parts (SVN.LOG appender and javasvn.output category) of the Log4j configuration file on the server and on the agent (if 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 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 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](#)

Remote Run Problems

The changes that are sent form the IDE to the server on a remote run can be retrieved from server's .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 from the URL of the change form 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 [%7r] %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 [%7r] %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`

Logging related to 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 VCS checkout mode and NOT manual VCS checkout mode (it is not possible to apply a personal patch for a build with 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 TeamCity server debug logs (see [above](#))
- In 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>` portion 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/performace/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 TeamCity Visual Studio Addin please run Microsoft Visual Studio executable (devenv.exe) with **/TeamCity.LogFile <PATH_TO_FILE>** switch specified.

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 `teamcity.agent.dotCover.log` configuration parameter to the build configuration with a path to an empty directory on 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 TeamCity server or agent process terminating unexpectedly with no apparent reason, it can happen that this is caused by Java runtime crash.

If this happens, Oracle JVM creates a file named `hs_err_pid*.log` in the working directory of the process. Working directory can be `<TeamCity server or agent home>/bin` or other like `C:\Windows\SysWOW64`. You can also search the disk for the recent files with "hs_err_pid" in the name.

See also related [posting](#).

Please send this file to us for investigation and consider updating JVM for [the server](#) (or for [agents](#)) to the latest version available.

Build Log Issues

While investigating issues related to build log we might need raw binary build log as stored by TeamCity. It can be downloaded via Web UI from the Build Log build's tab: select "Verbose" log detail and use "raw messages file" link at the top-right.

Sending Information to the Developers

Files under 10Mb 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-feedback@jetbrains.com Please do not forget to mention your TeamCity version and environment and archive the files before attaching.

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.

[Back to top](#)

Applying Patches

Microsoft Visual Source Safe Integration

To apply patch for `vss-native.exe`:

1. Shut down TeamCity server
2. Open `<TeamCity Home>/webapps/root/WEB-INF/plugins/vss/` or `<TeamCity Home>/webapps/root/WEB-INF/lib/` folder
3. Back up `vss-support.jar` file
4. Inside `vss-support.jar` file, replace `/bin/vss-native.exe` with the new one
5. Start the server

To apply full VSS plugin patch:

1. Shut down 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 to 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.

- Run the program with `/log` switch.
To get the commandline syntax and options reference, run the program without any switch.

Microsoft Team Foundation Server Integration

To apply the patch for `tf-native.exe`:

- Shutdown TeamCity server
- Open `<TeamCity Server>/webapps/root/WEB-INF/plugins/tfs` or `<TeamCity Server>/webapps/root/WEB-INF/lib/tfs-support.jar`
- Backup `tfs-support.jar`
- Inside the `tfs-support.jar` file, replace `/bin/tfs-native.exe` with the new one
- Start the server

To apply full TFS plugin patch:

- Shutdown TeamCity server
- Open `<TeamCity Home>/webapps/root/WEB-INF/plugins/tfs` or `<TeamCity Home>/webapps/root/WEB-INF/lib/tfs-support.jar`
- Back up `tfs-support.jar`
- Replace `tfs-support.jar` with the new one
- Start the server

Capturing logs from TFS-native

To enable creating logs from TFS-native:

- Locate `tf-native.exe` under TeamCity `temp` folder. File name should look like `TC-TFS-NATIVE-<digits>.exe`
- Create a copy of the file in any other folder.
- Run this program with `/log` switch.

To get command-line switches help, run the process with no parameters.

Log files will be created `<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:

- Open `<TeamCity Server>/webapps/root/update/plugins/`
- Copy `dotNetPlugin.zip` to temporary folder
- Back up `dotNetPlugin.zip`
- Extract `dotNetPlugin.zip`
- Replace contents of `/bin` folder with new files.
- Pack files again. Make sure there are no files in the root of the archive.
- Replace `dotNetPlugin.zip` file on the server. All build agents will upgrade automatically.
- Run the builds.

To enable logging from .NET runners:

- Open `<TeamCity Server>/webapps/root/update/plugins/`
- Copy `dotNetPlugin.zip` to temporary folder
- Back up `dotNetPlugin.zip`
- Extract `dotNetPlugin.zip`
- Copy `/bin/teamcity-log4net-debug.xml` to `/bin/teamcity-log4net.xml`
- You may patch Log4NET config file if you need.
- Pack files again. Make sure there is no files in the root of the plugin archive.
- Replace the `dotNetPlugin.zip` file on the server.
- All build agents should upgrade automatically.
- Run the builds.

By default, all of the log files will be stored in the `<TeamCity agent>/temp/buildTmp/TeamCity.NET` folder, log files are created for each process separately.

Enabling Detailed 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.

Do the following:

- Open the `<agent home>/plugins/dotnetplugin/bin` folder.
- 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.

Alternatively, you can set the `teamcity.agent.dotnet.debug=true` configuration parameter in a build configuration or on an agent.

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

In this section:

- Introduction to Continuous Integration
- TeamCity and Continuous Integration
- TeamCity Architecture
- Build Lifecycle in TeamCity
- Configuring Your First Build in TeamCity

Introduction to Continuous Integration

According to Martin Fowler, "Continuous Integration is a software development practice where members of a team integrate their work frequently,

usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible." To learn more about continuous integration basics, please refer to [Martin Fowler's article](#).

