

1. TeamCity Documentation	4
1.1 What's New in TeamCity 9.1	5
1.2 What's New in TeamCity 9.0	10
1.3 Concepts	14
1.3.1 Build Artifact	15
1.3.2 Build Configuration	16
1.3.3 Build Log	18
1.3.4 Build Queue	18
1.3.5 Build State	20
1.3.6 Build Tag	22
1.3.7 Change	22
1.3.8 Change State	23
1.3.9 Difference Viewer	24
1.3.10 Project	25
1.3.11 Agent Home Directory	26
1.3.12 Agent Requirements	27
1.3.13 Agent Work Directory	27
1.3.14 Authentication Modules	28
1.3.15 Build Agent	28
1.3.16 Build Chain	29
1.3.17 Build Checkout Directory	30
1.3.18 Build Configuration Template	32
1.3.19 Build Grid	34
1.3.20 Build History	34
1.3.21 Build Number	34
1.3.22 Build Runner	34
1.3.23 Build Working Directory	35
1.3.24 Clean Checkout	35
1.3.25 Clean-Up	36
1.3.26 Code Coverage	39
1.3.27 Code Duplicates	39
1.3.28 Code Inspection	39
1.3.29 Continuous Integration	40
1.3.30 Dependent Build	40
1.3.31 Guest User	42
1.3.32 History Build	42
1.3.33 Notifier	43
1.3.34 Personal Build	43
1.3.35 Pinned Build	43
1.3.36 Pre-Tested (Delayed) Commit	43
1.3.37 Remote Run	44
1.3.38 Role and Permission	45
1.3.39 Run Configuration Policy	46
1.3.40 TeamCity Data Directory	46
1.3.41 TeamCity Specific Directories	49
1.3.42 User Account	49
1.3.43 User Group	50
1.3.44 Wildcards	50
1.3.45 Already Fixed In	51
1.3.46 First Failure	52
1.3.47 Super User	52
1.3.48 Identifier	52
1.3.49 VCS root	53
1.3.50 Remote Debug	54
1.3.51 Favorite Build	55
1.3.52 Agent Cloud Profile	56
1.3.53 TeamCity Home Directory	56
1.3.54 Revision	57
1.4 Supported Platforms and Environments	57
1.4.1 Testing Frameworks	62
1.4.2 Code Quality Tools	62
1.5 Installation and Upgrade	64
1.5.1 Installation	64
1.5.1.1 Installing and Configuring the TeamCity Server	64
1.5.1.2 Setting up and Running Additional Build Agents	71
1.5.1.2.1 Build Agent Configuration	79
1.5.1.3 TeamCity Integration with Cloud Solutions	80
1.5.1.3.1 Setting Up TeamCity for Amazon EC2	83
1.5.1.4 Installing Additional Plugins	86
1.5.1.5 Installing Agent Tools	86
1.5.2 Upgrade Notes	87
1.5.3 Upgrade	106

1.5.4 TeamCity Maintenance Mode	109
1.5.5 Setting up an External Database	110
1.5.6 Migrating to an External Database	114
1.6 User's Guide	116
1.6.1 Managing your User Account	116
1.6.2 Subscribing to Notifications	117
1.6.3 Viewing Your Changes	120
1.6.4 Working with Build Results	121
1.6.5 Investigating Build Problems	127
1.6.6 Viewing Tests and Configuration Problems	128
1.6.7 Viewing Build Configuration Details	129
1.6.8 Statistic Charts	130
1.6.9 Search	131
1.6.10 Maven-related Data	134
1.7 Administrator's Guide	134
1.7.1 TeamCity Configuration and Maintenance	134
1.7.1.1 Configuring Authentication Settings	135
1.7.1.1.1 LDAP Integration	138
1.7.1.1.2 NTLM HTTP Authentication	147
1.7.1.1.3 Enabling Guest Login	149
1.7.1.2 TeamCity Data Backup	149
1.7.1.2.1 Creating Backup from TeamCity Web UI	150
1.7.1.2.2 Creating Backup via maintainDB command-line tool	151
1.7.1.2.3 Manual Backup and Restore	152
1.7.1.2.4 Backing up Build Agent's Data	154
1.7.1.2.5 Restoring TeamCity Data from Backup	154
1.7.1.3 Projects Import	155
1.7.1.4 TeamCity Startup Properties	157
1.7.1.5 Configuring Server URL	157
1.7.1.6 Configuring TeamCity Server Startup Properties	158
1.7.1.6.1 TeamCity Tweaks	159
1.7.1.7 Configuring UTF8 Character Set for MySQL	160
1.7.1.8 Setting up Google Mail and Google Talk as Notification Servers	160
1.7.1.9 Using HTTPS to access TeamCity server	161
1.7.1.10 TeamCity Disk Space Watcher	162
1.7.1.11 TeamCity Server Logs	163
1.7.1.12 Build Agents Configuration and Maintenance	165
1.7.1.12.1 Agent Pools	166
1.7.1.12.2 Configuring Build Agent Startup Properties	167
1.7.1.12.3 Viewing Agents Workload	168
1.7.1.12.4 Viewing Build Agent Details	169
1.7.1.12.5 Viewing Build Agent Logs	170
1.7.1.13 TeamCity Memory Monitor	171
1.7.1.14 Disk Usage	172
1.7.1.15 Server Health	173
1.7.1.16 Build Time Report	175
1.7.1.17 TeamCity Monitoring and Diagnostics	176
1.7.2 Managing Projects and Build Configurations	177
1.7.2.1 Creating and Editing Projects	177
1.7.2.2 Working with Meta-Runner	179
1.7.2.3 Archiving Projects	182
1.7.2.4 Creating and Editing Build Configurations	183
1.7.2.4.1 Configuring General Settings	185
1.7.2.4.2 Configuring VCS Settings	188
1.7.2.4.3 Configuring Build Steps	210
1.7.2.4.4 Adding Build Features	259
1.7.2.4.5 Configuring Unit Testing and Code Coverage	271
1.7.2.4.6 Build Failure Conditions	291
1.7.2.4.7 Configuring Build Triggers	294
1.7.2.4.8 Configuring Dependencies	304
1.7.2.4.9 Configuring Build Parameters	314
1.7.2.4.10 Configuring Agent Requirements	324
1.7.2.5 Copy, Move, Delete Build Configuration	324
1.7.2.6 Ordering Projects and Build Configurations	325
1.7.2.7 Working with Feature Branches	325
1.7.2.8 Triggering a Custom Build	329
1.7.2.9 Ordering Build Queue	331
1.7.2.10 Muting Test Failures	332
1.7.2.11 Changing Build Status Manually	333
1.7.2.12 Customizing Statistics Charts	333
1.7.2.13 Storing Project Settings in Version Control	334
1.7.3 Managing Licenses	336

1.7.3.1 Licensing Policy	336
1.7.3.2 Third-Party License Agreements	339
1.7.4 Integrating TeamCity with Other Tools	342
1.7.4.1 Mapping External Links in Comments	342
1.7.4.2 External Changes Viewer	343
1.7.4.3 Integrating TeamCity with Issue Tracker	344
1.7.4.3.1 Bugzilla	345
1.7.4.3.2 JIRA	346
1.7.4.3.3 YouTrack	346
1.7.5 Managing User Accounts, Groups and Permissions	346
1.7.5.1 Managing Users and User Groups	347
1.7.5.2 Viewing Users and User Groups	348
1.7.5.3 Managing Roles	349
1.7.6 Customizing Notifications	349
1.7.7 Assigning Build Configurations to Specific Build Agents	353
1.7.8 Patterns For Accessing Build Artifacts	354
1.7.9 Mono Support	356
1.7.10 Maven Server-Side Settings	357
1.7.11 Tracking User Actions	357
1.8 Installing Tools	358
1.8.1 IntelliJ Platform Plugin	358
1.8.2 Eclipse Plugin	359
1.8.3 Visual Studio Addin	361
1.8.4 Windows Tray Notifier	361
1.8.4.1 Working with Windows Tray Notifier	362
1.8.5 Syndication Feed	365
1.9 Extending TeamCity	365
1.9.1 Build Script Interaction with TeamCity	366
1.9.2 Accessing Server by HTTP	376
1.9.3 REST API	378
1.9.4 Including Third-Party Reports in the Build Results	390
1.9.5 Custom Chart	391
1.9.5.1 Edit Custom Chart Limitations	396
1.9.6 Developing TeamCity Plugins	396
1.9.6.1 Typical Plugins	397
1.9.6.1.1 Build Runner Plugin	397
1.9.6.1.2 Risk Tests Reordering in Custom Test Runner	399
1.9.6.1.3 Custom Build Trigger	400
1.9.6.1.4 Extending Notification Templates Model	400
1.9.6.1.5 Issue Tracker Integration Plugin	401
1.9.6.1.6 Version Control System Plugin	403
1.9.6.1.7 Version Control System Plugin (old style - prior to 4.5)	407
1.9.6.1.8 Custom Authentication Module	413
1.9.6.1.9 Custom Notifier	418
1.9.6.1.10 Custom Statistics	418
1.9.6.1.11 Custom Server Health Report	419
1.9.6.1.12 Extending Highlighting for Web diff view	421
1.9.6.2 Bundled Development Package	427
1.9.6.3 Open API Changes	428
1.9.6.4 Plugin Types in TeamCity	433
1.9.6.5 Plugins Packaging	433
1.9.6.6 Server-side Object Model	438
1.9.6.7 Agent-side Object Model	439
1.9.6.8 Extensions	440
1.9.6.9 Web UI Extensions	441
1.9.6.10 Plugin Settings	445
1.9.6.11 Development Environment	445
1.9.6.12 Developing Plugins Using Maven	445
1.9.6.13 Plugin Development FAQ	447
1.9.6.14 Getting Started with Plugin Development	448
1.10 How To...	451
1.11 Troubleshooting	469
1.11.1 Common Problems	469
1.11.2 Known Issues	474
1.11.3 Reporting Issues	479
1.11.4 Applying Patches	487
1.11.5 Visual C Build Issues	489
1.12 Getting Started	489
1.12.1 Continuous Delivery to Windows Azure Web Sites (or IIS)	494
1.12.2 Getting started with PHP	504

TeamCity Documentation

Welcome to the documentation space for JetBrains TeamCity 9.x! If you are using an earlier TeamCity version, please refer to documentation for your release.

Get Started

- Continuous integration with TeamCity
- TeamCity Concepts

Install

- Install the TeamCity server
- Set up Additional Build Agents

Administer

- Configure and maintain the TeamCity server
- Work with projects and build configurations
- Manage users and their permissions
- Upgrade

Extend and Develop

- Customize your TeamCity
- REST API
- Available TeamCity plugins
- Develop your own plugin

Find Answers

- How To...
- Common Problems
- Known Issues
- Public issue tracker
- Community forum
- Video user guide

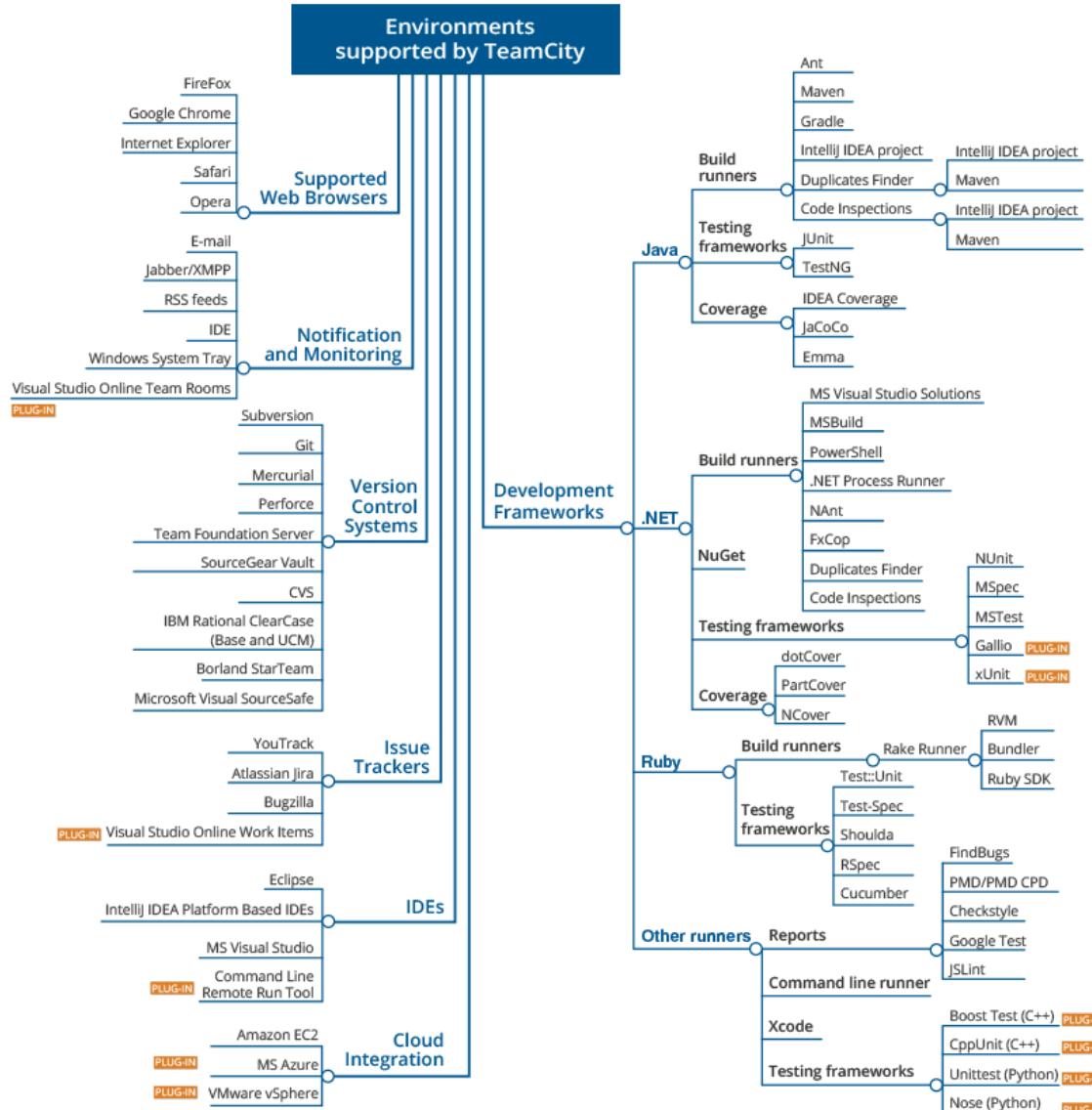
Learn More and Contact Us

- TeamCity Official Site
- Official TeamCity Blog
- Feedback

TeamCity 9.x 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. See details at [Supported Platforms and Environments](#).

The diagram is **clickable**: point to a component, click on it and jump to its description:



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 9.1

- Unidirectional Agent-Server Communication
- Versioned Settings Improvements
 - Selecting Project Settings to Run a Build on: "True" Historical Builds and more
 - Known Limitations
 - Versioned Project Settings in Subversion and Perforce
- Enhanced .NET tools Support
 - Support for the Latest Versions of Microsoft Tools
 - Visual Studio Tests Runner
 - Support for NUnit 3.0
- UI Improvements and Usability

- Administrator-defined Ordering of Projects and Build Configurations
- Specifying TeamCity Data Directory on the TeamCity First Start Page
- Create Build Configuration from URL
- Schedule Trigger Improvements
- Multiple Artifact Directories Paths
- Restricting Personal Builds
- ANSI-style Coloring in Build Logs
- Truncated Paths to Build configurations
- New Build Features
 - SSH Agent
 - File Content Replacer
- REST API Improvements
- Other Improvements
- Fixed Issues
- Previous Releases

Unidirectional Agent-Server Communication

Traditionally TeamCity requires two HTTP connections between the server and an agent: the server creates an HTTP connection to the agent and sends commands, like start/stop build, etc.; the agent also establishes an HTTP connection to the server and sends build results. Only the agent-to-server connection can be configured over HTTPS: it is not possible to configure HTTPS for the server-to-agent connection.

Now only the [agent-to-server HTTP connection](#) is required, which means that:

- it is possible to fully secure the communication between the agent and the server using the [HTTPS protocol](#)
- an agent which is inaccessible for the TeamCity server (because of the firewall on the agent or its deployment to a private network) can now work with the server.

The communication protocol used by a particular agent can be viewed on the **Agents\ | <Agent Name>\ | Agent Summary** tab, the **Details** section.

The screenshot shows the TeamCity interface with the navigation bar at the top. Below it, the 'Agents' tab is selected, followed by 'Agent Summary'. The main content area displays the 'Status' and 'Details' sections. In the 'Details' section, under the 'Port' heading, the 'Communication protocol' dropdown is set to 'unidirectional'. Other visible details include the agent name ('teamcity-linux-205'), IP address ('172.28.198.199'), and operating system ('Ubuntu, version 11.10.6414gen').

Versioned Settings Improvements

We have made significant improvements in versioned settings for projects.

Selecting Project Settings to Run a Build on: "True" Historical Builds and more

Now TeamCity can be configured to use different settings to start builds in projects where [versioned settings](#) are enabled, which means you have the following possibilities at hand:

- it is possible to configure on the project level whether builds of the current projects will use the current build configuration settings or whether the settings from VCS will be preferred. When the settings from your VCS are used, you can run a "true" [history build](#), i.e. the build using an older revision of the sources can now be run on the project settings that existed on the server when the selected source code changes were made.
- if you are using TeamCity feature branches, you can define a branch specification in the VCS root used for versioned settings, and TeamCity will run a build in the branch using the settings from this branch
- you can now start [remote run/pre-tested commit](#) with changes made in the .teamcity directory in your IDE, and these changes will affect the build behavior



Known Limitations

The following changes will be ignored:

- changes in snapshot dependencies: TeamCity will continue reading snapshot dependencies settings from the build configuration
- changes in build features working on the server ([Automatic Merge](#) and [VCS Labeling](#))
- changes of the attached VCS roots and [VCS checkout rules](#)
- changes of some settings which do not affect the build behavior on an agent, for instance, build triggers, general settings like [limitation on a number of concurrently running builds](#), [remote run restrictions](#).

Versioned Project Settings in Subversion and Perforce

Now TeamCity allows storing the project configuration settings in a Subversion and Perforce version control repository. Previously, the only supported version controls were Git and Mercurial.

The settings are stored in the `.teamcity` directory in the root of the repository.

Enhanced .NET tools Support

Support for the Latest Versions of Microsoft Tools

The latest versions of all tools included in the Visual Studio 2015 lineup (namely Visual Studio 2015, MSBuild 2015, TFS 2015, MSTest 2015 (14.0), and FxCop 2015) are now supported by TeamCity.

Windows PowerShell 5.0 is also supported.

Visual Studio Tests Runner

Visual studio comes with two very similar test runners - MSTest and VSTest. TeamCity supports MSTest out of the box; for VSTest a [separate plugin](#) had to be installed.

Now we combined them into a single [Visual Studio Tests](#) runner.

Note that after upgrade to TeamCity 9.1, MSTest build steps are automatically converted to the Visual Studio Tests runner steps, while VSTest steps remain unchanged.

Support for NUnit 3.0

The TeamCity [NUnit](#) runner supports NUnit 3.0. now. Starting with NUnit 3.0.0 Beta 2, NUnit detects if it is run by a TeamCity build agent and automatically switches to reporting test runs using TeamCity service messages.

As a coverage engine, JetBrains dotCover only is supported for NUnit 3.0.

UI Improvements and Usability

Administrator-defined Ordering of Projects and Build Configurations

By default, TeamCity displays projects, their subprojects, and build configurations on the [Projects Overview](#) page in the alphabetical order. Starting from this release, project administrators can apply [custom ordering to subprojects](#), and build configurations of a project on the [Project Settings](#) page and use it as the default one for all the team members, thus saving them the effort of defining the order manually. Individual users can still manually tweak the display using the up-down button on the [Configure Visible Projects](#) pop-up.

Specifying TeamCity Data Directory on the TeamCity First Start Page

The configuration of TeamCity data directory has been moved from the installation wizard to the TeamCity startup screens. The TeamCity data directory path specified on startup screens will be stored in the `<TeamCity installation directory>/conf/teamcity-startup.properties` file.

If the `TEAMCITY_DATA_PATH` environment variable is specified, the value of this variable will be used as the path to the data directory for compatibility reasons.