TeamCity and Continuous Integration

TeamCity is a user-friendly continuous integration (CI) server for developers and build engineers that is [easy to set up](#) and [free of charge](#) for small and medium teams. With TeamCity you can:

- Run parallel builds simultaneously on different platforms and environment
- Optimize the code integration cycle and be sure you never get broken code in the repository
- Detect hanging builds
- Review on-the-fly test results reporting with intelligent tests re-ordering
- Use over 600+ automated server-side inspections for Java, JSP, JavaScript and CSS
- 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.

Refer to the <http://www.jetbrains.com/teamcity/features/index.html> page to learn more about major TeamCity features. The complete list of supported platforms and environments can be found [here](#).

TeamCity Architecture

Unlike some build servers, TeamCity has distributed build grid architecture, which means that TeamCity build system comprises the **server** and a "farm" of **Build Agents** which run builds and altogether make up the so-called **Build Grid**.



A Build Agent is a piece of software that actually executes a build process. It is installed and configured separately from the TeamCity server. Although you can install an agent on the same computer as the server, we recommend to install it on a different machine for a number of reasons, first of all, for the sake of the server performance.

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.

While build agents are responsible for actually running builds, TeamCity server's job is to monitor all the connected build agents, distribute queued builds to the agents based on compatibility requirements and report the results. **The server itself runs neither builds nor tests.**

Since there is more than one participant involved into build process, it may not be clear how the data flows between the server and the agents, what is passed to the agents, how and when TeamCity gets the results, and so on. Let's sort this out by considering a simple case of a build lifecycle in TeamCity.

Related Documentation Pages

- [Installing and Configuring the TeamCity Server](#)
- [Build Agent](#)
- [Build Grid](#)
- [Setting up and Running Additional Build Agents](#)

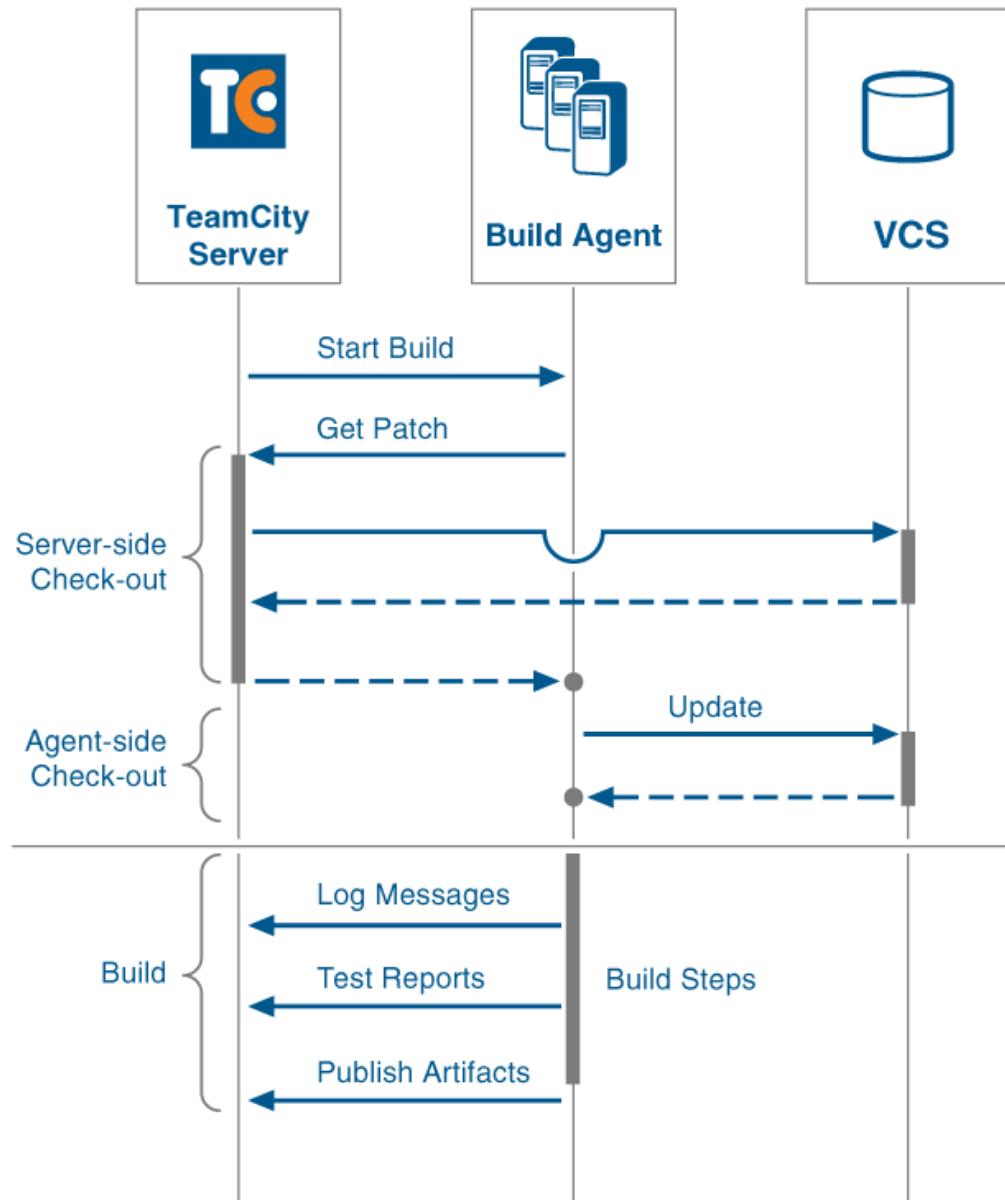
Build Lifecycle in TeamCity

To demonstrate a build lifecycle in TeamCity, we need to introduce another important term – Version Control System:



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.

Naturally, a VCS, the TeamCity server and a build agent are the three essential components required to create a build. Now, let's take a look at the data flow between them during a simple build lifecycle.



First of all, a build process is initiated by the TeamCity server when certain condition is met, for example, TeamCity has detected new changes in your VCS. In general, there is a number of such conditions which can trigger a build, but right now they are of no interest to us. To launch a build, the TeamCity server tries to select the fastest agent based on the history of similar builds, and of course it selects an agent with an appropriate environment. If there are no idle build agents among the compatible agents, the build is placed into the build queue, where it waits to be assigned to a particular agent. Once the build is assigned to a build agent, the build agent has to get the sources to run on.

At this point, TeamCity provides two possible ways for the build agent to get the sources needed for the build:

- Server-side Checkout
- Agent-side Checkout

Server-side checkout

If the server-side checkout is used, the TeamCity server exports the required sources and passes them to the build agent. Since the build agent itself does not interact with your version control system, you do not need to install a VCS client on agents. However, since sources are exported

rather than checked out, no administrative data is stored in the file system and the build agent cannot perform version control operations (like a checkin or label or update). TeamCity optimizes communications with the VCS servers by caching the sources and retrieving from the VCS server only the necessary changes: the TeamCity server sends incremental patches to the agent to update only the files which changed on the agent since the last build.

Agent-side checkout

If the agent-side checkout is used, the build agent itself checks out the sources before the build. The agent-side checkout frees more server resources and provides the ability to access version control-specific directories (.svn, CVS); that is, the build script can perform VCS operations (like check-ins into the version control). Note that not all VCS's support agent-side checkout.

When the Build Agent has all the required sources, it starts to execute **Build Steps** which are parts of the build process itself. Each step is represented by a particular **Build Runner**, which in its turn is a part of TeamCity that provides integration with a specific build tool (like Ant, Gradle, MSBuild, etc), testing framework (e.g. NUnit), or code analysis engine. Thus, in a single build you can sequentially invoke test tools, code coverage, and, for instance, compile your project.

While the build steps are being executed, the build agent sends all the log messages, test reports, code coverage results and so on to the TeamCity server on the fly, so you can monitor the build process in real time.

After finishing the build, the build agent sends **Build Artifacts** to the server. These are the files produced by a build, for example, installers, WAR files, reports, log files, etc, when they become available for download.

Related Documentation Pages

VCS Checkout Mode
Build Artifact

Configuring Your First Build in TeamCity

To configure your first build in TeamCity, perform the following steps:

1. Create a project (click to expand)

Start working with TeamCity by creating a project: a project in TeamCity is a collection of your build configurations. It allows you to organize your own projects and adjust security settings: you can assign users different permissions for each project.

Just click the *Create Project* link, then specify project's name, ID and add an optional description.

2. Create a build configuration (click to expand)

When you have created a project, TeamCity suggests to populate it with build configurations:

A **Build Configuration** in TeamCity is a number of settings that describe a class of builds of a particular type, or a procedure used to create builds. To configure a build, you need to create a build configuration, so click **Create build configuration**. Specify general settings for your build configuration, like:

- The build configuration name, **ID**, description
- The build number format: each build in TeamCity has a build number, which is a string identifier composed according to the pattern specified here. [Learn more](#). 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. You can specify the counter in the **Build counter** field.
- Artifact paths: 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, specify the paths to such artifacts here. [Learn more](#).

click the **VCS Settings** button to proceed.

3. Specify sources to be build (click to expand)

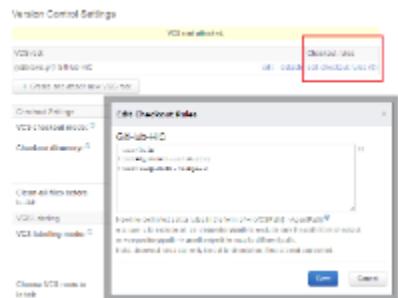
To be able to create a build, TeamCity has to know where the source code resides, thus setting up the VCS parameters is one of the mandatory steps during creating a build configuration in TeamCity.

At the Version Control Settings page, TeamCity suggests to create and attach a new VCS Root.

i 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. However, if your project resides in several version control systems, you can create 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 different 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:



Refer to the [VCS Checkout Rules](#) for details.

Also, specify whether you want TeamCity to checkout the sources on the agent or server (see [above](#)). Note, agent-side checkout is [supported not for all VCSs](#), and in case you want to use it, you need to have version control client installed at least on one agent.