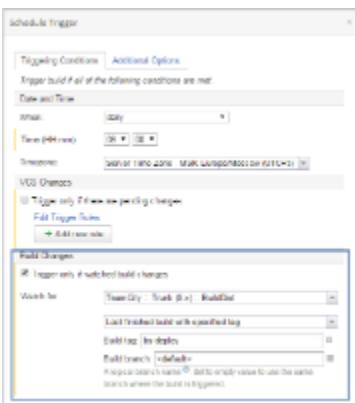
Create Build Configuration from URL

In addition to [creating a project from URL](#), TeamCity now comes with an option to [create a build configuration](#) from a VCS URL. All you need to do is enter a VCS URL in the *create configuration* wizard, select default options to create a build configuration and that's it!

Schedule Trigger Improvements

TeamCity has different triggers suitable for various use cases, and yet some of the cases are hard to automate. To cover the scenarios when a build chain should be continued automatically based on some event, we added a new option to the [Schedule Trigger](#). Now the Schedule Trigger can watch builds in other build configurations and trigger a build if these builds change. The watched build is specified similarly to selecting a build in the [artifact dependency](#).

To continue a chain once some build in the chain finishes, we recommend using the improved Schedule Trigger rather than the Finish Build Trigger due to limitations of the Finish Build Trigger.



Multiple Artifact Directories Paths

By default, TeamCity stores build artifacts in `<TeamCity data directory>/system/artifacts`. Now the storage is [not only configurable](#), [but also scalable](#): you can use multiple locations (internal or external) for your artifacts and conveniently configure the paths to the storage using the [Administration | Global Settings](#) in the TeamCity UI.

If a new build starts, its artifacts are published into the first directory in the list. When looking for artifacts of earlier builds, TeamCity will go through the list of the artifact directories to locate the directory where build artifacts are stored.

Restricting Personal Builds

A new build option in the [General settings](#) of a build configuration allows you to restrict running personal builds. It is useful to disallow remote runs with personal changes from IDEs for important build configurations like deployment ones.

By default, triggering personal builds is enabled; uncheck the **allow triggering personal builds** option to disable it.

ANSI-style Coloring in Build Logs

TeamCity build logs now supports ANSI color escape sequences. The logs also render clickable hyperlinks now.



Truncated Paths to Build configurations

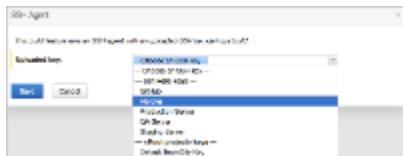
Instead of showing the full path to a build configuration in the web interface, the project name part of the path is truncated to <...>.

New Build Features

SSH Agent

This new build feature runs an SSH agent with the selected uploaded SSH key during a build. When your build script runs an SSH client, it uses the SSH agent with the loaded key.

You no longer need to manage SSH keys on the agent manually.



The first time you connect to a remote host, the SSH client asks if you want to add a remote host's fingerprint to the known hosts database at `~/.ssh/known_hosts`. To avoid such prompts during a build, you need to configure the known hosts database beforehand. If you trust the hosts you are connecting to, you can disable known hosts checks:

- either **for all connections** by adding something like this in `~/.ssh/config`:

```
Host *
  StrictHostKeyChecking no
```

- or **for an individual command** by running an ssh client with the `-o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no` options. You can find more information in the man pages for `ssh`, `ssh-agent` and `ssh-add` commands.

File Content Replacer

TeamCity has a general [file content replacer](#) now. Compared to [AssemblyInfo Patcher](#), the new feature enables you to change a wider range of values for the duration of the build in files specified by the user. It provides value pre-sets for replacement; custom values can also be used. The changes are valid for the duration of the build; on build finishing, the changed source files are reverted.



REST API Improvements

- `/app/builds` request supports more abilities to filter builds: "failed to start" builds via `failedToStart:any/failedToStart:false` locator, include queued and running builds via `state:(any)` locator, get builds within a build chain via `snapshotDependency:(to:(<build locator>))`, `defaultFilter:false` locator, etc.
- ability to get build's artifacts in archive via `/app/rest/builds/<build locator>/archived` request
- ability to change build parameter type specification
- ability to list only not archived projects via `archived:false` locator

Other Improvements

- Bundled Ant upgraded to version 1.9.6. Ant-runner build steps require Java 1.5 at least.
- The bundled Java has been updated to version 1.8.0_45
- Bundled IntelliJ IDEA is upgraded to version 14.1.4
- Xcode 7.0 is supported (tested with 7.0 beta)
- The default Subversion working copy format has been changed to 1.8
- [logging preset](#) selected on the [Diagnostics](#) page is preserved on server restart
- More reliable [TeamCity server shutdown](#)
- The TeamCity Web UI now allows specifying custom command line parameters to pass to dotCover.

- The Run Custom Build dialog allows adding a build to favorites using the corresponding checkbox on the Comments and Tags tab.
- [Simple Build Tool \(Scala\)](#) plugin is now bundled with TeamCity
- It is now possible to specify multiple VCS usernames for each level (default, VCS type or VCS root)
- SSH Key support for Subversion checkout on agent [TW-35092](#)
- It is now possible to identify not-actual tests on the Investigations page

Fixed Issues

- [full list of fixed issues](#)

Previous Releases

[What's New in TeamCity 9.0](#)

What's New in TeamCity 9.0

- Moving projects between TeamCity servers
- Storing project settings in Git and Mercurial
- Server clean-up in background
- Simplified custom charts management
- Favorite builds
- Build Time report
- Meta-Runner improvements
- 7-zip support for published artifacts
- Push parameters to dependencies
- Customizable behavior on snapshot dependency failure
- Per-project configuration of issue tracker integration
- VCSSs-related improvements
 - Perforce streams and new options support
 - Other version controls
- Tagging queued / running builds
- WebSockets for server events
- Gradle project in Inspections (IntelliJ IDEA) and Duplicates Finder (Java)
- Backup / restore performance improvements
- Other
- Fixed Issues
- Previous Releases

Moving projects between TeamCity servers

Now TeamCity provides the ability to move projects among servers: you can transfer projects with all their data (settings, builds and changes history, etc.) and with your TeamCity user accounts from one server to another.

All you need to do is create a usual backup file on the source TeamCity server containing the projects to be imported, put the backup file into the <TeamCity Data Directory>/import directory on the target server and follow the import steps on the [Administration | Projects Import](#) page.

For now there are still some limitations:

- Only backup files created with a TeamCity server of the same version as the current server are supported.
 - the backup files do not contain artifacts, so artifacts are not imported automatically; the same applies to build logs, because since version 9.0 build logs are stored under build artifacts - however, we provide scripts enabling you to move artifacts and logs.
 - audit records are not imported
 - global server settings (authentication schemes, custom roles, etc.) are not imported
- More details on this feature are available in [our documentation](#).

Storing project settings in Git and Mercurial

TeamCity provides the [two-way synchronization](#) of project settings with the version control. Synchronization is supported for Git and Mercurial only. You can enable synchronization on the [Versioned Settings](#) page in the project administration area. With the synchronization enabled:

- each administrative change you make to the project settings in the TeamCity Web UI is committed to the version control; changes to several projects will generate a cumulative commit;
- if you alter the settings in the version control, the TeamCity server will detect the modifications and apply them to the project on the fly. Enabling synchronization for the project also enables it for all its subprojects. TeamCity synchronizes all changes to the project settings (including modifications of build configurations, templates, VCS roots, etc.) with the exception of SSH keys. The project settings are stored in the `.teamcity` folder in the repository.

As soon as synchronization is enabled in a project, TeamCity will make an initial commit in the selected repository for the whole project tree (the project with all its subprojects) to import the current settings.

TeamCity will not only synchronize the settings, but will also display changes to the project settings the same way it is done for regular changes in

the version control. You can configure the changes to be displayed for the affected build configurations. Such changes are ignored by build triggers.

Server clean-up in background

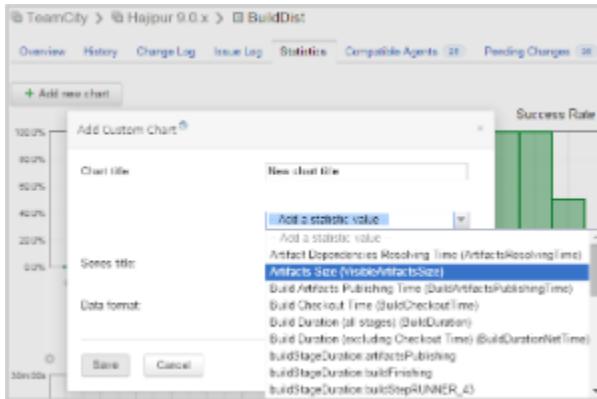
If you use TeamCity in a distributed team, you no longer have to face the pain point of finding a time slot suitable for everyone to perform your server clean-up: TeamCity now runs clean-up in background, which means that there is zero maintenance downtime.

Note that if you are using the HSQL database, there is still a short period of server unavailability when the HSQL database is being compacted. For production purposes, switching to an external database is recommended.

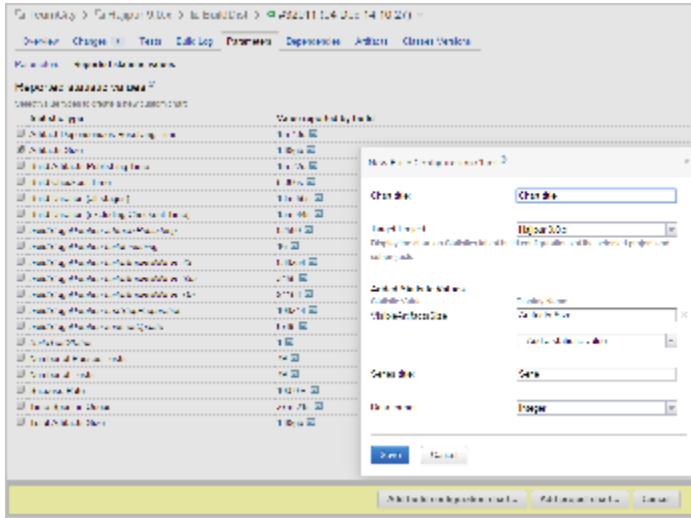
Simplified custom charts management

Earlier, to edit custom charts you had to manually modify xml files. Now you can easily manage custom charts using the TeamCity Web UI:

- on the **Statistics** tab for a project or build configuration using the **Add new chart** button.



- on the **Parameters** tab of the build results page, where the list of **Reported statistic values** provides checkboxes to select the statistic value for a new project- or build-configuration-level chart.



In addition, the custom charts definitions are stored on the project level, so they are available to Project Administrators. More details on this feature are available in [our documentation](#).

Note that the default charts cannot be modified.

Favorite builds

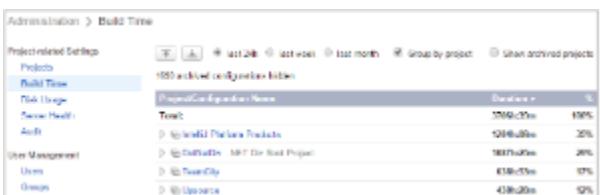
To easily access builds you want to monitor, you can mark them as favorite. Optionally, any manually triggered build can be added to your **favorites** automatically. The marked builds will be listed on the **My Favorite Builds** page available in your user profile.



You can also [configure notifications](#) on your favorite builds.

Build Time report

Similarly to the Disk usage, now you can see how many resources are taken up by a project with its subprojects: the new [Build Time Report](#) is now available in the Project-related Settings of the TeamCity **Administration** area. By default, the report displays total build duration within the last 24 hours for the server and individual projects with the percentage in relation to the parent project. The scope can be further drilled down to the build time of an individual build configuration.



Meta-Runner improvements

Working with meta-runners has become easier: now if you need to fix a meta-runner, there is no need to edit its xml definition; instead you can create a build configuration from the meta-runner, debug and fix it. Then you use the **Extract meta-runner** action to apply changes to the existing meta-runner.

Besides, you no longer need access to the <TeamCity Data Directory> on the disk to upload a meta-runner - you can now upload it from the TeamCity Web UI.

7-zip support for published artifacts

TeamCity can now compress published artifacts to a 7-zip archive:

`*/> dist.7z`

As with other supported compression algorithms, you can also specify artifact dependency for the files inside a 7-zip archive:
`dist.7z!folder/**`

Browsing inside 7-zip archives is also supported, but not browsing inside a 7-zip placed into another 7-zip archive.

Push parameters to dependencies

When you add a build which depends on other builds to the queue, you can change the parameters of the builds this build depends on. For example, our ReSharper team has a compile build producing dlls containing the debugging information which the installer build depends on. Most of the time we need the installer with the debug information, but for the release build we need the debug switched off. Now we can specify the parameter disabling the debug mode in the installer build and push it to all builds in the chain using the `reverse.dep.*` in the parameter name.

Customizable behavior on snapshot dependency failure

Previously TeamCity offered limited control over dependent build status if a dependency failed. Now new options are available for more flexible configuration.

For each failed or failed to start dependency you can select one of the four options:

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

Per-project configuration of issue tracker integration

The configuration of integration with issue trackers has moved to the project level, and now users with the Project Administrator permissions can access this feature in the Project Settings. If you are using different issue-trackers for different projects, you will definitely benefit from this

improvement.

Enabling integration for the project also enables it for all its subprojects; if the configuration settings are different in a subproject, its settings have priority over the project's settings.

VCSS-related improvements

Perforce streams and new options support

- You can configure a Perforce VCS Root to monitor Perforce streams. Both server and agent-side checkout modes are supported.
- New options were introduced for Perforce VCS roots: support for `p4 clean` introduced in Perforce 2014.1, ability to provide extra `p4 sync` options, like `--parallel`

Other version controls

- NTLM authentication can be used for Subversion repositories.
- TeamCity now supports configuration of Mercurial options per repository leaving behind the old approach which implied editing global mercurial configuration files on the server and agents. Now it is possible to enable some mercurial extensions only in the repositories where they are required.
- The `Use mirrors` option has been added to Git and Mercurial VCS root settings page.
- Better git/hg progress reporting for the agent-side checkout: now all executed commands are shown in a build log.

Tagging queued / running builds

TeamCity now provides an option of [tagging](#) queued builds. There are several ways of doing it:

- from the [Run Custom Build](#) dialog
- from the **Actions** menu of the queued build page

WebSockets for server events

Now TeamCity establishes a WebSocket connection between the server and the browser if both parties support the WebSocket connection. This allows delivering server events to the browser more efficiently, making the TeamCity Web UI more responsive.

TeamCity switches to polling when the WebSocket connection is unavailable in the following cases:

- TeamCity requires Tomcat version 7.0.43 and above for WebSockets to work; TeamCity will switch to polling automatically if this is not the case.
- The WebSocket connection is not available for Internet Explorer versions below 10 and Safari; for them polling will be used.
- If the TeamCity server is behind a proxy, [additional configuration](#) is required for the WebSocket connection to work.

Gradle project in Inspections (IntelliJ IDEA) and Duplicates Finder (Java)

TeamCity inspections and duplicate code analysis tasks are now available for Gradle projects in addition to IntelliJ IDEA and Maven projects.

Backup / restore performance improvements

After upgrading to 9.0, TeamCity will start a background process optimizing the VCS changes database structure. After the process finishes, backup and restore should start working faster. You can safely create backups while this process is running, but it is not recommended to use these backup files for projects import, as some VCS related data may not be imported to the target server.

Other

- Notifications can be configured to be sent out for builds on non-default branch. This is configured via "Edit Branch Filter" notification rules setting
- Floating point numbers support for statistic values: TeamCity supports fractional numbers with up to 6 decimal points for all statistic values
- Bundled Maven 3.2.3
- Now the Maven artifact dependency trigger can use authorization configured using the new advanced settings of the trigger.
- Instead of old school DTD files for project configuration files, TeamCity now uses the XML schema: <http://www.jetbrains.com/teamcity/schema/9.0/project-config.xsd>
- NuGet feed supports API v2. The feed performance should be much better.
- The ability to specify which fields to include into REST API responses via the `fields` parameter graduated from [experimental state](#) (was introduced in 8.1)
- LDAP error reporting into the `teamcity-ldap` log was improved a lot and now provides more guidance on fixing incorrect settings
- Unicode Support for MS SQL and Oracle: the national character sets (`nchar`, `nvarchar`, `nclob` types) for text fields are now supported in MS SQL and Oracle databases used by TeamCity. You can now use Unicode characters to store the user input and data from external

systems, like VCS, NTLM, etc.

Fixed Issues

- full list of fixed issues including 40 performance fixes

Previous Releases

[What's New in TeamCity 8.1](#)

Concepts

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

- [Agent Cloud Profile](#)
- [Agent Home Directory](#)
- [Agent Requirements](#)
- [Agent Work Directory](#)
- [Already Fixed In](#)
- [Authentication Modules](#)
- [Build Agent](#)

- Build Artifact
- Build Chain
- Build Checkout Directory
- Build Configuration
- Build Configuration Template
- Build Grid
- Build History
- Build Log
- Build Number
- Build Queue
- Build Runner
- Build State
- Build Tag
- Build Working Directory
- Change
- Change State
- Clean Checkout
- Clean-Up
- Code Coverage
- Code Duplicates
- Code Inspection
- Continuous Integration
- Dependent Build
- Difference Viewer
- Favorite Build
- First Failure
- Guest User
- History Build
- Identifier
- Notifier
- Personal Build
- Pinned Build
- Pre-Tested (Delayed) Commit
- Project
- Remote Debug
- Remote Run
- Revision
- Role and Permission
- Run Configuration Policy
- Super User
- TeamCity Data Directory
- TeamCity Home Directory
- TeamCity Specific Directories
- User Account
- User Group
- VCS root
- Wildcards

Build Artifact

TeamCity contains an integrated lightweight builds artifact repository.

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

To download artifacts of a build, use the **Artifacts** column of the build entry on the TeamCity pages that list the builds, or locate 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 storage format is described in the [TeamCity Data Directory#artifacts](#) section. The artifacts are stored on the server "as is" without additional compression, etc. By default, the artifacts are stored under the `<TeamCity data directory>/system/artifacts` directory which [can be changed since TeamCity 9.1](#).

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

Hidden Artifacts

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

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

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

Artifacts Cache on Agent

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

See also:

[Concepts: Dependent Build](#)

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

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](#).

The recommended way is not to merge builds which fall into different sets into a single build configuration, but rather have a separate build configuration for each build sequence (that is performing a specified task in a dedicated environment). This allows for proper features functioning, like detection of new problems/failed tests, first failed in/fixes in tests status, automatically removed investigations, etc.

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

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 [Changes](#) page.

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**.

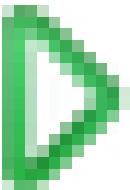
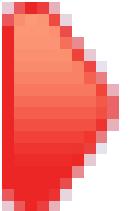
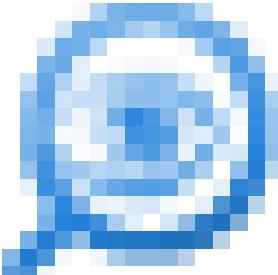
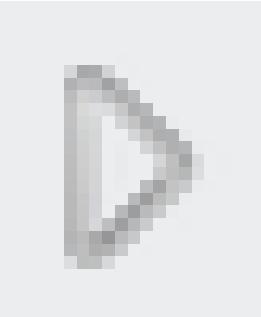
Build Configuration Status

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 on default branch executed successfully.
	The last build on default branch executed with errors or one of the currently running builds is failing. The build configuration status will change to "failed" when there's at least one 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).
no icon	<p>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</p>
	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 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.

On this page:

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

Viewing Build Log

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

The **Tree view** is the most capable view provided in the web UI. By default, all messages are displayed. Using the View drop-down, you can 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.

You can download a full build log in the textual form from the Build Results page by clicking  [Download full build log](#). Alternatively, you can use the following URL:

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

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

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

Build Log Size

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

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

ANSI-style Coloring in Build Log

Since TeamCity 9.1, TeamCity build logs render clickable hyperlinks and support ANSI-style escape color codes by default: if your tool produces a colored console output, you will see in the build log in TeamCity. See the related feature request for the [full list of supported sequences](#).

To disable the coloring, set the `teamcity.buildLog.ansiColoring.enabled=false` [internal property](#).