4. Configure build steps (click to expand)

When creating a build configuration, it is important to configure the sequence of build steps to be executed. 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.

[Learn more](#)

Basically, these are essential steps required to configure your first build. Now you can launch it by clicking **Run** in the upper right corner of the TeamCity web UI.

However, these steps cover only a small part of TeamCity features. Refer to [Creating and Editing Build Configurations](#) sections to learn more about triggering a build, adjusting agent requirements, making your builds dependent on each other, using properties and so on.

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:

AcmeCompany.Portal

Web.config Preview

Original Web.config	Transformed Web.config (transforms applied: Web.Rel)
13 </connectionStrings>	13 </connectionStrings>
14 <appSettings>	14 <appSettings>
15 <add key="webpages:Version" val	15 <add key="webpages
16 <add key="webpages:Enabled" val	16 <add key="webpages
17 <add key="ClientValidationEnab	17 <add key="ClientVa
18 <add key="UnobtrusiveJavaSc	18 <add key="Unobtrus
19 </appSettings>	19 </appSettings>
20 <system.web>	20 <system.web>
21 <authentication mode="None" />	21 <authentication mo
22 <compilation debug="true" target	22 <compilation targe
23 <httpRuntime targetFramework="4	23 <httpRuntime targe
24 </system.web>	24 </system.web>
25 <system.webServer>	25 <system.webServer>
26 <modules>	26 <modules>
27 <remove name="FormsAuthentica	27 <remove name="Fo
28 </modules>	28 </modules>

100 %

Removed Added Help

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.

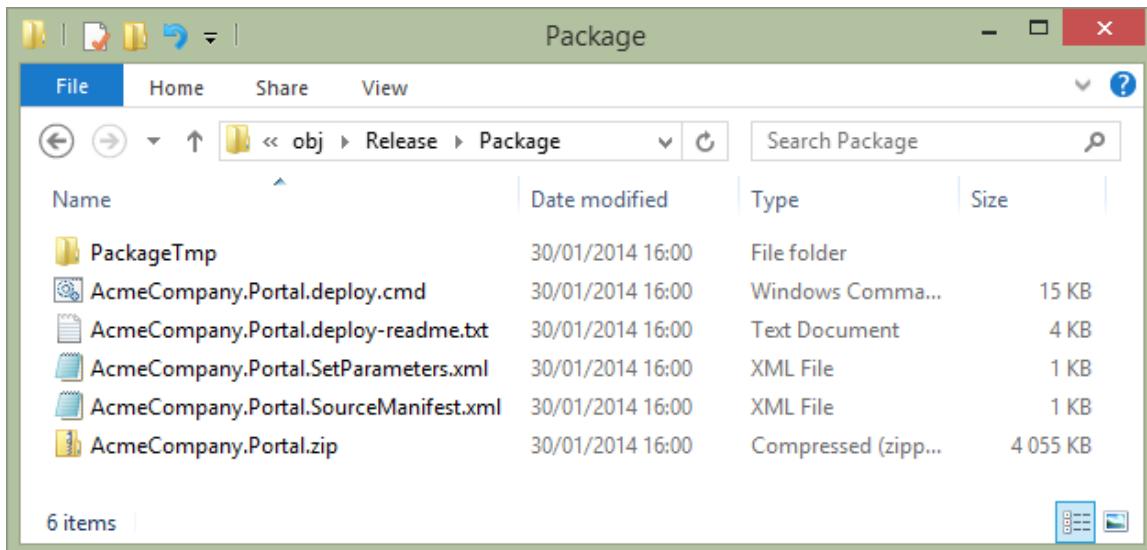
But before we deploy, let's first see how we can create a deployment package. And just so we learn about the package format, let's first do this manually by invoking msbuild.

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.