Build Queue

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

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

On this page:

- [Build Queue Optimization by TeamCity](#)

- Build Queue Tab
 - Agent Selection for Queued Build
 - Ordering Build Queue
 - Pausing/Resuming Build Queue

Build Queue Optimization by TeamCity

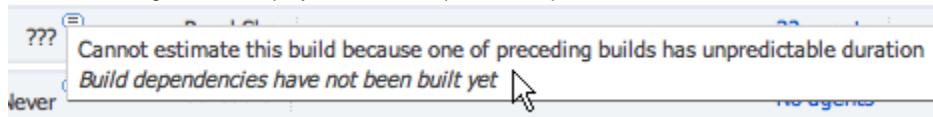
TeamCity optimizes the build queue as follows:

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

Build Queue Tab

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

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



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

Agent Selection for Queued Build

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

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

Ordering Build Queue

You can do the following:

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

Pausing/Resuming Build Queue

The build queue can be paused manually or automatically.

Users with 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.

See also:

Concepts: Build Chain

Administrator's Guide: Ordering Build Queue

Build State

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

Build States

Icon	State	Description
	running successfully	A build is running successfully.
	successful	A build finished successfully in all specified build configurations.
	running and failing	A build is failing.
	failed	A build failed at least in one specified build configuration.
	cancelled	A build was cancelled.

Canceled/Stopped build

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

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



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

Personal Build States

Icon	State	Description
	running successfully	A personal build is running successfully.
	successful	A personal build has completed successfully for all specified build configurations.
	running and failing	A personal build is running with errors.
	failed	A personal build failed at least in one specified build configuration.

Hanging and Outdated Builds

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

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

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

Failed to Start Builds

Builds which 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 a specific tag
- search for a build with a particular tag
- create an artifact dependency on a build with a particular tag

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

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

- using the [Build Configuration Home Page](#):
 - go to either the **Overview** or **History** tab
 - go to the build history table
 - click the down-arrow button in the **Tags** column for the desired build
 - click the **Edit** link in the drop-down list and add/modify the build tag.
- using the [Build Results Page](#) for the particular build:
 - click the **Actions** button
 - select **Tag...** and add/modify the build tag
- using the [Queued build page](#):
 - click the **Actions** button
 - select **Tag...** and add/modify the build tag
- using the [Run Custom Build](#) dialog
 - go to the **Comments and Tags** tab and add/modify the build tag

By clicking a tag, you filter out all of the builds in the history: only the builds marked with the tag are displayed.

Additionally you can search for builds with particular tags using the search field.

Change

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

TeamCity suggests several ways to view changes:

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

Viewing and analyzing changes involves the following possibilities:

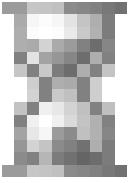
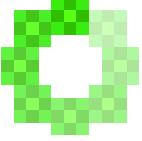
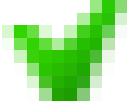
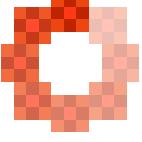
- Observing actual changes that are already included in the build using the [Changes](#) link on the [Projects](#) page or on the [Changes](#) tab of the [build results](#).
- Observing pending changes in the [Pending Changes](#) tab of the the Build Configuration Home Page.
- Viewing the [revision](#)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

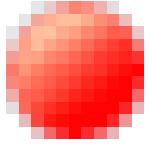
See also:

[Concepts: Revision, Build Configuration](#)

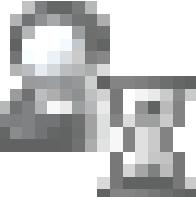
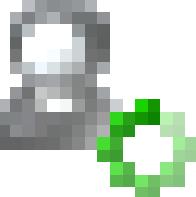
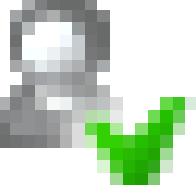
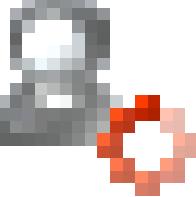
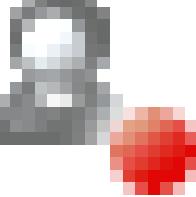
[User's Guide: Investigating Build Problems](#)

Change State

Icon	State	Description
	pending	<p>The change is scheduled to be integrated into a build that is currently in the build queue.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">i Even if a change has been successfully integrated into a build, the change will appear as pending when it is scheduled to be added to a different build.</div>
	running successfully	<p>The change is being integrated into a build that is running successfully.</p>
	successful	<p>The change was integrated into build that finished successfully.</p>
	running and failing	<p>The change is being integrated into a build that is failing.</p>

	failed	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:

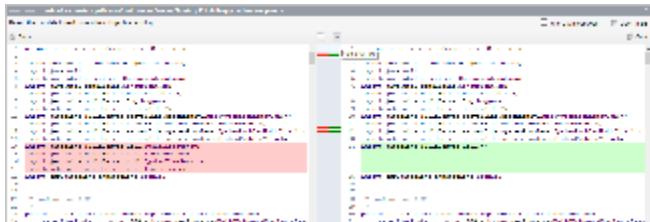
[Concepts: Build Configuration Status | Build State | Change](#)
[User's Guide: Viewing Your Changes](#)

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](#)

Project

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

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

In TeamCity, user roles and permissions are managed on per-project basis.

On this page:

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

Project Hierarchy

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

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

Settings Propagation

The projects hierarchy is used in the following ways:

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

- [Parameters](#)
- [Clean-up rules](#)
- [Since TeamCity 9.0, Settings for synchronizing the project settings with version control](#)

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

- [VCS Roots](#)
- [Build configuration template](#)
- [Issue Trackers \(since TeamCity 9.0\)](#)
- [Shared Resources](#)
- [SSH keys](#)
- [Maven Settings](#)
- [Meta-Runners](#)

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

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

- [User and User group roles](#)

- Investigations
- Muted Problems
- Notification rules

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

Root Project

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

The root project is special in the following ways:

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

See also:

[Concepts: Build Configuration](#)

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

Agent Home Directory

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

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

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

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

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

Agent Directories

The agent consists of:

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

Agent Files Modification

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

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

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

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

Important Agent Files and Directories

- **/bin**
 - `agent.bat` — batch script to start/stop the build agent from the console under Windows
 - `agent.sh` — shell script to start/stop the build agent under Linux/Unix
 - `service.install.bat` — batch file to install the build agent as a Windows service. See also [related section](#).
 - `service.start.bat` — starts the build agent using the installed build agent service
 - `service.stop.bat` — stops the installed build agent service
 - `service.uninstall.bat` — batch file to uninstall the currently installed build agent Windows service

- **/conf** — this folder contains all configuration files for the build agent
 - `buildAgent.properties` — [main configuration file](#). This file is generated by the TeamCity server .exe installer and build agent .exe installer.
 - `buildAgent.dist.properties` — sample configuration file. You can rename it to 'buildAgent.properties' to create an initial agent configuration file.
 - `teamcity-agent-log4j.xml` — build agent logging settings. For details, please refer to comments inside the file or to the [log 4j manual](#)
- **/launcher/conf/**
 - `wrapper.conf.template` — sample configuration file to be used as a template for creating the original configuration
 - `wrapper.conf` — current build agent Windows service configuration. This is a Java Service Wrapper configuration java properties file. For details, please see comments inside the file or Java Service Wrapper documentation.
- **/logs**
 - `launcher.log` — log of the build agent launcher
 - `teamcity-agent.log` — main build agent log
 - `wrapper.log` — log of the Java Service Wrapper. Available only if the build agent is running as a windows service
 - `teamcity-build.log` — log from the build
 - `upgrade.log` — log from the build agent upgrade process
 - `teamcity-vcs.log` — agent-side checkout logs
- **/system**
 - `.artifacts_cache` — cache for all build's artifacts; can be [configured](#)
- **/temp** — temporary folder; the path can be overridden in the `buildAgent.properties` file
 - `agentTmp` — temporary folder that is used by the build agent to store build-related files during the build. Is cleaned after each build.
 - `buildTmp` — temporary folder that is set as the default temp directory for the build process and is cleaned after each build
 - `globalTmp` — temporary folder that is used by the build agent for its own temporary files. Is cleaned on the agent restart.

Agent Requirements

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

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

The administrator can specify required environment variables and system properties for a build configuration on the build configuration'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 `teamcity.agent.jvm.os.name` system property on the build agent must contain the `Windows` string.

If the properties and environment variables on the build agent do not fulfill the requirements specified by the build configuration, then the build agent is incompatible with this build configuration. The Agent Requirements page lists both compatible and incompatible agents.

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

Implicit Requirements

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

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

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

See also:

[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](#)

Authentication Modules

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

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

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

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

TeamCity supports the following *credentials authentication modules*:

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

The following *HTTP authentication modules* are supported:

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

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

See also:

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

Build 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).

On this page:

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

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, Build Configuration restrictions configured for an agent and the selection algorithm described [here](#).

Build Agent Status

In TeamCity, a build agent can have following statuses:

Status	Description
Connected/ Disconnected	An agent is connected if it is registered on the TeamCity server and responds to server commands, otherwise it is disconnected. This status is determined automatically.

Authorized/ Unauthorized	Agents are manually authorized via the web UI on the Agents page. 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	Agents are manually enabled/disabled via the web UI . The TeamCity server only distributes builds to agents that are enabled. Info Agent disabling does not affect (stop) the build which is currently running on the agent. 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.

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 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".

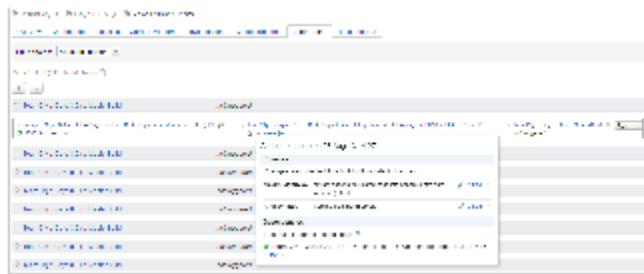
In case when all other parts of the build chain cannot be viewed by the current user, we show a yellow stripe with warning: **You don't have access rights to see its other parts**.

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 the build configuration, you can get the directory from the build log, or you can refer to the `<Agent Work Directory>/directory.map` generated file which lists build configurations with the directories they used last.

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

 By default, this is also the directory [where builds will run](#).

Checkout Process

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

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

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

That is why it is recommended to:

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

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

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 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.

On this page:

- How can I create build configuration template?
- Associating build configurations with templates
- Detaching build configurations from template
- Redefining settings inherited from template
 - Modifying Settings
 - Using parameter reference
 - Example of configuration parameters usage



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.

If you upgraded from versions earlier than TeamCity 8.0, your configurations from one project might be associated with templates from an unrelated project. After upgrading to TeamCity 8+, such templates can become inaccessible in out-of-order projects during the settings editing or new build configuration creation. To reuse build configuration templates from an unrelated project, it is recommended to manually move the templates 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 is the **Extract template** option available from the **Actions** button at top right corner of the screen. Note that if you extract a template from a build configuration, the original configuration automatically becomes [associated with](#) the newly created template.

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 a template: there's the **Associate with Template** option available from the **Actions** button at top right corner of the screen.

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

You can associate a build configuration to a template only if the template belongs to the current project or one of its parents.

A template which has at least one associated build configuration cannot be deleted, the associated build configurations need to be detached first.

Detaching build configurations from template

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

Redefining settings inherited from template

Although a build configuration associated with a template inherits all its settings (marked as *inherited* in the UI), it is still possible to redefine a number settings in an associated build configuration. Modifying settings in the template will influence **all configurations** associated with this template.

 Inherited settings cannot be deleted from an associated build configuration.

The following settings can be overridden in a build configuration inherited from a template:

- build number format
- artifact paths
- build options (hanging builds detection, triggering personal builds, 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
- **Parameters**
- **Agent Requirements**
- **Snapshot Dependencies**

 Note that if you rename an inherited parameter or an agent requirement in a build configuration, this parameter/ agent requirement is actually disconnected from the template and is treated as a new parameter created in the build configuration. For example, if you rename a parameter in an associated build configuration, and then rename the same parameter in the template, you will end up with two parameters in the build configuration: the new one inherited from the template, and the old one which was redefined in the build configuration.

Besides, you can redefine settings configured via parameter references or add new items to lists.

Modifying Settings

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 the 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 a 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 you can reorder the collection elements in the inherited configuration, for example, build steps.

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

Using parameter reference

To introduce a configuration parameter reference, use the `%ParameterName%` syntax in the template text field. Once introduced, this parameter appears on the **Parameters** page of the build configuration template marked as requiring a value.

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

See also [Configuring Build Parameters](#) and [Defining and Using Build Parameters in Build Configuration#Using Build Parameters in Build Configuration Settings](#).

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 this means changing the checkout rules in each build configuration separately.

To do so, perform the following steps:

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

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

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

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 Number

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

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

A build number can be:

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

See also:

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

Build Runner

Build runner is a part of TeamCity that allows integration with a specific build tool (Ant, MSBuild, Command line, etc.). In a build configuration, 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 its earlier version: [Ipr \(deprecated\)](#)
- Maven
- MSBuild
- MSpec
- MSTest
- NAnt
- NuGet-related runners
- NUnit
- PowerShell
- Rake
- Visual Studio (sln) — Microsoft Visual Studio 2005/2008/2010/2012/2013/2015 solutions
- Visual Studio 2003 — Microsoft Visual Studio 2003 solutions
- Xcode Project

Technically, build runners are implemented as plugins.

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

See also:

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

Build Working Directory

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

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

 Not all build runners provide the working directory setting.

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

See also:

[Concepts: Build Checkout Directory](#)

Clean Checkout

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

Enforcing Clean Checkout

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

You can enforce clean sources action for a build configuration from the Build Configuration home page (**Actions** drop-down in the top right corner), or for an agent from the [Agent Details](#) page. 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 in the checkout directory before the 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.

Since TeamCity 9.0, project-related Clean-Up settings are configured in the Project Settings, not in the Server [Administration|Clean-up Settings](#) section as before. The global server settings are available for general Clean-Up configuration.

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

Clean-up deletes the data stored under [TeamCity Data Directory](#)/system and in the database. Also, during the clean-up time the server

performs various maintenance tasks (e.g. resets VCS full patch caches).

On this page:

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

Server Clean-up Settings

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

Since TeamCity 9.0, the build history clean-up is run as a background process, which means that now there is no server maintenance down-time.

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

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

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

Manual Clean-up Launch

You can run clean-up manually in the **Start clean-up** section of the Clean-up Settings. 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.

Project Clean-up Rules

Clean-up rules are configured per project and define when and what data to clean. Different rules can be assigned to a project and templates or build configurations within this project.

To manage the rules, use the [Project-Settings | Clean-up Rules](#) page.

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 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.

Clean-up of Personal Builds

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

Deleted Build Configurations Cleanup

When a project or a build configuration is deleted, the corresponding builds data is removed during the cleanup, but only if 24 hours has passed since the deletion. To change the timeout, set the `teamcity.deletedBuildTypes.cleanupTimeout` internal property to the required number of seconds to protect the data from deletion.

See also:

Concepts: Dependent Build

Code Coverage

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

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

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

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

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

- Ant
- IntelliJ IDEA Project
- Maven
- MSBuild
- NAnt
- NUnit
- MSpec
- MSTest
- .NET Process Runner
- 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 and a related [blog post](#).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Artifact Dependency

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

To create and configure an artifact dependency, use the **Dependencies** build configuration settings page. If 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](#)

Guest User

TeamCity allows you to turn on guest login which allows anonymous access to the TeamCity web UI.

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

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

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

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

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

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

You can use guest login to download artifacts:

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

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

See also:

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

History Build

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

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

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

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

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

See also:

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

[Administrator's Guide: Triggering a Custom Build](#)

Notifier

TeamCity supports the following notifiers:

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

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

See also:

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

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

Personal Build

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

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

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

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

One can also mark a build as personal using the corresponding option of the [Run...](#) dialog.

By default, only users with the [Project Developer](#) role can initiate a personal build.

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

See also:

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

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

[Troubleshooting: Remote Run Problems](#)

Pinned Build

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

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.

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

Developers test their changes by performing a remote run. A pre-tested commit is enabled when the **commit changes if successful** option is selected.

The pre-tested commit is initiated via a plugin to one of supported IDEs (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.

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 the TeamCity server.

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

This includes:

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

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

General Flow of a pre-tested commit

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

See also:

[Concepts: Remote Run](#)

[Remote Run on Branch: Remote Run on Branch Build Trigger for Git and Mercurial](#)

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

Remote Run

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

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

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

See also:

Concepts: [Pre-tested Commit](#) | [Personal Build](#)

Remote Run on Branch: [Remote Run on Branch](#) [Build Trigger for Git and Mercurial](#)

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

Troubleshooting: [Remote Run Problems](#)

Role and Permission

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

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

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

On this page:

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

Authorization Mode

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

In the **simple** mode, there are only three types of authorization levels: guest, logged-in user and administrator.

In the **per-project** mode, you can assign users *Roles* in projects or server-wide. The set of permissions in roles is editable.

Permissions within a role granted at the project level are automatically propagated in all the subprojects of this project.

The **View project and all parent projects** permission allows you to view not only the project (with its subprojects) but its parent projects too.

Changing Authorization Mode

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

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

Simple Authorization Mode

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

Per-Project Authorization Mode

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

By default, TeamCity provides the following roles:

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 view 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](#)

TeamCity Data Directory

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

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

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

On this page:

- Location of the TeamCity Data Directory
 - Recommendations as to choosing Data Directory Location
- Structure of TeamCity Data Directory
- Direct Modifications of Configuration Files
 - `.dist` Template Configuration Files
 - XML Structure and References

Location of the TeamCity Data Directory

Since TeamCity 9.1 the location of the directory can be set on the first server startup screens (only when TeamCity .tar.gz or .exe distribution is used). The specified data directory is then saved into <TeamCity home directory>/conf/teamcity-startup.properties file.

The location of the directory can be set manually using TEAMCITY_DATA_PATH environment variable. The variable can be either system-wide or defined for the user under whom TeamCity server is started. After setting/changing the variable, you might need to restart the computer for the changes to take effect.

If the TEAMCITY_DATA_PATH environment variable is not set and <TeamCity home directory>/conf/teamcity-startup.properties file does not define it either, the TeamCity Data Directory is assumed to be located in the user's home directory (e.g. it is \$HOME/.BuildServer under Linux and %USERPROFILE%\BuildServer under Windows).

Before TeamCity 9.1, the TeamCity Windows installer configured the TeamCity data directory during installation by setting the TEAMCITY_DATA_PATH environment variable. The default path suggested for the directory is: %ALLUSERSPROFILE%\JetBrains\TeamCity.

Since TeamCity 9.1, installer does not ask for the TeamCity data directory and it can be configured on the first TeamCity start.

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

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

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

Recommendations as to choosing Data Directory Location

Note that by default the `system` directory stores all the [artifacts](#) and build logs of the builds in the history and can be quite large, so it is recommended to place TeamCity Data Directory on a non-system disk. Refer to [Clean-Up](#) section to configure automatic cleaning of older builds.

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

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 file system links. **Since TeamCity 9.1 EAP2**, you can configure [multiple artifacts paths](#).