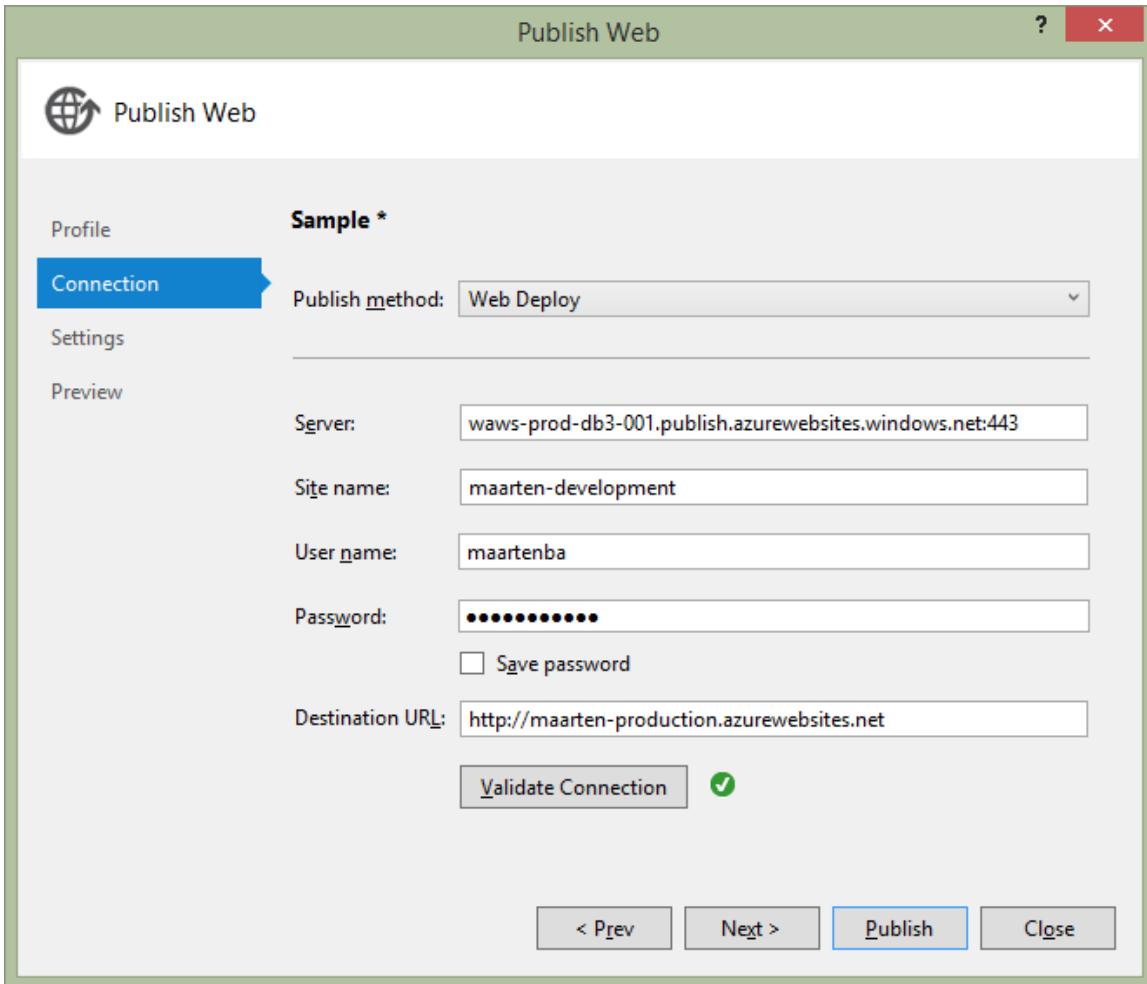
For the remainder of this tutorial, we will be covering deployment to Windows Azure Web Sites using WebDeploy, which is identical to how it works on IIS.

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.

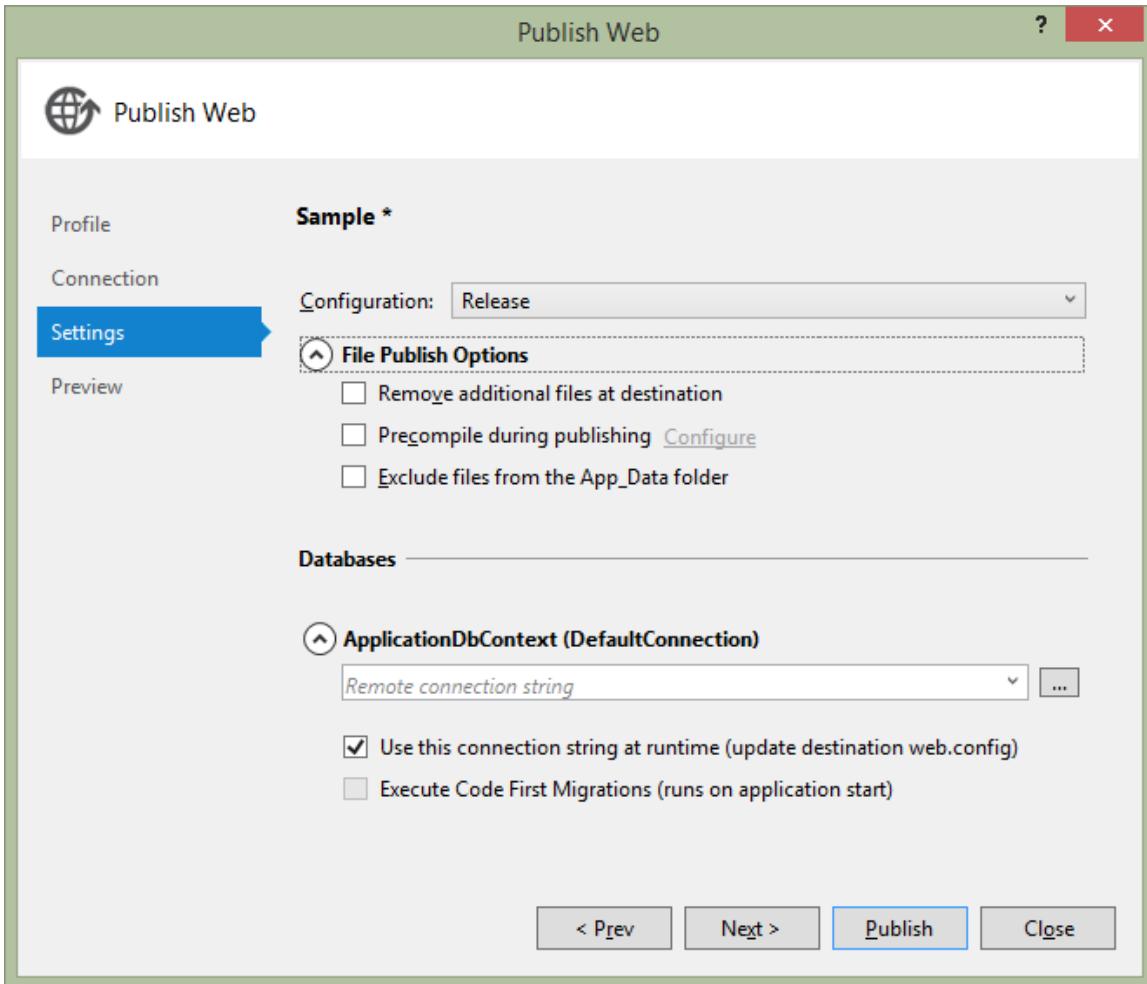
In the first step, we have to specify destination server details. This would typically be the HTTPS endpoint to the WebDeploy host (or FTP or file share details if that option was selected). After providing all details, we can validate the connection to see if it works.