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. or details on restoring the projects check [How To...#Restore Just Deleted Project](#)
 - `_notifications` — notification templates and notification configuration settings, including syndication feeds template
 - `_logging` — [internal server logging](#) configuration files, new files can be added to the directory
 - `projects` — a directory which contains all project-related settings. Each project has its own directory. Project hierarchy is not used and all the projects have a corresponding directory residing directly under "projects"
 - `<projectId>` - a directory containing all the settings of a project with the "`<projectId>`" id (including build configuration settings and excluding subproject settings). New directories can be created provided they have mandatory nested files. The `_Root` directory contains settings of the `root` project. Whenever `*.xml.N` files occur under the directory, they are backup copies of corresponding files created when a project configuration is changed via the web UI. These backup copies are not used by TeamCity.
 - `buildNumbers` — a directory which contains `<buildConfigurationID>.buildNumbers.properties` files which store the current build number counter for the corresponding build configuration
 - `buildTypes` — a directory with `<buildConfiguration or template ID>.xml` files with corresponding build configuration or template settings
 - `pluginData` — a directory to store optional and plugin-related project-level settings. Bundled plugins settings and auxiliary project settings like custom project tabs are stored in `plugin-settings.xml` file in the directory
 - `vcsRoots` — a directory which contains project's VCS roots settings in the files `_<VcsRootID>.xml`
 - `project-config.xml` — the project configuration file containing the project settings, such as `parameters` and `clean-up rules`.
- `main-config.xml` — server-wide configuration settings
- `database.properties` — database connection settings, see more at [Setting up an External Database](#)
- `license.keys` — a file which stores the license keys entered into TeamCity
- `change-viewers.properties` — [External Changes Viewer](#) configuration properties, if available
- `internal.properties` — file for specifying various [internal TeamCity properties](#). It is **not** present by default and needs to be created if necessary

- `auth-config.xml` — a file storing server-wide authentication-related settings
- `ldap-config.properties` — [LDAP authentication](#) configuration properties
- `ntlm-config.properties` — [Windows domain authentication](#) configuration properties
- `issue-tracker.xml` — issue tracker integration settings
- `cloud-profiles.xml` — Cloud (e.g. Amazon EC2) integration settings
- `backup-config.xml` — web UI backup configuration settings
- `roles-config.xml` — roles-permissions assignment file
- `database.*.properties` — default template connection settings files for different external databases
- `*.dtd` — DTD files for the XML configuration files
- `*.dist` — default template configuration files for the corresponding files without `.dist`. See [below](#).
- **`.BuildServer/plugins`** — a directory where TeamCity plugins can be stored to be loaded automatically on the TeamCity start. New plugins can be added to the directory. Existing ones can be removed while the server is not running. The structure of a plugin is described in [Plugins Packaging](#).
 - `.unpacked` — directory that is created automatically to store unpacked server-side plugins. Should not be modified while the server is running. Can be safely deleted if the server is not running.
 - `.tools` — create this directory to centralize tools to be installed on all agents. Any folder or .zip file under this folder will be distributed to all agents and appear under `<agent root>/tools` folder.
- **`.BuildServer/system`** — a directory where build results data is stored. The content of the directory is generated by TeamCity and is not meant for manual editing.
 - `artifacts` — the [default directory](#) where the builds' artifacts, logs and other data are stored. The format of the artifact storage is `<project ID>/<build configuration name>/<internal build id>`. If necessary, the files in each build's directory can be added/removed manually - this will be reflected in the corresponding build's artifacts.
 - `.teamcity` subdirectory stores build's [hidden artifacts](#) and build logs (see below). The files can be deleted manually, if necessary, but the build will lack the corresponding feature backed by the files (like the build log, displaying/using finished build parameters, coverage reports, etc.)
 - `/logs` subdirectory stores [build logs](#) in an internal format. Build logs store the build output, compilation errors, test output and test failure details. The files can be removed manually, if necessary, but corresponding builds will drop build log and failure details (as well as test failure details).
 - `messages` — a directory where build logs used to be stored before TeamCity 9.0. After automatic build logs migration to the new place under artifacts, the directory stores the files which could not be moved (see server log on the server start about details).
 - `changes` — a directory where the [remote run](#) changes are stored in internal format. Name of the files inside the directory contains internal personal change id. The files can be deleted manually, if necessary, but corresponding personal builds will lose personal changes in UI and when affected queued builds try to start, they fail or run without personal patch.
 - `pluginData` — a directory where various plugins can store their data. It is not advised to delete or modify the directory. (e.g. state of build triggers is stored in the directory)
 - `audit` — directory holding history of the build configuration changes and used to display diff of the changes. Also stores related data in the database.
 - `repositoryStates` — directory holding current state of the VCS roots. If dropped, some changes might not be detected by TeamCity (between the state last queried by TeamCity and the current state after first server start without this data).
 - `caches` — a directory with internal caches (of the VCS repository contents, search index, other). It can be [manually deleted](#) to clear caches: they will be restored automatically as needed. It is safer to delete the directory while server is not running.
 - `buildserver.*` — a set of files pertaining to the embedded HSQLDB.
- **`.BuildServer/backup`** — default directory to store backup archives created via [web UI](#). The files in this directory are not used by TeamCity and can be safely removed if they were already copied for safekeeping.
- **`.BuildServer/lib/jdbc`** — directory that TeamCity uses to search for [database drivers](#). Create the directory if necessary. TeamCity does not manage the files in the directory, it only scans it for .jar files that store the necessary driver.

Direct Modifications of Configuration Files

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

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

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

.dist Template Configuration Files

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

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

XML Structure and References

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

See also 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 [rol](#)

es 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](#)

Wildcards

TeamCity supports wildcards in different configuration options.

Ant-like wildcards are:

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

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

Examples

For a directory structure:

```

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

```

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

Already Fixed In

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

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

"After" means here that

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

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

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

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

See also:

[Concepts: First Failure](#)

First Failure

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

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

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

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. [Since TeamCity 9.0](#), the way TeamCity counts tests [has changed](#).

See also:

[Concepts: Already Fixed In](#)

Super User

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

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 the "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 a login attempt without a username specified, the token is printed into the log again.

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

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

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

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

See also:

[Concepts: Guest User](#)

Identifier

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

Each entity has two identifiers:

- a so-called *external ID*
- a Universally Unique ID (since TeamCity 9.0)

On this page:

- External IDs
 - Using IDs
 - Assigning IDs
- Universally Unique IDs

External IDs

External ids are configured in the TeamCity web UI and must be unique within all the objects of the same type on the entire server; note that build configurations and templates share the same ID space.

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

Using IDs

External IDs are used:

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

Assigning IDs

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

The ID can be modified manually.

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

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



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

If any of the URLs containing the old IDs were bookmarked or hard-coded in the scripts, they will stop to function and will need update. At the moment of the ID change, the correspondingly named directories under TeamCity Data Directory (including directories storing settings and artifacts) are renamed and this can take time.

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

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

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

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

Universally Unique IDs

Since TeamCity 9.0, projects, build configurations and VCS roots have a UUID, an automatically generated, globally unique ID. UUID is stored in the corresponding entity XML configuration file located in the <TeamCity data directory>/config directory. These UUID should never be edited manually. When a new entity is created by placing a file in the data directory, it should have no "uuid" attribute. In this case TeamCity will generate it automatically and will persist in the file.

See also:

Concepts: Project | Build Configuration

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

VCS root

VCS Roots in TeamCity

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

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

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

See also:

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

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

Remote Debug

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



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:

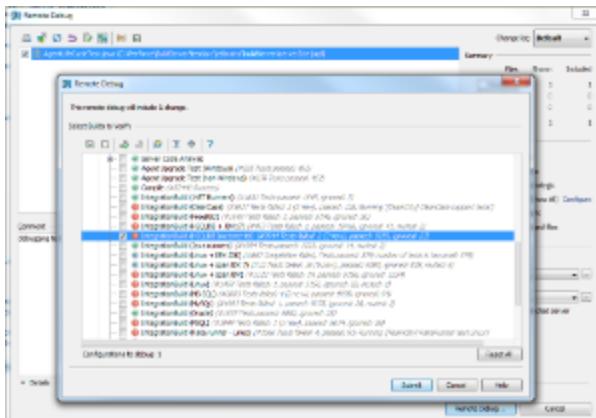
- Java application run configuration
- JUnit run configuration
- TestNG run configuration

Prerequisites:

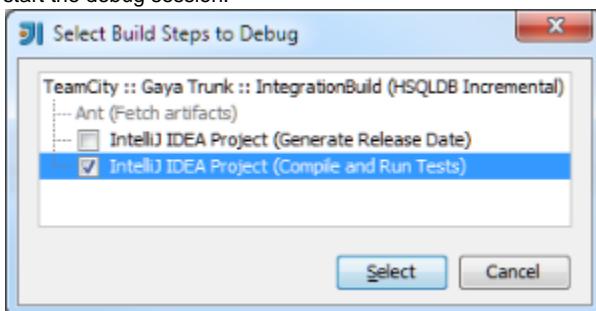
1. an IntelliJ IDEA run configuration on the local developer machine with the [TeamCity plugin for IntelliJ IDEA](#) installed,
2. a build configuration on the TeamCity Server with the [IntelliJ IDEA Project](#) runner as one of the [build steps](#)
3. a remote [TeamCity agent](#) to run this build available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

Favorite Build

Since TeamCity 9.0. to easily access builds you want to monitor, you can mark them as favorite.

On this page:

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

Adding Builds to Favorites

To manually add a build to your favorites, use the appropriate option from the **Actions** menu on the **Build results** page. Your favorite build will be

marked with a star symbol .

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

Since **TeamCity 9.1**, you can add a custom build to favorites by checking the corresponding box.

Viewing Your Favorites

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

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

Configuring Notifications on Your Favorite Builds

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

Removing Builds from Favorites

To remove builds from favorites:

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

Agent Cloud Profile

Configuring a cloud provider profile is one of the steps required to [enable agent cloud integration](#) between TeamCity and a cloud provider.

A *cloud profile* is a collection of settings for TeamCity to start virtual machines with installed TeamCity agents on-demand while distributing a build queue.

Profiles are configured on the **TeamCity | Administration | Agent Cloud** page. The TeamCity server URL must be accessible to the build agent.

The settings of profiles slightly vary depending on the integrated cloud type.

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

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

TeamCity Home Directory

The *TeamCity Home Directory* or the *TeamCity Installation Directory* is the directory where the TeamCity server application files and libraries have been unpacked when TeamCity was [installed](#).

The location of the TeamCity Home directory is defined when you install the TeamCity server. The default directory suggested by the Windows [installation package](#) is `C:\Program Files (x86)\JetBrains\TeamCity`; however, TeamCity can be installed into any directory.

Important Files and Directories

`TeamCity-readme.txt` – the directory description

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

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

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

- **/work** - standard Tomcat folder where Tomcat writes cache files for every page it serves.

Revision

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

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

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

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

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

See also:

[Concepts: Change, Build Configuration](#)

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

Supported Platforms and Environments

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

In this section:

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

Platforms (Operating Systems)

TeamCity Server

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

Requirements:

- Java (JRE)

- Oracle Java 6-8, (Java 7, and **since TeamCity 9.1** Java 8, is included in the Windows .exe distribution). **The latest Oracle 1.7 JDK, and since TeamCity 9.1 JDK 1.8, is recommended.**
- OpenJDK 7 is supported, but Oracle JDK is recommended.
- Both 32-bit and 64-bit Java versions can be [used](#).
- For .war distribution:
J2EE Servlet (2.5+) container, JSP 2.0+ container based on Apache Jasper. TeamCity is tested under Apache 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.
 - due to [bug](#) in Tomcat 7.0.25 and [another one](#) in Tomcat 7.0.54 these versions cannot be used with TeamCity
 - 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 6 Update 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**. If you find any compatibility issues with any of the operating systems please make sure to [let us know](#).

The TeamCity server is tested under the following operating systems:

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

Reportedly works without known issues 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)
 - Oracle Java 6-8, (Java 7, and **since TeamCity 9.1** Java 8, is included in the Windows .exe distribution). **The latest Oracle 1.7 JDK, and since TeamCity 9.1 Oracle JDK 1.8, is recommended.**
 - OpenJDK 7 is supported, but Oracle JDK is recommended.
 - Both 32-bit and 64-bit Java versions can be [used](#).

TeamCity agent is tested under the following operating systems:

- Linux
- MacOS X
- Windows XP/XP x64/Vista/Vista x64
- Windows 7/7x64
- 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 8+ (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 1.8.4, **since TeamCity 9.1** - with Ant 1.9.6.)
- [Maven](#) versions 2.0.x, 2.x, 3.x (known at the moment of the TeamCity release). Java 1.5 and higher is supported. TeamCity comes bundled with Maven 2.2.1, 3.0.5, 3.1, 3.2.3 and **since TeamCity 9.0.4** Maven 3.3.1.
- [IntelliJ IDEA Project runner](#) (requires Java 1.6+)
- [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. Microsoft Build Tools 2013 and **since TeamCity 9.1** Microsoft Build Tools 2015 preview 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, 2013, and **since TeamCity 9.1** 2015) (requires a corresponding version of Microsoft Visual Studio installed on the build agent).
- [FxCop](#) (requires FxCop installed on the build agent)
- [Duplicates Finder for C# and VB.NET code](#) based on [ReSharper Command Line Tools](#). Supported languages are C# up to version 4.0 and VB.NET version 8.0 - 10.0. Requires .Net Framework 4.0+ installed on the build agent.
- [Inspections for .NET](#) based on [ReSharper Command Line Tools](#). Requires .Net Framework 4.0+ installed on the build agent.
- [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](#) versions 1.0 through 4.0 are supported. **Since TeamCity 9.1** version 5.0 is supported as well.
- [.NET Process Runner](#) for running any .NET application (requires .NET installed on the build agent)
- [Xcode](#) 3, 4, 5, 6 are supported. **Since TeamCity 9.1**, [Xcode 7](#) 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.x (dedicated build runner)
- [TestNG](#) 5.3+
- [MSTest](#) 8.x, 9.x, 10.x, 11.x, 12.x.; **since TeamCity 9.1** MSTest 2015 (14.0) is also supported. The dedicated build runner; requires appropriate Microsoft Visual Studio edition installed on the build agent. **Since TeamCity 9.1 EAP3** the MSTest runner is merged into the [Visual Studio Tests](#) runner.
- [Visual Studio Tests](#) runner, available **since TeamCity 9.1**, integrates MSTest runner and VSTest console runner formerly provided as an

- external plugin.
- [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). Check [compatibility issues](#).
- 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, 2013. **Since TeamCity 9.1**, TFS 2015 is supported as well. Requires Team Explorer installed on the TeamCity server.
- CVS
- 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 1.4-1.8 format are supported)
- CVS
- 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, 2010, 2012, 2013, and **since TeamCity 9.1** 2015. (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 reportedly 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 3.6.2-3.8 and 4.2-4.4 are supported. Eclipse must be run under JDK 1.5+
- **IntelliJ Platform Plugin:** compatible with IntelliJ IDEA 11.1.x - 14.x (Ultimate and Community editions) and other IDEs based on the same version of the platform, including JetBrains RubyMine 4.0+, JetBrains PyCharm 2.5+, JetBrains PhpStorm/WebStorm 4.0+, AppCode 1.5+
- **Microsoft Visual Studio 2005, 2008, 2010, 2012, 2013.** Since TeamCity 9.1 Visual Studio 2015 is supported by the TeamCity Visual Studio Add-in shipped as a part of ReSharper Ultimate. Installed .NET Framework is required.

Remote Run and Pre-tested Commit

Remote Run and Pre-tested commit functionality is available for the following IDEs and version control systems:

IDE	Supported VCS
Eclipse	<ul style="list-style-type: none">• Subversion 1.4-1.7 with Subclipse and Subversive Eclipse integration plugins.• Subversion 1.7-1.8 via Subversive/Subclipse/SvnKit.• Perforce (P4WSAD 2008.1 - 2010.1, P4Eclipse 2010.1 - 2014.1)• ClearCase (the client software is required)• CVS• Git (the EGit 1.0+ Eclipse integration plugin) see also
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. 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)  The internal database suits evaluation purposes only; we strongly recommend using an external database in a production environment.
- MySQL 5.0.33+, 5.1.49+, 5.5+, 5.6+ (Please note that due to bugs in MySQL, versions 5.0.20, 5.0.22 and 5.1 up to 5.1.48 are not compatible with TeamCity)
- Microsoft SQL Server 2005, 2008, 2012, and 2014 (including Express editions), SQL Azure; SSL connections support might require [these updates](#).
- PostgreSQL 8+
- Oracle 10g+ (TeamCity is tested with [driver](#) version 10.2.0.1.0)

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 [Maven Surefire plugin](#) or [Maven Failsafe plugin](#); tests reporting occurs after each module test run finish)
 - [IntelliJ IDEA](#) project (when run with appropriate IDEA run configurations)
- NUnit for the following runners:
 - The NAnt (`nunit2` task)
 - The MSBuild ([NUnit community](#) or [NUnitTeamCity](#) tasks)
 - Microsoft Visual Studio Solution runners (2003, 2005, 2008, 2010, 2012, 2013).
 - Any runner provided [TeamCity Addin for NUnit](#) is installed.
- MSTest 2005, 2008, 2010, 2012, 2013 (On-the-fly reporting is not available due to MSTest limitations)
- MSpec

Ruby testing frameworks, [Test::Unit](#), [Test-Spec](#), [Shoulda](#), [RSpec](#), [Cucumber](#), are supported by the TeamCity [Rake](#) runner. The [minitest](#) framework requires the `minitest-reporters` gem to be additionally installed.

There are also testing frameworks that have embedded support for TeamCity. e.g. [Gallio](#) and [xUnit](#).
See also 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 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. See the list of [supported reports](#).

Including HTML Reports

If your reporting tool is not supported by TeamCity directly, you can make it produce reports in the HTML format via a build script and add a build results [report tab in TeamCity](#).

Importing Code Coverage Results

You can also [import code coverage results in TeamCity](#).

Integration with External Tools

TeamCity can also be integrated with external build tools or tools generating some report/providing code metrics which are not yet supported by TeamCity. The integration tasks involved are collecting the data in the scope of a build and then reporting the data to TeamCity.

Installation and Upgrade

In this part you will learn how to install and upgrade TeamCity.

- [Installation](#)
- [Upgrade Notes](#)
- [Upgrade](#)
- [TeamCity Maintenance Mode](#)
- [Setting up an External Database](#)
- [Migrating to an External Database](#)

Installation

If you are upgrading your existing TeamCity installation, please refer to [Upgrade](#).

Check the System Requirements

Before you install TeamCity, please familiarize yourself with [Supported Platforms and Environments](#).

Additionally, read the [hardware requirements for TeamCity](#). However, note that these requirements differ significantly depending on the server load and the number of builds run.

Select TeamCity Installation Package

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. Since TeamCity 9.1 , Java 1.8 is used.
Linux, Mac OS X	TeamCity<version number>.tar.gz	Archive for manual installation bundled with Tomcat servlet container
J2EE container	TeamCity<version number>.war	Package for installation into an existing J2EE container.

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 build agents on separate machines or ensure that there is enough CPU/memory/disk throughput on the server machine.

To setup additional build agents, follow the [instructions](#).

See also:

[Installation and Upgrade: Installing and Configuring the TeamCity Server | Setting up and Running Additional Build Agents](#)

Installing and Configuring the TeamCity Server

This page covers a new TeamCity server installation.

For **upgrade instructions**, please refer to [Upgrade](#).

To install a TeamCity server, perform the following:

- Choose the appropriate TeamCity distribution (.exe, .tar.gz or .war) based on the details below
- Download the distribution

- Review software requirements and hardware requirements notes and platform selection
- Review TeamCity Licensing Policy
- Install and configure the TeamCity server per instructions below

This page covers:

- Installing TeamCity Server
 - Installing TeamCity via Windows installation package
 - Installing TeamCity bundled with Tomcat servlet container (Linux, Mac OS X, Windows)
 - Installing TeamCity into Existing J2EE Container
 - Unattended TeamCity server installation
 - Using another Version of Tomcat
- Starting TeamCity server
 - Autostart TeamCity server on Mac OS X
- Installation Configuration
 - Troubleshooting TeamCity Installation
 - Changing Server Port
 - Changing Server Context
 - Java Installation
 - Using 64 bit Java to Run TeamCity Server
 - Setting Up Memory settings for TeamCity Server
- Configuring TeamCity Server
 - Configuring TeamCity Data Directory
 - Editing Server Configuration
 - Configuring Server for Production Use

Installing TeamCity Server

After you obtained the TeamCity installation package, proceed with corresponding installation instructions:

- **.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. Please consider using .tar.gz distribution instead. .war is not recommended to use unless really required (please let us know the reasons).

Compared to the .war distribution, the .exe and .tar.gz distributions:

- include a Tomcat version which TeamCity is tested with, so it is known to be a working combination. This might not be the case with an external Tomcat.
- define additional JRE options which are usually recommended for running the server
- have the **teamcity-server startup script** which provides several convenience options (e.g. separate environment variable for memory settings) and configures TeamCity correctly (e.g. log4j configuration)
- (at least under Windows) provide better error reporting for some cases (like a missing Java installation)
- under Windows, allow running TeamCity as a service with the ability to use the same configuration as if run from the console
- come bundled with a build agent distribution and single startup script which allows for easy TeamCity server evaluation with one agent
- come bundled with the devPackage for **TeamCity plugin development**.
- may provide more convenience features in the future

After installation, the TeamCity web UI can be accessed via a web browser. The default addresses are <http://localhost/> for Windows distribution and <http://localhost:8111/> for tar.gz distribution.

If you cannot access the TeamCity web UI after successful installation, please refer to the [#Troubleshooting TeamCity Installation Issues](#) section.

i 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.

! During the server setup you can select either an internal database or an existing external database. By default, TeamCity uses an HSQLDB database that does not require configuring. This database suites the purposes of testing and evaluating the system. For production purposes, using a standalone external database is recommended.