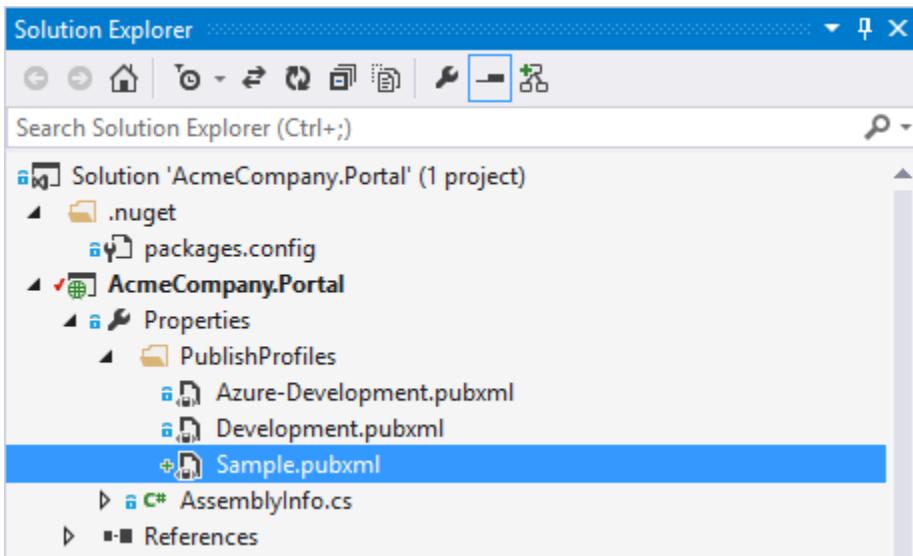
Note that instead of going through the entire wizard, Windows Azure Web Sites tooling allows importing the publish profile from the Windows Azure Management Portal. I'm showing the entire process here for when deploying to IIS.

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
Use this	Visual Studio (sln)	Build file path: AcmeCompany.Portal.sln Targets: Rebuild Configuration: Release Platform: <default>

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:ProfileTransformWebConfigEnabled=False"/>

Enter additional command line parameters to MSBuild.exe.

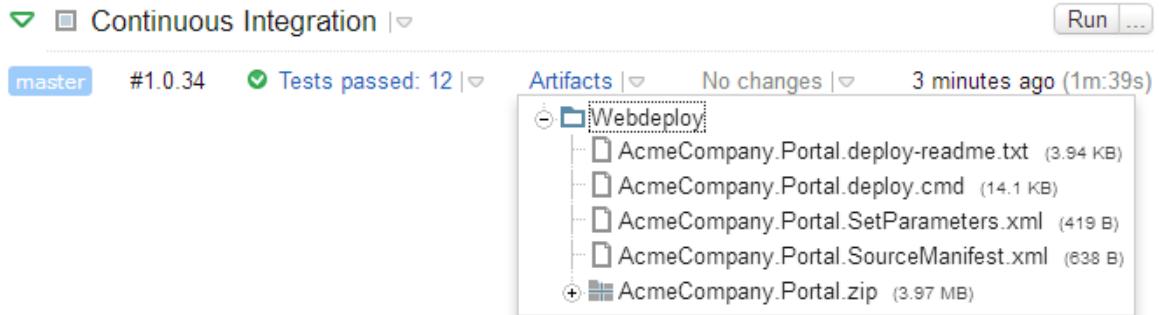
 [Hide advanced options](#)

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 a TeamCity build interface. At the top, it says "Continuous Integration" with a "Run" button. Below that, it shows "master" and "#1.0.34". It also indicates "Tests passed: 12" and "No changes" with a timestamp of "3 minutes ago (1m:39s)". A "Artifacts" section is expanded, showing a "Webdeploy" folder. Inside "Webdeploy", there are five 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).

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.