- [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:

- log on as service right ([related Microsoft page](#))
- write permissions for the [TeamCity Data Directory](#),
- write permissions for the [TeamCity Home](#), i.e. directory where TeamCity was installed,
- all the necessary permissions to work with the source controls used. This includes:
 - access to Microsoft Visual SourceSafe database (if [Visual SourceSafe](#) integration is used).
 - the user, under whose account the TeamCity server service runs, and ClearCase view owner are the same (if the [ClearCase](#) integration is used).

By default, the Windows service is installed under the SYSTEM account. To change it, use the Services applet ([Control Panel | Administrative Tools | Services](#))

Installing TeamCity bundled with Tomcat servlet container (Linux, 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 latest Oracle Java 1.7 JDK, and **since TeamCity 9.1** JDK 1.8, is recommended.

Installing TeamCity into Existing J2EE Container

It is recommended to [use](#) the TeamCity .tar.gz distribution (bundled with Tomcat web server). .war distribution is not recommended unless there are important reasons to deploy TeamCity into existing 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.

Unattended TeamCity server installation

For automated server installation, use .tar.gz distribution.

Typically, you will need to unpack it and make the script perform the steps noted in [Configuring Server for Production Use](#) section.

If you want to get a pre-configured server right away, put files from a previously configured server into the Data Directory. For each new server you will need to ensure it points to a new database (configured in <Data Directory>\config\database.properties) and change <Data Directory>\config\main-config.xml file not to have "uuid" attribute in the root XML element (so new one can be generated) and setting appropriate value for "rootURL" attribute.

Using another Version of Tomcat

If you want to use another version of Tomcat web server instead of the 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

Starting TeamCity server

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

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

- **To start/stop the TeamCity server and one default agent at the same time**, use the `runAll` script, e.g.:
 - Use `runAll.bat start` to start the server and the default agent
 - Use `runAll.bat stop` to stop the server and the default agent
- **To start/stop the TeamCity server only**, use the `teamcity-server` scripts and pass the required parameters. Start the script without parameters to see the usage instructions.
Since TeamCity 9.1, the `teamcity-server` scripts support new options for the `stop` command:
 - `stop n` - Sends the stop command to the TeamCity server and waits up to n seconds for the process to end.
 - `stop n -force` - Sends the stop command to the TeamCity server, waits up to n seconds for the process to end, and terminates the server process if it did not stop.

By default, TeamCity runs on <http://localhost:8111> and has one registered build agent that runs on the same computer.

See the information [below](#) for changing the server port.

If you need to pass special properties to the server, refer to [Configuring TeamCity Server Startup Properties](#).

If TeamCity is installed into an existing web server (.war distribution), start the server according to its documentation. Make sure you configure TeamCity-specific logging-related properties and pass suitable `memory` options.

Autostart TeamCity server on Mac OS X

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

1. Install TeamCity and make sure it works if started from the command line with `bin/teamcity-server.sh start`. We'll assume that TeamCity is installed in the /Library/TeamCity folder
2. Create the `/Library/LaunchDaemons/jetbrains.teamcity.server.plist` file with the following content:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>WorkingDirectory</key>
<string>/Library/TeamCity</string>
<key>Debug</key>
<false/>
<key>Label</key>
<string>jetbrains.teamcity.server</string>
<key>OnDemand</key>
<false/>
<key>KeepAlive</key>
<true/>
<key>ProgramArguments</key>
<array>
<string>bin/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>

```

3. Test your file by running `launchctl load /Library/LaunchDaemons/jetbrains.teamcity.server.plist`. This command should start the TeamCity server (you can see this from `logs/teamcity-server.log` and in your browser).
4. If you don't want TeamCity to start under the root permissions, specify the `UserName` key in the plist file, e.g.:

```

<key>UserName</key>
<string>teamcity_user</string>

```

The TeamCity server will now start automatically when the machine starts. To configure automatic start of a TeamCity Build Agent, see [the dedicated section](#).

Installation Configuration

Troubleshooting TeamCity Installation

Upon successful installation, the TeamCity server web UI can be accessed via a web browser. The default address that can be used to access TeamCity from the same machine depends on the installation package and installation options. (Port 80 is used for Windows installation, unless another port is specified, port 8111 for .tar.gz installation unless not changed in the server configuration).

If the TeamCity web UI cannot be accessed, please check:

- the "TeamCity Server" service is running (if you installed TeamCity as a Windows service);
- the TeamCity server process (Tomcat) is running (it is a java process run in the `<TeamCity home>/bin` directory);
- the console output if you run the server from a console;
- the `teamcity-server.log` and other files in the `<TeamCity home>\logs` directory for error messages.

One of the most common issues with the server installation is using a port that is already used by another program. See [below](#) on changing the default port.

Changing Server Port

If you use the TeamCity server Windows installer, you can set the port to be used during installation.
If you use the .war distribution, refer to the manual of the application server used.

Use the following instructions to change the port if you use the .tar.gz distribution.

If another application uses the same port as the TeamCity server, the TeamCity server (Tomcat server) won't start and this will be identified by "Address already in use" errors in the server logs or server console.

To change the server port, in the `<TeamCity Home>/conf/server.xml` file, change the port number in the not commented "`<Connector>`" XML node (here the port number is 8111):

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

To apply the changes, restart the server. If the server was working with the old port previously, you would need to change the port in all the stored URLs of the server (browser bookmarks, agents' `serverUrl` property, URL in user's IDEs, "Server URL" setting on the "Administration | Global" Settings page).

If you run another Tomcat server on the same machine, you might need to also change other Tomcat server service ports (search for "port=" in the `server.xml` file).

If you want to use the https:// protocol, it should be enabled separately and the process is not specific to TeamCity, but rather for the web server used (Tomcat by default). See also [Using HTTPS to access TeamCity server](#)

Changing Server Context

By default, the TeamCity server is accessible under the root context of the server address (e.g. `http://localhost:8111/`). To make it available under a nested path instead (e.g. `http://localhost:8111/teamcity/`), you need to:

- stop the TeamCity server;
- move the `<TeamCity home>/webapps/ROOT` directory to the `<TeamCity home>/webapps/teamcity` directory;
- start the 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 the latest Oracle Server JRE 1.7 and **since TeamCity 9.1**, 1.8 is recommended ([download page](#)). It is recommended to use the 32-bit installation unless you need to [dedicate more memory](#) to TeamCity server. Please check [the 64-bit Java notes](#) before upgrade.

If you configured any native libraries for use with TeamCity (like a .dll for using Microsoft SQL database Integrated Security option), you need to update the libraries to match the JVM x86/x64 platform.

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

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

- if your TeamCity installation has a bundled JRE (there is the `<TeamCity Home>/jre` directory), update it by installing a newer JRE per installation instructions and copying the content of the resulting directory to replace the content of the existing `<TeamCity home>/jre` directory.



Note that on upgrade, TeamCity will overwrite the existing JRE with the bundled 32-bit version, so you'll have to update to the 64-bit JRE again after upgrade.

- If you also run a TeamCity agent from the `<TeamCity home>/buildAgent` directory, install JVM (Java SDK) installation instead of JRE and copy content of JVM installation directory into `<TeamCity Home>/jre`.
- if there is no `<TeamCity Home>/jre` directory present, set `JRE_HOME` or `JAVA_HOME` environment variables to be available for the process launching the TeamCity server (setting global OS environment variables and system restart is recommended). The variables should point to the home directory of the installed JRE or JVM (Java SDK) respectively and only one variable should be present.
- if you use the .war distribution, Java update depends on the application server used. Please refer to the manual of your application server.

Using 64 bit Java to Run TeamCity Server

TeamCity server can run under both the 32- and 64-bit JVM.

It is recommended to use the 32-bit JVM unless you need to dedicate more than 1.2Gb of memory (via -Xmx JVM option) to the TeamCity process (see [details](#)) or your [database requirements](#) are different.

If you choose to use the 64-bit JVM, note that the memory usage is almost doubled when switching from the 32- to 64-bit JVM, so please make sure you specify at least twice as much memory as for 32-bit JVM, see [#Setting Up Memory settings for TeamCity Server](#).

To update to the 64-bit Java:

- update Java to be used by the server
- set **JVM memory options**. It is recommended to set the following options for the 64-bit JVM: `-Xmx4g -XX:MaxPermSize=270m -XX:ReservedCodeCacheSize=350m`

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.

Once you start using TeamCity for production purposes or you want to load the server during evaluation, you should manually set appropriate memory settings for the TeamCity server.

To **change the memory settings**, refer to [Configuring TeamCity Server Startup Properties](#), or to the documentation of your application server, if you run TeamCity using the .war distribution.

Generally this means setting `TEAMCITY_SERVER_MEM_OPTS` environment variable to the value like `-Xmx750m -XX:MaxPermSize=270m`.

If slowness, OutOfMemory errors occurs or you consistently see memory-related warning in TeamCity UI, increase the settings to the next level.

- **minimum settings** (the 32-bit Java should be used (bundled in .exe distribution)): `-Xmx750m -XX:MaxPermSize=270m`
- **recommended settings** for medium server use (the 32-bit Java should be used): `-Xmx1024m -XX:MaxPermSize=270m` Greater settings with the 32-bit Java can cause OutOfMemoryError with "Native memory allocation (malloc) failed" JVM crashes or "unable to create new native thread" messages
- **maximum settings** for large-scale server use (**64-bit Java** should be used): `-Xmx4g -XX:MaxPermSize=270m -XX:ReservedCodeCacheSize=350m` These settings will be suitable for an installation with more than a hundred of agents and thousands of build configurations. Custom plugins installed might require increasing the values defined via `MaxPermSize` and `Xmx` parameters

Tips:

- the 32-bit JVM can reliably work with up to 1Gb heap memory (`-Xmx1024m`). (This can be increased to `-Xmx1200m`, but JVM under Windows might crash occasionally with this setting.) If more memory is necessary, the 64-bit JVM should be used assigning not less than 2.5Gb (`-Xmx2500m`). It's highly unlikely that you will need to dedicate more than 4Gb of memory to the TeamCity process.
- A rule of thumb is that the 64-bit JVM should be assigned twice as much memory as the 32-bit for the same application. If you switch to the 64-bit JVM, make sure you adjust the memory settings (both `-Xmx` and `-XX:MaxPermSize`) accordingly. It does not make sense to switch to 64 bit if you dedicate less than the double amount of memory to the application.

The recommended approach is to start with initial settings and monitor for the percentage of used memory (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

Configuring Server for Production Use

Out-of-the-box TeamCity server installation is suitable for evaluation purposes. For production use you will need to perform additional configuration which typically includes:

- Select [TeamCity Data Directory](#) location
- Configuring correct server port, Server URL, email and (optionally) Jabber server settings
- Configuring OS-dependent autostart on machine reboot
- Set up external database
- Configure recommended memory settings, use "maximum settings" for active or growing servers
- Plan for regular backups

See also:

[Installation and Upgrade: Setting up and Running Additional Build Agents](#)

Setting up and Running Additional Build Agents

This page covers:

- Prerequisites
 - Necessary OS and environment permissions

- Server-Agent Data Transfers
 - Bidirectional Communication
 - HTTPS agent-server connection
 - Unidirectional Agent-to-Server Communication
 - Changing Communication Protocol
- Installing Additional Build Agents
 - Installing via MS Windows installer
 - Installing via ZIP File
 - Installing via Agent Push
 - Remote Host Requirements
 - Installation
- Starting the Build Agent
 - Manual Start
 - Automatic Start
 - Automatic Agent Start under Windows
 - Build Agent as a Windows Service
 - Automatic Agent Start under Linux
 - Automatic Agent Start under MacOsx
 - Install and start build agent
 - Configure automatic build agent start
 - Reboot
- Stopping the Build Agent
- Configuring Java
 - Upgrading Java on Agents
- Installing Several Build Agents on the Same Machine

Before you can start customizing projects and creating build configurations, you need to configure build agents.



- 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.

Prerequisites

Necessary OS and environment permissions

Before the installation, please review the [Conflicting Software](#) section. In case of any issues, make sure no conflicting software is used.

Please note that in order to run a TeamCity build agent, the user account used to run the Agent requires the following privileges:

Network

- An agent should be able to open HTTP connections to the server using the server address specified in the `serverUrl` property (usually the same URL as server web UI)
- Unless the [unidirectional](#) agent-to-server connection supported [since TeamCity 9.1](#) is configured, the server should be able to open HTTP connections to the agent. The port is determined using the `ownPort` property of the `buildAgent.properties` file as the starting port (9090 by default), and the following IP addresses are tried:
 - the address specified in the `ownAddress` property of the `buildAgent.properties` file (if any)
 - the source IP of the HTTP request received by the server when the agent establishes a connection to the server. If a proxying server is used, it must be [correctly configured](#).
 - the addresses of the network interfaces on the agent machine

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

Common

The agent process (java) should be able to:

- open outbound HTTP connections to the server address (the same address you use in the browser to view the TeamCity UI)
- unless the [unidirectional](#) agent-to-server connection supported [since TeamCity 9.1](#) is configured, accept inbound HTTP connections from the server to the port specified as the `ownPort` property in the `buildAgent.properties` file (9090 by default). Please ensure that any firewalls installed on the 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 the agent upgrade to work, see also Microsoft KB article)
- Debug programs (for take process dump functionality to work)
- Reboot the machine (for agent reboot functionality to work)

For granting necessary permissions for unprivileged users, see Microsoft [SubInACL](#) utility. For example, to grant Start/Stop rights, you might need to execute the `subinac1.exe /service browser /grant=<login name>=PTO` command.

Linux

- user should be able to run the `shutdown` command (for the agent machine reboot functionality and machine shutdown functionality when running in cloud)

Build-related Permissions

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

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

Server-Agent Data Transfers

The default communication between the TeamCity server and an agent is bidirectional, i.e two HTTP connections are established: the server establishes a connection to the agents and the agent establishes a connection to the server. By default, the connections are based on HTTP, which may raise your concern if you are deploying the agent and server into non-secure network environments.

Since TeamCity 9.1, it is sufficient to set up only one connection, enabling [unidirectional](#) communication from the agent to the server.

To view whether the agent-server communication is bidirectional or unidirectional for a particular agent, navigate to [Agents|<Agent Name>|Agent Summary](#) tab, the **Details** section, **Communication Protocol**.

Bidirectional Communication

The bidirectional TeamCity server and agent communication is based on HTTP: the server listens for incoming connection requests from the agent, while the agent listens for incoming server requests.

The HTTP connections from the server to the agents are not secured and thus are exposing potentially sensitive data to any third party that may listen to the traffic between the server and the agents. Moreover, since the agent and server can send "commands" to each other, an attacker that can send HTTP requests and capture responses may in theory trick the agent into executing an arbitrary command and perform other actions with a security impact.

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.

HTTPS agent-server connection

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

However, the data that is transferred via the connections established by the server to agents (all the settings configured on the web UI including the VCS root data) is passed via an unsecured HTTP connection.

Prior to TeamCity 9.1, you could only establish appropriate network security configurations like VPN connections between agent and server machines to secure the data. **Since TeamCity 9.1** you can eliminate the server-to-agent connection by switching to unidirectional agent-to-server communication. See the next section for details.

Unidirectional Agent-to-Server Communication

Since TeamCity 9.1, it is sufficient to set up only the agent-to-server connection.

When an agent is configured for unidirectional communication, the TeamCity server does not establish a connection to the agent. Instead, at startup, the agent establishes a connection to the TeamCity Server, and polls the server periodically for server commands.

The polling protocol used for the agent-to-server connection allows for increased security when the agent-to-server connection is based on HTTPS; it also increases the agents accessibility, e.g. the agents can be behind a firewall with all incoming connections blocked or the agents can be deployed to a network different from that of the server.

Changing Communication Protocol

The communication protocol used by TeamCity agents is determined by the value of the `teamcity.agent.communicationProtocols` internal property. The property accepts a comma-separated ordered list of protocols and is set to `teamcity.agent.communicationProtocols=xm`

`l-rpc`, polling by default, which means that the agent tries to connect using the first protocol from this list and if it fails, it will try to connect via the second protocol in the list.

- To change the communication protocol **for all agents**, set the TeamCity server `teamcity.agent.communicationProtocols internal` property. The new setting will be used by all agents which will connect to the server after the change. To change the protocol for the existing connections, restart the TeamCity server.
- By default, the agent's property is not configured; when the agent first connects to the server, it receives it from the TeamCity server. To change the protocol **for an individual agent** after the initial agent configuration, change the value of the `teamcity.agent.communicationProtocols` property in the [agent's properties](#). The agent's property overrides the server property. After the change the agent will restart automatically upon finishing a running build, if any.

Installing Additional Build Agents

1. Install a build agent using any of the following options:
 - [Using MS Windows installer](#)
 - [By downloading a zip file and installing manually](#)
2. After installation, configure the agent specifying its name and the address of the TeamCity server in the `conf/buildAgent.properties` file.
3. [Start](#) the agent. If the agent does not seem to run correctly, please check the [agent logs](#).

When the newly installed agent connects to the server for the first time, it appears on the [Agents](#) page, [Unauthorized agents](#) tab visible to administrators/users with the permissions to authorize it. Agents will not run builds until they are authorized in the TeamCity web UI. The agent running on the same computer as the server is authorized by default.

The number of authorized agents is limited by the number of agents licenses on the server. See more under [Licensing Policy](#).

Installing via MS Windows installer

1. In the TeamCity Web UI, navigate to the [Agents](#) tab.
2. Click the [Install Build Agents](#) link and select **MS Windows Installer** to download the installer.
3. Run the `agentInstaller.exe` Windows Installer and follow the installation instructions.



Please ensure that the user account used to run the agent service has appropriate [permissions](#)

Installing via ZIP File

1. Make sure a JDK(JRE) 1.6+ is properly installed on the agent computer.
2. On the agent computer, make sure the `JRE_HOME` or `JAVA_HOME` environment variables are set (pointing to the installed JRE or JDK directory respectively).
3. In the TeamCity Web UI, navigate to the [Agents](#) tab.
4. Click the [Install Build Agents](#) link and select **Zip file distribution** to download the archive.
5. Unzip the downloaded file into the desired directory.
6. Navigate to the `<installation path>\conf` directory, locate the file called `buildAgent.dist.properties` and rename it to `buildAgent.properties`.
7. Edit the `buildAgent.properties` file to specify the TeamCity server URL and the name of the agent. Please refer to [Build Agent Configuration](#) section for details on agent configuration.
8. Under Linux, you may need to give execution permissions to the `bin/agent.sh` shell script.



On Windows you may also want to install the [build agent windows service](#) instead of the manual agent startup.

Installing via Agent Push

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

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



SSH note

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

Remote Host Requirements

There are several requirements for the remote host:

Platform	Prerequisites
Linux	<ol style="list-style-type: none">1. Installed JDK(JRE) 1.6+. The JVM should be reachable with the JAVA_HOME(JRE_HOME) global environment variable or be in the global path (i.e. not in user's .bashrc file, etc.)2. The unzip utility.3. Either wget or curl.
Windows	<ol style="list-style-type: none">1. Installed JDK/JRE 1.6+.2. Sysinternals psexec.exe on the TeamCity server. 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

1. In the TeamCity Server web UI navigate to the **Agents | Agent Push** tab, and click **Install Agent....**
If you are going to use same settings for several target hosts, you can [create a preset](#) with these settings, and use it each time when installing an agent to another remote host.
2. In the **Install agent** dialog, either select a saved preset or choose "Use custom settings", specify the target host platform and configure corresponding settings.
Since TeamCity 9.1, agent push to a Linux system via SSH supports custom ports (the default is 22) specified as the **SSH port** parameter. The port specified in a preset can be overridden in the host name, e.g. `hostname.domain:2222`, during the actual agent installation.
3. You may need to download [Sysinternals psexec.exe](#), in which case you will see the corresponding warning and a link to the [Administration | Tools](#) page where you can download it.

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

Starting the Build Agent

TeamCity build agents can be started manually or configured to start automatically.

Manual Start

To start the agent manually, run the following script:

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

Automatic Start

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

[Windows](#)

[Linux](#) : configure daemon process with `agent.sh start` command to start it and `agent.sh stop` command to stop it.

[Mac OS X](#)

Automatic Agent Start under Windows

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

Using the Windows service approach is the easiest way, but Windows applies some constraints to the processes run this way.

A TeamCity agent works reliably under Windows service provided all the [requirements](#) are met, but is often not the case for the build processes configured to be run on the agent.

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

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

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

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 the Windows agent installer, you have an option to install the service in the installation wizard.



Service system account

To run builds, the build agent must be started under a user with sufficient permissions for performing a build and [managing](#) the service. By default, a Windows service is started under the SYSTEM account. To change it, use the standard Windows Services applet (Control Panel\Administrative Tools\Services) and change the user for the TeamCity Build Agent service.

 If you start an Amazon EC2 cloud agent as a Windows service, add a dependency from the TeamCity Build Agent service to the [EC2C onfig](#) service.

The following instruction can be used to install the service manually. This procedure should also be performed to install the 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.
2. Check that the `wrapper.java.command` property in the `<agent home>\launcher\conf\wrapper.conf` file contains a valid path to the Java executable in the JDK installation directory. You can use `wrapper.java.command=.../jre/bin/java` for the agent installed with the Windows distribution. Make sure to specify the path of the `java.exe` file without any quotes.
3. 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 the Windows `net.exe` utility to manage the service once it is installed.

For example (assuming the default service name):

```
net start TCBuildAgent
```

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

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

 A method for assigning rights to manage services is to use the Subinac.exe 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>

Automatic Agent Start under Linux

To run agent automatically on the machine boot under Linux, configure daemon process with the `agent.sh start` command to start it and `agent.sh stop` command to stop it. Refer to an example procedure below:

1. Navigate to the services start/stop services scripts directory:

```
cd /etc/init.d/
```

2. Open the build agent service script:

```
sudo vim buildAgent
```

3. Paste the following into the file :

```

#!/bin/sh
### BEGIN INIT INFO
# Provides:          TeamCity Build Agent
# Required-Start:    $remote_fs $syslog
# Required-Stop:     $remote_fs $syslog
# Default-Start:    2 3 4 5
# Default-Stop:     0 1 6
# Short-Description: Start build agent daemon at boot time
# Description:       Enable service provided by daemon.
### END INIT INFO
#Provide the correct user name:
USER="agentuser"

case "$1" in
start)
  su - $USER -c "cd BuildAgent/bin ; ./agent.sh start"
;;
stop)
  su - $USER -c "cd BuildAgent/bin ; ./agent.sh stop"
;;
*)
  echo "usage start/stop"
  exit 1
;;
esac

exit 0

```

4. Set the permissions to execute the file:

```
sudo chmod 755 buildAgent
```

5. Make links to start the agent service on the machine boot and on restarts using the appropriate tool:

- For Debian/Ubuntu:

```
sudo update-rc.d buildAgent defaults
```

- For Red Hat/CentOS:

```
sudo chkconfig buildAgent on
```

Automatic Agent Start under MacOsx

For MacOsx, TeamCity provides an ability to load a build agent automatically when a build user logs in. For that, TeamCity uses standard MacOs way to start daemon processes - LaunchDaemon plist files.

To configure an automatic build agent startup, follow these steps:

Install and start build agent

- Install a build agent on Mac via buildAgent.zip
- Prepare the conf/buildAgent.properties file

- Fix the launcher permissions, if needed:

```
chmod \+x buildAgent/launcher/bin/\*
```

- Make sure that all files under the `buildAgent` directory are owned by `your_build_user` to ensure a proper agent upgrade process.
- Load the build agent as 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. You can watch the process in the logs:

```
tail -f buildAgent/logs/teamcity-agent.log
```

- When build agent is upgraded and successfully connects to TeamCity server, stop it:

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

Configure automatic build agent start

- Copy the `buildAgent/bin/jetbrains.teamcity.BuildAgent.plist` file to `$HOME/Library/LaunchAgents/` directory.



Old `jetbrains.teamcity.BuildAgent.plist` files bundled with TeamCity before 9.0.4 had a bug which prevent iOS simulator from starting on **OS X Yosemite** ([TW-38954](#)). The fix is to remove `SessionCreate` property from this file.



You can configure to start build agent on system boot, and place `plist` file to `/Library/LaunchDaemons` directory. But in this case there could be some troubles with running GUI tests, and with build agent auto-upgrade. So we **don't recommend** this approach.

See this [external posting](#) for some more details on LaunchDaemons.

- Configure your Mac system to **automatically login** as a build user, as described [here](#)

Reboot

On system startup, build user should automatically log in, and build agent should start.

Stopping the Build Agent

To stop the agent manually, run the `<Agent home>\agent` script with a `stop` parameter.

Use `stop` to request stopping after the current build finished.

Use `stop force` to request an immediate stop (if a build is running on the agent, it will be stopped abruptly (canceled))

Under Linux, you have one more option top use: `stop kill` to kill the agent process.

If the agent runs with a console attached, you may also press **Ctrl+C** in the console to stop the agent (if a build is running it will be canceled).

Configuring Java

A TeamCity Agent is a Java application and it requires JDK version 1.6 or later to work. Oracle Java SE JDK 1.732 bit is recommended ([download page](#)).

The (Windows) .exe TeamCity distribution comes bundled with Java 1.7. If you run a previous version of the TeamCity agent, you might need to repeat the agent installation to update the JVM.

Using x32 bit JDK is recommended. JDK is required for some build runners like [IntelliJ IDEA Project](#), [Java Inspections](#) and [Duplicates](#). If you do not have Java builds, you can install JRE instead of JDK.

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

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

- the <Agent home>/jre directory
- in the directory pointed to by the 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 the 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, do one of the following:

- If the appropriate Java version is detected on the agent, the agent page provides an action to upgrade the Java automatically. Upon the action invocation, the agent will restart using another JVM installation.
- (Windows) Since the build agent .exe installation comes bundled with the required Java, you can just reinstall the agent using the .exe installer obtained from the TeamCity server | **Agents** page.
- Install a required Java on the agent and restart the agent - it should then detect it and provide an action to use a newer Java in web UI.
- Install a 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 all agents equally regardless of whether they are installed on the same or on different machines.

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

- The builds running on such agents should not conflict by any resource (common disk directories, OS processes, OS temp directories).
- Depending on the hardware and the builds, you may experience degraded builds' performance. Ensure there are no disk, memory, or CPU bottlenecks when several builds are run at the same time.

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

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

Make sure there are no build configurations with the absolute [checkout directory](#) specified (alternatively, make sure such build configurations have the "clean checkout" option enabled and they cannot be run in parallel).

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

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

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

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



For step-by-step instructions on installing a second Windows agent as a service, see a related [external blog post](#).

See also:

[Concepts: Build Agent](#)

Build Agent Configuration

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

All the system and environment properties defined in the file will be passed to every build run on the agent.

The file is a Java properties file.

A quick guide is:

- use `property_name=value<newline>` syntax
- use `#` in the first position of the line for a comment
- use `/` instead of `\` as the path separator. If you need to include `\` escape it with another `\`.
- whitespaces within a line matter

This is an example of the file:

```
## The address of the TeamCity server. The same as is used to open 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 the "name" property is not specified, the server will generate a build agent name automatically. By default, this name will be created from the build agent's host name.

The file can be edited while the agent is running: the agent detects the change and (upon finishing a running build, if any) restarts automatically loading the new settings.

See also:

[Concepts: Build Agent](#)

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

TeamCity Integration with Cloud Solutions

TeamCity integration with cloud (IAAS) solutions allows TeamCity to provision virtual machines running TeamCity agents on-demand based on the build queue state.

This page covers **general information** about the configuration of integration. For the list of currently supported solutions, refer to [Available Integrations](#).

On this page:

- General Description
- Available Integrations
- TeamCity Setup for Cloud Integration
 - Preparing a virtual machine with an installed TeamCity agent
 - Preparing a virtual machine
 - Capturing an image from a virtual machine
 - Configuring a cloud profile in TeamCity
- Estimating Costs
 - Traffic Estimate
 - Running Costs

General Description

In a large TeamCity setup with many projects, it's can be difficult to predict the load on build agents, and the number of agents we need to be running. With the cloud agent integration configured, TeamCity will leverage clouds elasticity to provision additional build agents on-demand.

For each queued build TeamCity first tries to start it on one of the regular, non-cloud agents. If there are no usual agents available, TeamCity finds a matching cloud image with a compatible agent and starts a new instance for the image. TeamCity ensures that number of running cloud instances limit is not exceeded.

The integration requires:

- a configured virtual machine with an installed TeamCity agent in your cloud pre-configured to start the TeamCity agent on boot,
- a configured cloud [profile in TeamCity](#).

Once a cloud profile is configured in TeamCity with one or several images, TeamCity does a test start of one instance for all the newly added images to learn about the agents configured on them. When the agents are connected, TeamCity stores their parameters to be able to correctly process build configurations-to-agents compatibility. An agent connected from a cloud instance started by TeamCity is automatically authorized, provided there are available agent licenses: the number of cloud agents is limited by the total number of agent licenses you have in TeamCity. After that the agent is processed as a regular agent.

Depending on the profile settings, when TeamCity realizes it needs more agents, it can either

- start an existing virtual machine and stop it (after the build is finished or an idle timeout elapses). The machines that are stopped will be deallocated so the virtual machine fee does not apply when the agent is not active. The storage cost for this type of TeamCity agent will still apply.
- create a new virtual machine from an image. Such machines will be destroyed (after the build is finished or an idle timeout elapses). This ensures that the machines will incur no further running costs.

The disconnected agent will be removed from the authorized agents list and deleted from the system to free up TeamCity build agent licenses.

Available Integrations

Integration with cloud solutions is implemented as plugins. The platform-specific details are covered on the following pages:

- [Amazon EC2 \(bundled\)](#)
- [Windows Azure](#)
- [VMWare vSphere](#)

For integration with other cloud solutions, see [Implementing Cloud support](#).

TeamCity Setup for Cloud Integration

This section describes general steps required for cloud integration.

Preparing a virtual machine with an installed TeamCity agent

The requirements for a virtual machine/image to be used for TeamCity cloud integration:

- The TeamCity agent must be correctly [installed](#) and configured to start [automatically](#) on the machine startup.
- the `buildAgent.properties` file can be left "as is". The `serverUrl`, `name`, and `authorizationToken` properties can be left empty or set to any value, they are ignored when TeamCity starts the instance **unless otherwise specifically stated** in the platform-specific documentation.

Provided these requirements are met, the usual TeamCity agent installation and cloud-provider image bundling procedures are applicable.

If you need the [connection](#) between the server and the agent machine to be secure, you will need to set up the agent machine to establish a secure tunnel (e.g. VPN) to the server on boot so that the TeamCity agent receives data via the secure channel. Please keep in mind that communication between TeamCity agent and server is bi-directional and requires an open port on the agent as well as on the server.

Preparing a virtual machine

1. Create and start a virtual machine with desired OS installed.

2. Connect and log in to the virtual machine.
3. Configure the running instance:
 - a. [Install](#) and configure build agent.
 - Configure the server name and agent name in the `buildAgent.properties` file — this is optional if TeamCity will be configured to launch the image, but it is useful to test the agent is configured correctly.
 - It usually makes sense to specify `tempDir` and `workDir` in `conf/buildAgent.properties` to use a non-system drive (e.g. D drive under Windows)
 - b. Install any additional software necessary for the builds on the machine (e.g. Java, the .Net framework)
 - c. Start the agent and wait until it connects to server, ensure it is working OK and is compatible with all necessary build configurations (in the TeamCity Web UI, go to the **Agents** page, select the build agent and view the **Compatible Configurations** tab), etc.
 - d. Configure the system so that the agent is [started on the machine boot](#) (and make sure TeamCity server is accessible on the machine boot).
 - e. Check the port on which the build agent will listen for incoming data from TeamCity and open the required firewall ports (usually 9090).
4. Test the setup by rebooting machine and checking that the agent connects normally to the server. Once the agent connects, it will automatically update all the plugins. Please wait until the agent is connected completely so that all plugins are downloaded to the agent machine.

If you want TeamCity to start an existing virtual machine and stop it after the build is finished or an idle timeout elapses, the setup above is all you need. If you want to TeamCity to create and start virtual machines from an image and terminate the machine after use, the image should be captured from the virtual machine that was created.

Capturing an image from a virtual machine

Do the following:

1. Complete the steps for [creating a virtual machine](#).
 - Remove any temporary/history information in the system.
 - Stop the agent (under Windows, stop the service but leave it in the *Automatic* startup type)
 - (optional) Delete the content of the `logs` and `temp` directories in the `agent home`
 - (optional) Clean up the `<Agent Home>/conf/` directory from platform-specific files
 - (optional) Change the `buildAgent.properties` file to remove the `name`, `serverUrl`, and `authorizationToken` properties **unless otherwise specifically stated in the platform-specific documentation**.
2. Make a new image from the running instance. Refer the cloud documentation on how to do this.



TeamCity agent auto-upgrades whenever distribution of agent (e.g. after TeamCity upgrade) or agent plugins on the server changes. If you want to reduce the agent startup time, you might want to capture a new virtual machine image after the agent distribution or plugins have been updated.

Configuring a cloud profile in TeamCity

Agent Cloud Profiles are configured in the Server Administration UI, on the **Administration | Agent Cloud**.

Estimating Costs

The cloud provider pricing applies. Please note that the charges can depend on the specific configuration implemented to deploy TeamCity. We advise you to check your configuration and the cloud account data regularly in order to discover and prevent unexpected charges as early as possible.

Please note that traffic volumes and necessary server and agent machines characteristics depend a big deal on the TeamCity setup and nature of the builds run.

Traffic Estimate

Here are some points to help you estimate TeamCity-related traffic:

If the TeamCity server is not located within the same region or affinity group as the agent, the traffic between the server and agent is subject to usual external traffic charges imposed by your provider.

When estimating traffic, please remember that there are many types of traffic related to TeamCity (see the non-complete list below).

External connections originated by server:

- VCS servers
- Email and Jabber servers
- Maven repositories
- NuGet repositories

Internal connections originated by server:

- TeamCity agents (checking status, sending commands, retrieving information like thread dumps, etc.)

External connections originated by agent:

- VCS servers (in case of agent-side checkout)
- Maven repositories
- NuGet repositories
- any connections performed from the build process itself

Internal connections originated by agent:

- TeamCity server (retrieving build sources in case of server-side checkout or personal builds, downloading artifacts, etc.)

Usual connections used by the server

- Web browsers
- IDE plugins

Running Costs

Cloud providers calculate costs based on the virtual machine uptime, so it is recommended to adjust the timeout setting according to your usual builds length. This reduces the amount of time a virtual machine is running.

It is also highly recommended to set an execution timeout for all your builds so that a build hanging does not cause prolonged instance running with no payload.

Setting Up TeamCity for Amazon EC2

TeamCity Amazon EC2 integration allows you to configure TeamCity with your Amazon account and then start and stop images with TeamCity agents on-demand based on the queued builds.

For integrations with other cloud solutions, see [Cloud-VMWare plugin](#) and [Implementing Cloud support](#).

On this page:

- General Description
- Configuration
 - Required IAM permissions
 - Optional permissions
 - Preparing Image with Installed TeamCity Agent
 - Configuring a cloud profile in TeamCity
 - IAM profiles
 - New instance types
 - Estimating EC2 Costs
 - Traffic Estimate
 - Uptime Costs

General Description

It is assumed that the machine images are pre-configured to start TeamCity agent on boot. (See details [below](#)). Exception is the 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. If an EBS-based instance id is specified in the images list, the instance is stopped instead. On instance terminating/stopping, its disconnected agent is removed from authorized agents list and is deleted from the system.

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](#).

Required IAM permissions

TeamCity requires the following permissions for Amazon EC2 Resources:

- ec2:Describe*
- ec2:StartInstances
- ec2:StopInstances
- ec2:TerminateInstances
- ec2:RebootInstances
- ec2:RunInstances
- ec2:ModifyInstanceAttribute
- ec2:*Tags

In order to use spot instances feature the following additional permissions are required:

- ec2:RequestSpotInstances
- ec2:CancelSpotInstanceRequests

An example of custom IAM policy definition (allows all ec2 operations from a specified IP address):

AWS IAM Policy Definition Example » [Expand source](#)

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "1",  
            "Effect": "Allow",  
            "Action": "ec2:*",  
            "Condition": {  
                "IpAddress": {"aws:SourceIp": "<TeamCity server IP address>"}  
            },  
            "Resource": "*"  
        }  
    ]  
}
```

Optional permissions

See the section below for permissions to set IAM roles on an agent instance.

View information on example policies for [Linux](#) and [Windows](#) on the Amazon website.

Preparing Image with Installed TeamCity Agent

Here are the requirements for an image that can be used for TeamCity cloud integration:

- TeamCity agent should be correctly [installed](#).
- TeamCity agent should start on machine startup
- `buildAgent.properties` can be left "as is". `"serverUrl"`, `"name"` and `"authorizationToken"` properties can be empty or set to any value, they are ignored when TeamCity starts the instance.

Provided these requirements are met, usual TeamCity agent installation and cloud-provider image bundling procedures are applicable.

If you need the [connection](#) between the server and the agent machine to be secure, you will need to 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).

Configuring a cloud profile in TeamCity

Next configure Amazon EC2 [Agent Cloud Profile](#) in the Server Administration UI, on the **Administration | Agent Cloud**.

IAM profiles

Since **TeamCity 9.1.1** it is possible to use IAM profiles with build agents launched as Amazon EC2 instances, which requires the supplied AWS account to have the following permissions:

- `iam:ListInstanceProfiles`
- `iam:PassRole`

IAM profiles must be [preconfigured](#) in Amazon EC2. In the Teamcity Web UI, the **IAM profile** dropdown enables you to select a role. Every new launched EC2 instance will assume the [selected IAM role](#).

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.

New instance types

Since Amazon doesn't provide a robust API method to retrieve all instance types, Amazon integration relies on periodical update of AWS SDK to make new instance types available.

However, there's a workaround if you are not willing to wait. To register new Instance Types, use the following [internal property](#):