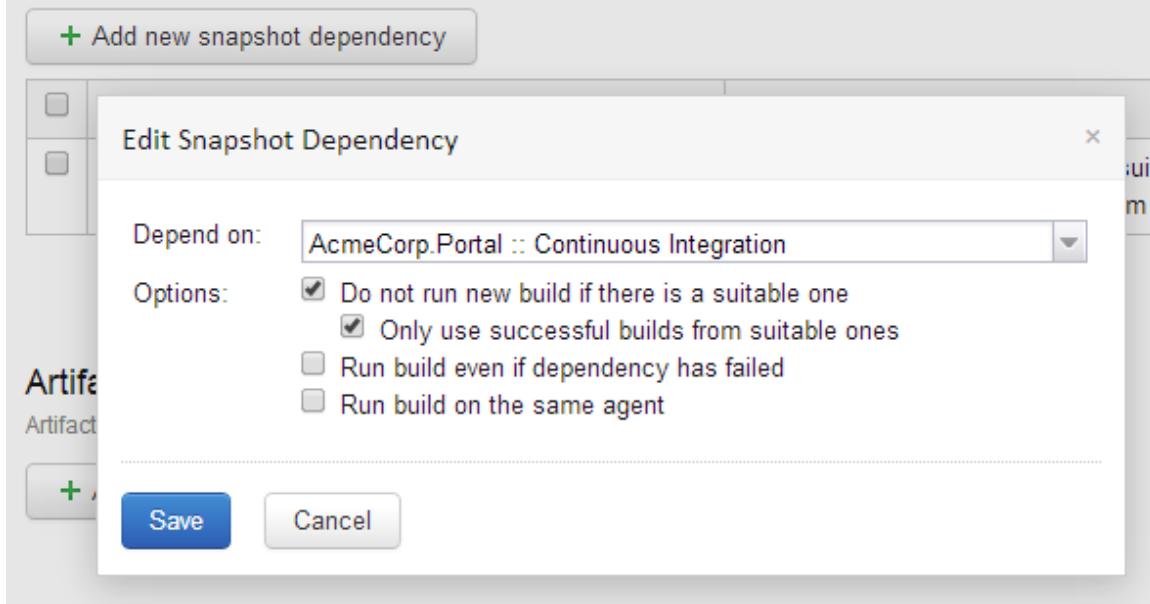


From the TeamCity Administration, copy the CI build configuration and name it differently, for example "Deploy to Windows Azure Web Sites - Development". Next, we will make some changes to the build configuration.

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. [?](#)



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.



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 publish URL>: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 file 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 `password` 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 connection 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.

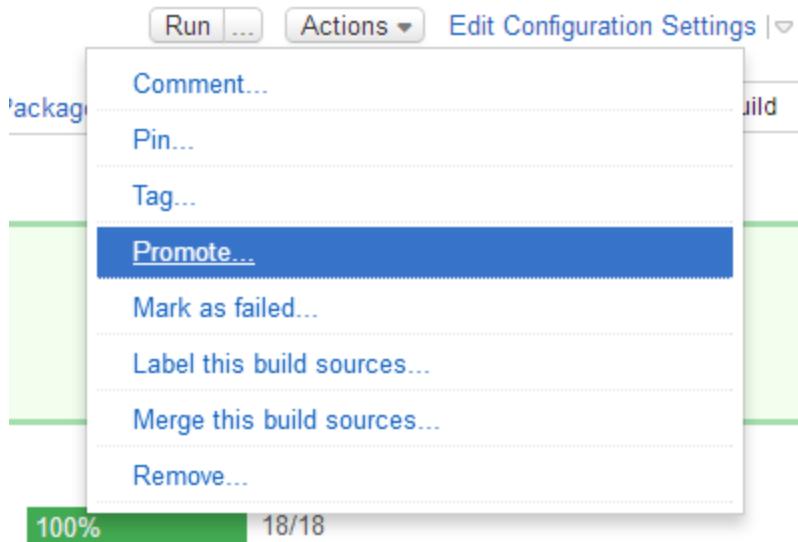
Here's what this could look like: 3 different build configurations, denoting different versions that are deployed to each target environment:

The screenshot shows the TeamCity build history for the 'AcmeCorp.Portal' project. It displays three distinct build configurations:

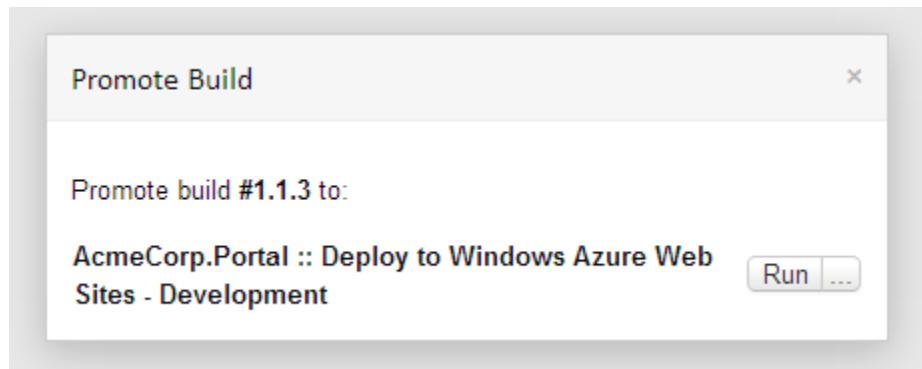
- Continuous Integration**: Build #1.1.3, Tests passed: 12, Artifacts, No changes, one minute ago (2m:55s).
- Deploy to Windows Azure Web Sites - Development**: Build #1.1.0, Tests passed: 12, Artifacts, No changes, moments ago (3m:56s). This build is a dependency of the Production environment.
- Deploy to Windows Azure Web Sites - Production**: Build #1.0.33, Tests passed: 12, Artifacts, No changes, 4 hours ago (5m:22s). This build is a dependency of the Development environment.

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.



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:

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.

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.

The screenshot shows the 'Create Build Configuration' dialog in TeamCity. The 'General Settings' section is visible, containing the following fields:

- Name:** Main
- Description:** (empty)
- Build number format:** 1.7.6.{0} (with a note explaining it's a placeholder for build counter value, up to 256 characters)
- Build counter:** 1
- Artifact paths:** A list box containing:
 - build/release/*\$system.build.number\$*
 - unitTests/codeCoverage => coverage.zipA note below explains artifact paths: "New line or comma separated paths to build artifacts. 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."
- Build options:**
 - enable hanging builds detection
 - enable status widgetA note below says: "Limit the number of simultaneously running builds (0 — unlimited) 0"

At the bottom right are 'VCS settings >>' and 'Cancel' buttons.

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.

The screenshot shows the 'New VCS Root' configuration page in TeamCity. At the top, there are navigation links for 'Projects', 'My Changes', 'Agents', 'Build Queue', and a search bar. The current user is Maarten Balliauw. The main section is titled 'Create Build Configuration > New VCS Root'. The 'Type of VCS' is set to 'Git'. The 'VCS Root Name' is 'maartenba/PHPExcel - develop'. The 'Fetch URL' is 'https://github.com/maartenba/PHPExcel.git'. The 'Default Branch' is 'develop'. The 'Branch Specification' field contains a single entry: '+*:branch name (with optional * placeholder)'. A note below the specification says: '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 'Create Build Configuration' dialog in TeamCity. The 'Administration' tab is selected. The 'Build Step' section is open, showing a 'New Build Step' configuration. The 'Runner type' is set to 'Command Line'. The 'Step name' is 'Run tests with coverage'. The 'Execute step' condition is 'Only if all previous steps were successful'. The 'Working directory' is optional. The 'Run' dropdown is set to 'Custom script'. The 'Custom script' content is a command-line script:

```
phpunit \-c phpunit.xml
```

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 web interface for the PHPEXCEL-CI project. The main navigation bar includes 'Projects', 'My Changes', 'Agents', 'Build Queue', 'Maarten Balliauw', and 'Administration'. The current build configuration is 'Main' (revision #1.7.6.2, started 07 Jan 13 08:59). The build status is red, indicating failure. The 'Tests' tab is selected, showing 66 test cases failed, with 66 new ones. The test failures are categorized under 'All tests' for the 'PHPEXCEL Unit Test Suite: DateTimeTest' category.

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 the PHPEXcel CI project. The main navigation bar includes 'Projects', 'My Changes', 'Agents', 'Build Queue', 'Maarten Balliauw', and 'Administration'. The current build is #1.7.6.4 (07 Jan 13 09:07). The 'Artifacts' tab is selected, displaying two files: 'coverage.zip' (2.41Mb) and 'PHPEXcel_1.7.6.4_doc.zip' (5.79Mb). A link to 'Download all (.zip)' is available. The total size is 8.21Mb. There are also hidden artifacts. The bottom of the page includes links for 'Help', 'Feedback', 'TeamCity Professional 7.1.3 (build 24266)', and 'License agreement'.

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.

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 'Tests' tab of the TeamCity interface. At the top, it displays 'Total test count: 3626 (145 failed); total duration: 1s'. Below this is a search bar with 'View: tests' and a dropdown for filtering. The main area is a table with columns 'Status' and 'Test'. It lists several test cases: 'Failure' for 'testXIRR with data set #2' and 'testXIRR with data s...', 'OK' for 'testToString' and 'testCurrency with da...', and 'OK' for 'AdvancedValueBind'. A context menu is open over the first failed test, with options 'Test Details', 'Investigate / Mute...', 'Open in IDE' (which is highlighted with a red box), and 'Show in Build Log'.

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!

Build Configuration General Settings

When creating a build configuration, specify the following settings:

Setting	Description
Name	The configuration name
Build Configuration ID	A unique ID of the configuration across all build configurations and templates in the system automatically generated from the build configuration name, but can also be set manually. Make sure you give a globally unique id to the build configuration and prefix it with the project id. After a build configuration is created, its ID can be changed and it is highly recommended to make corresponding changes to the bookmarked links to the web UI and calls to REST API using the ID.
Description	Optional description for the build configuration.
Build Number Format	A pattern which is resolved and assigned to the Build Number on the build start.
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 restore the counter value to 1.
Artifact Paths	Patterns to define artifacts of a build. Since TeamCity 8.1 , after the first build is run, you can browse the agent checkout directory to configure artifacts paths
Build Options	Specify additional options for the builds of this build configuration