```
teamcity.ec2.instance.types property with new instance types separated by ","
```

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

The TeamCity [Administration|Plugins List](#) page allows you to:

- view the plugins currently installed on the server
- view the full list of [available plugins](#)
- upload the plugin .zip file to the [TeamCity Data Directory](#)/plugins directory.

Steps to install a TeamCity plugin:

To install an uploaded plugin, restart the TeamCity server.

Alternatively, do the following:

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.

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

Installing Agent Tools

In TeamCity *an agent tool* (i.e. a set of files/a binary distribution) is a type of plugin without any classes loaded into the runtime; agent tools are used to only distribute binary files to agents.

TeamCity allows you to install additional tools on all the agents, which is especially useful in the environments with a large number of build agents as you can distribute tools to or remove them from all build agents at once, centralize configuration files distribution (e.g. you want to distribute a custom configuration file/library to all agents), etc.

The structure of the tool plugin is described [on the Plugins Packaging page](#).

A 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 the name of the zip file as the tool name on all agents. The zip file will be automatically unpacked on the agents to the directory with the same name.

 Make sure the name of your tool is not the same as an existing TeamCity build agent tool: check the [Agent Home Directory](#)/tools folder for the TeamCity tools. A custom tool with the same name as an existing TeamCity build agent tool

is ignored and will not be installed.

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 9.1.2 to 9.1.3

Known issues

Connection to CloudForge SVN servers could be broken due to JVM update in TeamCity windows distribution [TW-42577](#)

Changes from 9.1.1 to 9.1.2

Known issues

Command line runner can fail to execute custom script if it has non-default hashbang specified at the beginning of the script: [TW-42498](#)

Build status icons

Build status icons updated to more "standard" look and are of a bit larger size now.

Bundled tools updates

JetBrains ReSharper command line tools (.NET inspection and duplicates) updated to match Resharper 9.2 release

TeamCity Visual Studio Addin Web installer updated to ReSharper 9.2 release

Bundled JetBrains dotCover updated to version 3.2

Bundled Oracle JRE (in both Server and Agent .exe installers) updated to version 1.8.0_60 (32-bit)

Changes from 9.0 to 9.1.1

Bundled Jacoco coverage library updated to version 0.7.5

Perforce VCS Roots with disabled ticket authentication won't run 'p4 login' operation anymore if password authentication is disabled on the Perforce server.

I.e. if password authentication is disabled, "Use ticket-based authentication" option must be enabled on the VCS Root. [TW-42818](#)

Changes from 9.0.x to 9.1

Bundled Ant

Bundled Ant distribution has been upgraded from 1.8.4 to 1.9.6. Note that Ant build steps using bundled Ant will use another version of Ant after the server upgrade. Ant 1.9.6 requires Java 1.5 at least, so builds using Ant and running under Java 1.4 will stop working.

MSTest runner converted into Visual Studio Tests runner

MSTest runner is merged with [VSTest console runner](#) (previously provided as a separate plugin) into the [Visual Studio Tests](#) runner.

Note that after upgrade to TeamCity 9.1, MSTest build steps are automatically converted to the Visual Studio Tests runner steps, while VSTest steps remain unchanged.



If you have used [VSTest.Console runner](#) plugin, make sure that you have latest version (build [32407](#)) installed. The plugin version can

be viewed on Administration->Plugins List page. Earlier versions of this plugin are **not compatible** with TeamCity 9.1 and may cause malfunction of .Net related build runners which can manifest with "java.lang.NoSuchMethodError: jetbrains.buildServer.runner.NUnit.NUnitVersion.parse(Ljava/lang/String;)" build errors. The plugin can be downloaded from [its page](#). Consider migrating your vstest.console execution steps to the bundled Visual Studio Tests runner.

MSTest installation agent properties

TeamCity agent automatically detects the installed MSTest and used to expose the locations in system.MSTest.N.N system properties.

Since TeamCity 9.1, the locations are exposed via teamcity.dotnet.mstest.N.N configuration parameters. Check [TW-41845](#) for a workaround if you cannot easily change the properties usage.

Nested test reporting

Previously TeamCity supported a case when one test could have been reported from within another test using [service messages](#). Now, after the fix of [TW-40319](#), starting another test finishes the currently started test in the same "flow". To still report tests from within other tests, you will need to specify another [flowId](#) in the nested test service messages.

REST API

REST API uses version 9.1. Previous versions of API are still available under /app/rest/9.0, /app/rest/8.1, /app/rest/7.0, /app/rest/6.0 URLs.

Finding builds

Summary (tl;dr): Some build filtering rules has subtle changes. Most importantly, a queued build can now be returned instead of 404 when searching by build id and meaning of the "project" locator dimension has changed to be not recursive. Also, failed to start builds are now not included until "failedToStart:any" locator dimension is specified. Details:

Affected requests: /app/builds/<locator>..., /app/builds?locator=<locator>, /app/buildTypes/<btLocator>/builds and others with build locator locator: id:<number> or taskId:<number>

- previously, if the matching build was a queued one, 404 (Not Found) was returned
- now the queued build is returned

locator: project:<id>...

- previously, all the builds belonging to build configurations of the project and all its subprojects (recursively) were found
- now only the builds belonging to build configurations of the project specified are found. For finding the builds recursively, use "affectedProject:<id>." dimension. This makes the usage consistent with build type locators.

locator: tag:<text>

- previously, when "<text>" used ":" character, that used to treat the entire "<text>" as tag name
- now the "<text>" is parsed as a nested locator. For searching tags with ":" character, locator "tag:(name:(<tag>))" should be used

locator: <text>

- previously, if <text> is not a number, response was 400 (Bad Request) with "LocatorProcessException: Invalid single value: '<text>'". Should be a number." message.
- now search by build number across all builds on the server is performed (this is not recommended to be used on production servers). For not found builds 404 (Not found) response is returned

locator: id:<number>,xxx:yyy

- previously, build was found by id "<number>", other dimensions were ignored
- now if the build found by id does not match other dimensions, response is 404 (Not Found)

locator: agent:<agentLocator>

- previously <agentLocators> was used as is, without applying any defaults (unauthorized agents were included until specifically excluded)
- now <agentLocator> has the same behavior as in /app/rest/agents request: unauthorized agents are excluded by default

Finding projects

Searching project by name used to return 404 error if several projects were matched. Now will return the first project found.

Build's artifacts

There are several bugs fixed in listing build artifacts via /app/rest/builds/<locator>/artifacts/* requests which can cause subtle changes in the results for the request. Check the new behavior if you relied on the response.

The most important changes are:

- the initial path specified via URL part is searched for without current locator value, it will not generate 404 responses until there is no such artifact on disk.
- archives are not treated as directories (do not have children elements) by default. Specify "browseArchives:true" to treat archives as

directories (in "recursive:true" mode only one level of archives is treated as directories)

Agents

Agents which are not known to the system (which were deleted) used to have id "-1". Now "id", "pool" and some other entries are no longer included for such agents.

Xcode 7 Support

Experimental support for Xcode 7 has been added.

Issue trackers integration

Due to API changes, third party issue trackers integration plugins might not be compatible with TeamCity version 9.1. Old plugins will not work and will report "java.lang.NoSuchMethodError":

`jetbrains.buildServer.issueTracker.AbstractIssueProviderFactory.<init>(Ljetbrains/buildServer/issueTracker/IssueFetcher;Ljava/lang/String;)V`
error in `teamcity-server.log` log (more details in the [issue](#)). If you observe such errors, please contact the [plugin authors](#). If you are the author of affected plugin, please refer to the related notes in [Open API Changes](#).

Changes from 9.0.4 to 9.0.5

No noteworthy changes.

Changes from 9.0.3 to 9.0.4

No noteworthy changes.

Changes from 9.0.2 to 9.0.3

No noteworthy changes.

Changes from 9.0.1 to 9.0.2

No noteworthy changes.

Changes from 9.0 to 9.0.1

Known Issues

If you have enabled versioned settings for projects which use meta-runners in TeamCity 9.0, on upgrade and following commit into the settings VCS root, the meta runners will be deleted from the server. Workaround is to commit the meta-runners definitions into the settings repository manually. Related issue: [TW-39519](#).

Oracle 10.x JDBC driver is not supported anymore

Due to missing support for national character sets (nvarchar) in Oracle 10.x JDBC drivers, TeamCity 9.0.1 will ask to upgrade Oracle JDBC drivers to the latest version. The minimal supported version of Oracle JDBC driver is 11.1.

Changes from 8.1.x to 9.0

Known Issues

- If you have custom artifact cleanup rules configured which mention ".teamcity" directory, build logs can be deleted by the cleanup procedure. Make sure you have build logs backup before upgrade and remove all the custom artifacts cleanup rules with ".teamcity". Related issue: [TW-40042](#). This issue is fixed in 9.0.3 release.
- If you use Microsoft SQL Server database with TeamCity, after the scheduled cleanup background run, TeamCity UI pages can lock until the server restart. See [TW-39549](#) for details. This issue is fixed in 9.0.2 release.
- If you use LDAP authentication on the server and there are lots of login attempts on the server (e.g. there is an active REST-using script), OutOfMemory errors can occur and require server restart. Consider installing an LDAP plugin with a fix from the [issue](#). This issue is fixed in 9.0.1 release.
- If you have large Maven projects, you can see builds failing with OutOfMemoryError. This is caused by update of back-end embedded Maven to 3.2.3 which has bigger memory footprint. Consider increasing Build Agent [memory limits](#) Related issue: [TW-41052](#)

UUID in XML settings files

Since TeamCity 9.0, disk-stored XML settings definitions of projects, build configurations and VCS roots have unique non-human-readable id (uuid) stored. These ids are automatically generated and are assumed to globally unique. On settings files copying, you need to change/make unique not only id (file name) and name (across siblings) of the entity , but also remove it's uuid from the file. TeamCity will generate new uuid automatically.

Build logs storage

The location of the build logs in the internal format stored under [TeamCity Data Directory](#) has changed. The build log files in internal format are now stored under hidden build artifacts.

Namely, the location has changed from `system/messages/CHyy/xxyy.*` to `system/artifacts/<PROJECT EXTERNAL ID>/<BUILD CONFIGURATION NAME>/xxyy/.teamcity/logs/buildLog.*`.

Old build logs are migrated to the new location on TeamCity server startup ([TW-37362](#)). To avoid this migration, `teamcity.skip.logs.migration` internal property should be set **before** server startup.

Builds re-indexing after upgrade

On the first server start after upgrade from a version prior to 9.0, the server will reindex all builds for the purpose of builds search functionality and NuGet feeds. During the indexing time, some builds will not be available in the search results and in NuGet feeds. The server can also behave in less performant manner way during the indexing. `teamcity-server.log` has corresponding logging. On indexing finishing, there are "BuildIndexer (search) - Finished re-indexing builds" and "BuildIndexer (metadata) - Finished re-indexing builds" lines in the log.

Integration with issue trackers

Since TeamCity 9.0, the issue trackers are configured on the project level instead of the global server-wide configuration.

On the server upgrade, all existing issue tracker integrations are moved to the Root project, which makes them still accessible to all the projects on the server.

WebSocket connections and proxy servers

Since 9.0, TeamCity tries to establish WebSocket connections between the browser and the server for the UI updates. If you have a proxy server (like nginx) in front of the TeamCity web UI, make sure that the proxy is [configured](#) properly to support WebSocket connections.

If the proxy is misconfigured or does not support the WebSocket protocol, a server health item will be shown for TeamCity system administrators. In this case TeamCity will use plain old polling for the UI updates as before.

REST API

REST API uses version 9.0. Previous versions of API are still available under `/app/rest/8.1`, `/app/rest/7.0`, `/app/rest/6.0` URLs.

- Change bean: the `webLink` attribute is renamed to `webUrl` to match other beans ([TW-34398](#)).
- Sub-elements representing empty collections in some of the beans are no longer included into responses (used to be included as an empty tag in XML).
- the `builds changes` element does not include the "count" attribute by default (for performance reasons), count can still be included by providing fields parameter like: "`fields=$long,changes(count,href)`"
- The `/app/rest/agents` request now returns all the authorized agents by default (used to include unauthorized connected agents as well)
- queued builds now have the `id` attribute instead of the `taskId` attribute (they are the same for new builds since TeamCity 9.0)

Build tags-related changes

The `/app/rest/builds/<buildLocator>/tags` build request now returns a different XML: `<tags count="1"><tag name="TAG"/></tags>` instead of `<tags><tag>TAG</tag></tags>`.

The same applies to the `/app/rest/buildTypes/<buildTypeLocator>/<buildTypeLocator>/buildTags` request.

The same change in the structure also applies to the build's entity nested "tags" element.

To create a tag, there is an old way to post a plain-text tag name to the `app/rest/builds/<buildLocator>/tags` URL.

When sending POST or PUT XML or JSON requests to the URL, the new XML format is to be used (`<tag name="TAG"/></tags>` instead of `<tag>TAG</tag>`).

Handling tests with the same name within a build

In TeamCity 9.0, multiple tests with the same name within the same build are considered a single test with an invocation count. If any of these test runs fail, the whole test is considered failed in the build. The related issue is [TW-24212](#).

This change results in drop of test number counters in builds which have multiple runs for the same test. If you have a build failure condition which relies on test number in the build, this change may affect you.

If you need the tests to be treated as separate ones, consider running them in the test suites with different names or otherwise changing the test/running logic to change the full test name displayed in TeamCity.

Database-related changes

The national character sets (nchar, nvarchar, nclob types) for text fields are now supported in MS SQL databases used by TeamCity. It is recommended to use the Microsoft native JDBC driver, as JTDS JDBC driver does not support the nchar and nvarchar characters.

Upon upgrade and entering the normal working status, TeamCity starts a background process to move the entries from the `vcs_changes`

database table to vcs_change table. This process is transparent and you can continue working with the server as usual. It has negligible impact on the server performance and the only affected logic is the projects import feature (it is recommended to be used only with backups taken after the process is completed). The progress of the process can be seen on the Backup section in the server administration, along with "TeamCity is currently optimizing VCS-related data in the database for better backup/restore performance" message.

The other important thing is that the data copying increases the size of the raw database storage.

If this is an issue for your case (e.g. it might be with Microsoft SQL Server database with set database size limit), it is recommended to ensure the database size limit is twice the current size before the upgrade. It is possible to perform database-specific procedures to shrink the storage to match the actually stored data after the VCS changes migration process finishes.

VCS Root-related changes

The Git and Mercurial VCS roots no longer provide the ability to specify a custom clone path on the server for new VCS roots. If you need this ability, set the following internal properties to `true` for git and mercurial respectively: `teamcity.git.showCustomClonePath`, `teamcity.hg.showCustomClonePath`.

Visual Studio Addin

The TeamCity Add-in installed as a part of [ReSharper Ultimate](#) will remove the pre-bundle products versions: TeamCity and ReSharper versions prior to 9.0, dotCover prior to 3.0, dotTrace prior to 6.0.

Besides, it will not use the settings provided by the 8.1 version. The traditional add-in downloaded from TeamCity server can still use settings from previous version.

Other

TeamCity agent installation via the Java web start installation package is no longer available.

Changes from 8.1.1 to 8.1.4

No noteworthy changes.

Changes from 8.1 to 8.1.1

Command Line Runner

The change in behavior introduced in 8.1 (see [below](#)) has been fixed. Command line runners using "Executable with parameters" option which were created/changed with TeamCity 8.1 can expose a change in behavior with the upgrade. The recommended approach is to switch to "Custom script" option instead of "Executable with parameters" in command line runner.

Separate download for VSTest.Console runner

VSTest console runner is no longer bundled with TeamCity and is available as a separate plugin. For download details, see the [plugin page](#)

Changes from 8.0.6 to 8.1

Known issue with creating MS SQL database with integrated security

When installing TeamCity anew and creating an MS SQL database with integrated security using the database setup UI, you may receive an error. We are planning to resolve it in the next bugfix, meanwhile use the following workaround:

1. After receiving the error, stop the TeamCity server.
2. Start the TeamCity server.
3. On the database configuration screen, fill out the required information. **Do not use the Refresh button.** Make sure the information specified is correct.
4. Continue with the configuration steps.

If for some reason the workaround above does not resolve the problem, do the following:

1. Start the TeamCity server, approve a new database creation, configure the MS SQL access with a login and password, without [integrated security](#).
2. Ensure that TeamCity works properly.
3. Stop the TeamCity server.
4. Modify the `database.properties` file: configure MS SQL connection string to use integrated security, and remove the login and password.
5. Start the TeamCity server again.

Known issue with VSTest.Console runner

A new "VSTest.Console" runner which first appeared in TeamCity 8.1 is in experimental state and is not recommended for production use at this time. It will not be present in TeamCity 8.1.x by default (will be available as a separate download).

Known issue with PowerShell runner

PowerShell runner plugin is broken in 8.1. Fix is available, please follow instructions in [issue comment](#).

Known issue with Command Line Runner

Command line runner using "Executable with parameters" option can process quotes ("") and percentage signs (%) in a bit different way 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.

Memory Settings

If you have not [switched to 64 bit JVM](#) yet and use -Xmx1300 memory setting for the server and the server is running on Windows, consider decreasing the setting to -Xmx1200 as otherwise you might encounter "Native memory allocation (malloc) failed" JVM crash. See the [recommended memory settings](#) for details.

Actions menu

Some actions has moved under the "Actions" button available at the top-right of the page, near Run button.

These include:

"Label this build sources" on Changes tab of a build,
"Pause", "Copy", "Move", "Delete", "Associate with Template", "Extract Template", "Extract Meta-Runner" on build configuration settings administration page,
"Copy", "Move", "Delete", "Archive", "Bulk edit IDs" on project settings administration page.

Create Maven build configuration is not available by default

Action "Create Maven build configuration" is no longer available. Most of its functionality is covered by create project from URL and create VCS root from URL pages.

triggeredBy parameter from GroovyPlug plugin

The build.triggeredBy and build.triggeredBy.username configuration parameters provided by the [plugin](#) added by the plugin are now [available](#) without the plugin under teamcity.build.triggeredBy and teamcity.build.triggeredBy.username names respectively. Consider migrating to the latter set of parameters 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

The latest version of the API has not changed, it is still "8.0" while there are changes in the API detailed below. If you find this inconvenient for your REST API usages, please comment in the corresponding [issue](#).

Entities returned in the response of REST API requests might now exclude attributes/elements with empty/default values. This is relevant for boolean fields with "false" value and empty collections. The recommended approach is to make sure the client code assumes "false" as a value for not present boolean attributes/elements.

"projectName" of buildType node now contains full project name (with the names of the parent projects) instead of the short name of the project.

In the lists of builds, "startDate" attribute is not longer included in the "build" node. It has become an element instead of attribute to match the full build data representation. If your REST API usage is affected, check [a way](#) to get that element in a request for the list of builds.

Requests /app/rest/buildTypes/XXX/parameters/YYY and /app/rest/projects/XXX/parameters/YYY now support "text/plain" and "application/xml" responses. To get plain text response (which was the only supported way before 8.1) you will need to supply "Accept: text/plain" header to the request.

Password properties of the VCS roots are now included into the responses, just without values.

CCTray-format XML (app/rest/cctray/projects.xml) does not include paused build configurations now.

Response to the experimental request /app/rest/buildTypes/XXX/investigations has changed the format and got additional fields to cover tests and problem investigations. There is an internal property rest.beans.buildTypeInvestigationCompatibility to include removed sub-items. Please let us know via [support email](#) if you need to use the internal property.

Eclipse plugin

Dropped support of Subversion 1.4-1.6. Now only Subversion 1.7-1.8 working copies formats supported.

Changes from 8.0.5 to 8.0.6

No noteworthy changes.

Changes from 8.0.4 to 8.0.5

No noteworthy changes.

Changes from 8.0.3 to 8.0.4

First Cleanup

First Cleanup after server upgrade might take a bit more time then regularly if there are many builds on the server. Following cleanups will then run a bit faster then in previous versions.

Changes from 8.0 to 8.0.3

No noteworthy changes.

Changes from 7.1.x to 8.0

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=<locator>`" no longer return personal and canceled builds by default. To include those, make sure you add ",personal:any,canceled:any" to the locators.

The "relatedIssues" element of the build entity no longer contains a full list of related issues. It has only the "href" attribute whose value can be used to get the related issues via a separate request.

There is also an internal property "rest.beans.build.inlineRelatedIssues" which can be set to `true` to return the "relatedIssues" node back for compatibility. See [TW-20025](#) for details. Also, the "`.../builds/xxx/related-issues`" URL is renamed to "`.../builds/xxx/relatedIssues`".

The "source_buildTypeId" property is dropped from snapshot and artifact dependency nodes. Instead, the "source-buildType" sub-element is added with a reference to the build type.

Creating dependencies is still supported with the "source_buildTypeId" property, but is deprecated. There is an internal property "rest.compatibility.includeSourceBuildTypeInDependencyProperties" which can be set to true to include the "source_buildTypeId" property back.

In version 8.0 VCS roots support project hierarchy:

- When creating a VCS root, the `project` element should always be provided now. The element supports the `locator` attribute to specify the project.
- the `shared` attribute is dropped from the VCS root: after upgrade, such VCS roots are attached to the root project (with the "_Root" ID) and become globally available.
- when copying projects and build configurations, the `shareVCSRoots` attribute is no longer present. To make the VCS root available to projects and build configurations, move it to the parent/root project and then proceed with the copying.

It is recommended to create projects hierarchy which corresponds to organizational/settings sharing structure and move the VCS roots to most nested umbrella projects. Users then can be granted "Create / delete VCS root" role in the project to be able to edit VCS roots. Please note that users can edit a VCS root only if it is used in the projects they have "Edit project" permission for.

The "template" attribute in a build configuration template node is renamed to "templateFlag".

PUT for /users/<locator>/roles and /userGroups/<locator>/roles now accepts list of roles as it should and replaces existing roles instead of accepting single roles and adding it.

Many of PUT and POST requests which used to return nothing now return the entities created.

Open API changes

See [details](#)

Shared Resources plugin

If you used the [Shared Resources plugin](#) with TeamCity 7.1.x, make sure to remove it as it is now bundled. See the [upgrade instructions](#).

Queue Manager plugin

If you used the [QueueManager plugin](#), make sure to remove it as it is now bundled. See the [upgrade instructions](#)

Bundled Maven

Maven bundled with TeamCity upgraded to version 3.0.5.

HTTPS connections from agents to server

If your agents connect to the TeamCity server by HTTPS protocol, and after upgrade agents fail to connect with error messages like:
`javax.net.ssl.SSLHandshakeException: Received fatal alert: handshake_failure`

then you should change Tomcat SSL connector configuration, i.e. add the following attribute to SSL connector and restart TeamCity server:
`sslEnabledProtocols="TLSv1,SSLv3,SSLv2Hello"`

The issue only manifests when the server runs under Java 1.7.

See also:

- http://mail/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](http://youtrack.jetbrains.com/issue/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 from `.BuildServer/plugins`) before starting upgraded TeamCity version.

Maven runner

Java older than 1.5 is no longer supported by the agent part of Maven runner. Please make sure you specify 1.6+ JVM in Maven runner settings or ensure `JAVA_HOME` points to such JVM.

NUnit and MSTest Tests

If you had NUnit or MSTest tests configured in TeamCity UI (sln and MSBuild runners), the settings are extracted from the runners and converted to a new runner of corresponding type.

Please note that implementation of tests launching has changed and this affected relative paths usage: in TeamCity 6.0 the working directory and all the UI-specified wildcards are resolved based on the build's [checkout directory](#), while they used to be based on the directory containing `.sln` file. Simple settings are converted on TeamCity upgrade, but you might need to verify the runners contain appropriate settings.

"%" Escaping in the Build Configuration Properties

Now, two percentage signs (%%) in values defined in Build Configuration settings are treated as escape for a single percentage sign. Your existing settings are converted on upgrade to preserve functioning like in previous versions. However, you might need to review the settings for unexpected "%" sign-related issues.

.Net Framework Properties are Reported as Configuration Parameters

In previous TeamCity versions, installed .Net Frameworks, Visual Studios and Mono were reported as System Properties of the build agents. This made the properties available in the build script.

In order to reduce number of TeamCity-specific properties pushed into the build scripts, the values are now reported via Configuration Parameters (that is, without "system." prefix) and are not available in the build script by default. They still be used in the Build Configuration settings via %-references by their previous names, just without "system." prefix.

Ipr runner is deprecated in favor of IntelliJ IDEA Project runner

Runner for IntelliJ IDEA projects was completely rewritten. It is not named "IntelliJ IDEA Project" runner. Previously available Ipr runner is also preserved but is marked as deprecated and will be removed in one of the further major releases of TeamCity. It is highly recommended to migrate your existing build configurations to the new runner.

Please note that the new runner uses different approach to run tests: you need to have a shared Run Configuration created in IntelliJ IDEA and reference it in the runner settings.

Cleanup for Inspection and Duplicates data

Starting from 6.0 Inspection and Duplicates reports for the builds are cleaned when build is cleaned from history, not when build's artifacts are cleaned as it used to be.

Inspection and Duplicates runners require Java 1.6

"Inspections" and "Duplicates (Java)" runners now require Java JDK 1.6. Please ensure Java 1.6 is installed on relevant agents and check it is specified in the "JDK home path" setting of the runners.

XML Report Validation

If you had invalid settings of "XML Report Processing" section of the build runners, you might find the Build Configurations reporting "Report paths must be specified" messages upon upgrade. In this case, please go to the runner settings and correct the configuration. (related [issue](#))

Open API Changes

See [Open API Changes](#)

Several jars in `devPackage` were reordered, some moved under `runtime` subdirectory. Please update your plugin projects to accommodate for these changes.

REST API Changes

Several objects got additional attributes and sub-elements. Please check that your parsing code still works.

Perforce Clean Checkout

All builds using Perforce checkout will do a clean checkout after server upgrade. Please note that this can impose a high load on the server in the first hours after upgrade and server can be unresponsive while many builds are in "transferring sources" stage.

Changes from 5.1.2 to 5.1.3

Path to executable in Command line runner

The [bug](#) was fully fixed. The behavior is the same as in pre-5.1 builds.

Changes from 5.1.1 to 5.1.2

Jabber notification sending errors are displayed in web UI for administrators again (these messages were disabled in 5.1.1). If you do not use Jabber notifications, please pause the Jabber notifier on the Jabber settings server settings page.

Changes from 5.1 to 5.1.1

Path to executable in Command line runner

The [bug](#) was partly fixed. The behavior is the same as in pre-5.1 builds except for the case when you have the working directory specified and have the script in both checkout and working directory. The script from the working directory is used.

Path to script file in Solution runner and MSBuild runner

The [bug](#) was fixed. The behavior is the same as in pre-5.1 builds.

Changes from 5.0.3 to 5.1

 If you plan to upgrade from version 3.1.x to 5.1, you will need to modify some dtd files in <TeamCity Data Directory>/config before upgrade, read more in the issue: [TW-11813](#)

 NCover 3 support may not work. See [TW-11680](#)

Notification templates change

Since 5.1, TeamCity uses [new template engine \(Freemarker\)](#) to generate notification messages. New default templates are supplied and customizations to the templates made prior to upgrading are no longer effective.

If you customized notification templates prior to this upgrade, please review the new notification templates and make changes to them if necessary. Old notification templates are copied into <TeamCity Data Directory>/config/_trash/_notifications directory. Hopefully, 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.vc.s.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 a `agent.sh kill` command) and kill any running `java jetbrains.buildServer.agent.Launcher` processes and start the agent again.

Changes from 2.x to 3.0

Incompatible changes

Please note that TeamCity 3.0 introduces several changes incompatible with TeamCity 2.x:

- `build.working.dir` system property is renamed to `teamcity.build.checkoutDir`. If you use the property in your build scripts, please update the scripts.
- `runAll.bat` script now accepts a required parameter: `start` to start server and agent, `stop` to stop server and agent.
- Under Windows, `agent.bat` script now accepts a required parameter: `start` to start agent, `stop` to stop agent. Note that in this case agent will be stopped only after it becomes idle (no builds are run). To force immediate agent stopping, use `agent.bat stop force` command that is available under both Windows and Linux (`agent.sh stop force`). Under Linux you can also use `agent.sh stop kill` command to stop agents not responding to `agent.sh stop force`.

Build working directory

Since TeamCity 3.0 introduces ability to configure VCS roots on per-Build Configuration basis, rather than per-Project, the default directory in which build configuration sources are checked out on agent now has generated name. If you need to know the directory used by a build configuration, you can refer to `<agent home>/work/directory.map` file which lists build configurations with the directory used by them. See also [Build Checkout Directory](#)

User Roles when upgrading from TeamCity 1.x/2.x/3.x Professional to 3.x Enterprise

When upgrading from TeamCity 1.x/2.x/3.x Professional to 3.x Enterprise for the first time TeamCity's accounts will be assigned the following roles by default:

- *Administrators* become System Administrators
- *Users* become Project Developers for all of the projects
- The *Guest* account is able to view all of the projects
- *Default user* roles are set to Project Developer for all of the projects

Changes from 1.x to 2.0

Database Settings Move

Move your database settings from the `<TeamCity installation folder>/ROOT/WEB-INF/buildServerSpring.xml` file to the `database.properties` file located in the TeamCity configuration data directory (`<TeamCity Data Directory>/config`).

See also:

[Administrator's Guide: Licensing Policy](#)

.NET inspection and duplicates

Upgrade

 Unless specifically noted, TeamCity does not support downgrade between major/minor releases (changes in first two version numbers). It is strongly recommended to [back up your data](#) before any upgrade.

TeamCity supports upgrades from any of the previous versions to the later ones. All the settings and data are preserved unless noted in the [Upgrade Notes](#).

Before Upgrade

Before upgrading TeamCity:

1. For a major upgrade, review what you will be getting in [What's New](#) (follow the links at the bottom of What's New if you are upgrading not from the previous major release)
2. [Check your license keys](#) unless you are upgrading within bugfix releases of the same major X.X version
3. [Download the new TeamCity version \(extended download options\)](#)
4. Carefully review the [Upgrade Notes](#)
5. Consider probing the upgrade on a [test server](#)
6. If you have non-bundled plugins installed, check plugin pages for compatibility with the new version and upgrade/uninstall the plugins if necessary

To upgrade the server:

1. [Back up the current TeamCity data](#)
2. Perform the upgrade steps:
 - [Upgrading Using Windows Installer](#)
 - [Manual Upgrading on Linux and for WAR Distributions](#)

If you plan to upgrade a production TeamCity installation, it is recommended to install a [test server](#) and check its functioning in your environment before upgrading the main one.

Licensing

Before 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](#).

Typically all the bugfix updates (releases with changes in the third release number) use the same effective release date (that of the major/minor release).

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.

Unless specifically noted, downgrades with preserving the data are not possible with changing major.minor version and are possible within bugfix releases (without changing major.minor version).

The general policy is that bugfix updates (changes in the Z part of X.Y.Z TeamCity version) do not change data format, so you can freely upgrade/downgrade within the bugfix versions. However, when upgrading to the next major or minor version (changed X or Y in X.Y.Z TeamCity version), you will not be able to downgrade with the data preservation: you will need to restore a backup of the appropriate version.

On upgrade, all the TeamCity configuration settings and other data are preserved unless noted in [Upgrade Notes](#). If you have customized TeamCity installation (like Tomcat server settings change), you will need to repeat the customization after the upgrade.

Agents connected to the server are upgraded automatically.



Important note on TeamCity data structure upgrade

TeamCity server stores its data in the database and in [TeamCity Data Directory](#) on the file system. Different TeamCity versions use different data structure of the database and data directory. Upon starting newer version of TeamCity, the data is kept in the old format until you confirm the upgrade and data conversion on the Maintenance page in the web UI. Until you do so, you can back up the old

data; however, once the upgrade is complete, the data is updated to a newer format.

Once the data is converted, downgrade to the previous TeamCity versions which uses different data format is not possible!

There are several important issues with data format upgrade:

- Data structure downgrade is not possible. Once newer TeamCity version changes the data format of database and data directory, you cannot use this data to run an older TeamCity version. Please ensure you [backup](#) the data before upgrading TeamCity.
- Both the database and the data directory should be upgraded simultaneously. Ensure that during the first start of the newer server it uses the correct [TeamCity Data Directory](#) that in its turn has the correct database [configured](#) in the [<TeamCity Data Directory>\config\database.properties](#) file. Also make sure the data directory is complete (e.g. all the build logs, artifacts, etc. are in place), no data directory content supports copying from the data directory of the older server versions.

If you did performed inconsistent upgrade accidentally, check the [recovery instructions](#).

Upgrading Using Windows Installer

1. [Create a backup](#). When upgrading from TeamCity 6.0+ you will also have a chance to create a backup with the "basic" profile on the [TeamCity Maintenance Mode](#) page on the updated TeamCity start.
2. Note the username used to run the TeamCity server. You will need it during the new version installation.
3. (optional as these will not be overwritten by the upgrade) If you have any of the Windows service settings customized, store them to repeat the customizations later. The same applies to customizations of the bundled Tomcat server (like port, https protocol, etc.) and JRE, if any.
4. Run the new installer. Confirm uninstalling the previous installation. After uninstalation finishes, make sure the [TeamCity server installation directory](#) does not contain any non-customized files. If there are any, backup/remove them before proceeding with installation. **Since TeamCity 9.1**, the TeamCity uninstaller ensures proper uninstalation (see a related issue [TW-39884](#)).



Since TeamCity 9.1, the main server configuration file [<TeamCity Home Directory>/conf/server.xml](#) is updated automatically when there were no changes to it since the last installation. If modification were made, the installer will detect them and backup the old `server.xml` file displaying a warning about the overwrite and the backup file location.

5. When prompted, specify the [<TeamCity data directory>](#) used by the previous installation.
6. (optional as these will not be overwritten by the upgrade) Make sure you have the external database driver [installed](#) (this applies only if you use external database).
7. Check and restore any customizations of Windows services and Tomcat configuration that you need. When upgrading from versions 7.1 and earlier, make sure to transfer the [server memory setting](#) to the [environment variables](#).
8. If you use customized Log4J configuration in `conf\teamcity-server-log4j.xml` and want to preserve those, compare and merge `conf\teamcity-server-log4j.xml.backup` created by installer from existing copy with the default file saved with default name.
9. Start up the TeamCity server (and agent, if it was installed together with the installer).
10. 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 encounter errors which cannot be resolved, make sure old TeamCity is not running/does not start on boot, restart the machine and repeat the installation procedure.

Manual Upgrading using .tar.gz or .war Distributions

Please note that it is recommended to use `.tar.gz` or `.exe` TeamCity distribution. Using `.war` is not a recommended way to install TeamCity.

1. [Create a backup](#).
2. Backup files customized since previous installation (most probably `[TOMCAT_HOME]/conf/server.xml`)
3. Remove old installation files (the entire [TeamCity home directory](#) or `[TOMCAT_HOME]/webapps/TeamCity/*` if you are installing from a [war](#) file). It's advised to backup the directory beforehand.
4. Unpack the new archive to the location where TeamCity was previously installed.
5. If you use 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.
6. Restore customized settings backed up in step 2 above. If you have customized `[TOMCAT_HOME]/conf/server.xml` file, apply your changes into the appropriate sections of the default file.
7. Make sure previously configured [TeamCity server startup properties](#) (if any) are still actual.
8. Start up the TeamCity server.
9. Review the [TeamCity Maintenance Mode](#) page to make sure there are no problems encountered, and confirm the upgrade by clicking corresponding button. Only after that all the configuration data and database scheme are updated by TeamCity converters.

IDE Plugins

Generally, versions of IntelliJ IDEA TeamCity plugin, Eclipse TeamCity plugin and Visual Studio TeamCity Addin should be the same as the TeamCity server version. Users with not matching plugin versions get a message on attempt to login to TeamCity server with not matching

version.

The only exception is that TeamCity versions 9.0 - 9.1.x use compatible protocol and any plugin of these versions can be used with any server of these versions. Updating IDE plugin to the matching server version is still recommended.

Upgrading Build Agents

- #Automatic Build Agent Upgrading
- Upgrading Build Agents Manually
 - #Upgrading the Build Agent Windows Service Wrapper

Automatic Build Agent Upgrading

On starting TeamCity server (and updating agent distribution or plugins on the server), TeamCity agents connected to the server and [correctly installed](#) are automatically updated to the version corresponding to the server. This occurs for both server upgrades and downgrades. If there is a running build on the agent, the build finishes. No new builds are started on the agent unless agent is up to date with the server.

The agent update procedure is as follows: The agent (agent.bat, agent.sh, or agent service) will download the current agent package from the TeamCity server. When the download is complete and the agent is idle, it will start the upgrade process (the agent is stopped, the agent files are updated, and 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 an 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 Database Creation

TeamCity Data Upgrade

When upgrading your TeamCity instance, the newly installed version of TeamCity checks if the TeamCity data directory and database use the old data format when it is started for the first time. If the newer version requires data conversion, this page is displayed with data format details. Please, review them carefully.

! If you haven't backed 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.

{anchor:dbCreation}

TeamCity Database Creation

If you see this screen when [settings up TeamCity with an external database](#) or [after migrating to an external database](#), click **Proceed** to create a new database for TeamCity.

See also:

- [Installation and Upgrade: Upgrade](#)
- [Administrator's Guide: TeamCity Data Backup](#)

Setting up an External Database

TeamCity stores build history, users, build results and some run time data in an SQL database. See also description of what is stored where on [Manual Backup and Restore](#) page.

This page covers external database setup for the first use with **TeamCity 8.1 and above**. For earlier versions, select the documentation corresponding to your TeamCity version.

If you evaluated TeamCity with the internal database, please refer to [Migrating to an External Database](#).

On this page:

- Default Internal Database
- Selecting External Database Engine
 - Supported Databases
- General Steps
- Database-specific Steps
 - MySQL
 - On MySQL server side
 - On TeamCity server side (with MySQL)
 - PostgreSQL
 - On PostgreSQL server side
 - On TeamCity server side (with PostgreSQL)
 - Oracle
 - On Oracle server side
 - On TeamCity server side (with Oracle)
 - Microsoft SQL Server
 - On MS SQL server side
 - On TeamCity server side (with MS SQL)
 - Additional connection settings:
- Database Configuration Properties

Default Internal Database

On the first TeamCity run, using an internal database based on the HSQLDB database engine is suggested by default. The internal database suits evaluation purposes only; it works out of the box and requires no additional setup.

However, we strongly recommend using an external database as a back-end TeamCity database in a production environment. An external database is usually more reliable and provides better performance: the internal database may crash and lose all your data (e.g. on the "out of disk space" condition). Also, the internal database may become extremely slow on large data sets (say, database storage files over 200Mb). Please also note that our support does not cover any performance or database data loss issues if you are using the internal database.

In short, **do not EVER use internal HSQLDB database for production TeamCity instances**.

Selecting External Database Engine

As a general rule you should use the database that better suits your environment and that you can maintain/configure better in your organization. While we strive to make sure TeamCity functions equally well under all of the supported databases, issues can surface in some of them under high TeamCity-generated load.

You may also want to estimate [the required database capacity](#).

Supported Databases

TeamCity supports the following databases:

- MySQL
- PostgreSQL
- Oracle
- MS SQL

General Steps

1. Configure the external database to be used by TeamCity (see the [database-specific sections below](#)).
2. Run the TeamCity server for [the first time](#).
3. Select an external database to be used and specify the database connection settings.

If required, you can later manually modify your [database connection settings](#).

 Note that TeamCity creates its own database schema on the first start and actively modifies it during the upgrade. The schema is not changed when TeamCity is working normally.

The user account used by TeamCity should have **permissions to create new, modify and delete existing tables** in its schema, in addition to usual **read/write permissions** on all tables.

4. You may also need to [download the JDBC driver for your database](#).

Due to licensing terms, TeamCity does not bundle driver jars for external databases. You will need to download the Java JDBC driver and put the appropriate .jar files (see driver-specific sections below) from it into the <TeamCity Data Directory>/lib/jdbc directory. Please note that the .jar files should be compiled for the Java version not greater than the one used to run TeamCity, otherwise you might see "Unsupported major.minor version" errors related to the database driver classes.

Database-specific Steps

The section below describes the required configuration on the database server and the TeamCity server.

MySQL

Supported versions

On MySQL server side

Recommended database server settings:

- use InnoDB storage engine
- use [UTF-8 character set](#)
- use case-sensitive collation
- [see also recommendations for MySQL server settings](#)

The MySQL user account that will be used by TeamCity must be granted all permissions on the TeamCity database. This can be done by executing the following SQL commands from the MySQL console:

```
create database <database-name> collate utf8_bin;
create user <user-name> identified by '<password>';
grant all privileges on <database-name>.* to <user-name>;
grant process on *.* to <user-name>;
```

On TeamCity server side (with MySQL)

JDBC driver installation:

1. Download the MySQL JDBC driver from <http://dev.mysql.com/downloads/connector/j/>. If the MySQL server version is 5.5 or newer, the JDBC driver version should be 5.1.23 or newer.
2. Place mysql-connector-java-*-.bin.jar from the downloaded archive into the <TeamCity Data Directory>/lib/jdbc. Proceed with the TeamCity setup.

PostgreSQL

Supported versions

On PostgreSQL server side

1. Create an empty database for TeamCity in PostgreSQL.
 - Make sure to set up the database to use UTF8.
 - Grant permissions to modify this database to the user account used by TeamCity to work with the database.
2. see also recommendations for PostgreSQL server settings

TeamCity does not specify which schema will be used for its tables. By default, PostgreSQL creates tables in the 'public' schema ('public' is the name of the schema). TeamCity can also work with other PostgreSQL schemas. To switch to another schema, do the following:
Create a schema named exactly as the user name: this can be done using the pgAdmin tool or with the following SQL:

```
create schema teamcity authorization teamcity;
```

The schema has to be empty (it should not contain any tables).

On TeamCity server side (with PostgreSQL)

1. Check your JRE version to determine which JDBC driver is required. If you are using Java 1.6 to run TeamCity server, you need the JDBC4 version. For Java 1.7 (recommended) or 1.8 (recommended **since TeamCity 9.1**), use the JDBC41 version.
2. Download the required [PostgreSQL JDBC driver](#) and place it into the <TeamCity Data Directory>/lib/jdbc. Proceed with the TeamCity setup.

Oracle

Supported versions

On Oracle server side

1. Create an Oracle user account/schema for TeamCity.



TeamCity uses the primary character set (char, varchar, clob) for storing internal text data and the national character set (nvarchar2, nvarchar, nclob) to store the user input and data from external systems, like VCS, NTLM, etc.

- Make sure that the national character set of the database instance is UTF or Unicode.
- Grant the CREATE SESSION, CREATE TABLE, permissions to a user whose account will be used by TeamCity to work with this database. TeamCity, on the first connect, creates all necessary tables and indices in the user's schema. (Note: TeamCity never attempts to access other schemas even if they are accessible)



Make sure TeamCity user has quota for accessing table space.

On TeamCity server side (with Oracle)

1. Get the Oracle JDBC driver.

Supported driver versions are 11.1 and higher.

Place the following files:

- ojdbc6.jar
- ora18n.jar (can be omitted if missing in the driver version) into the <TeamCity Data Directory>/lib/jdbc directory.



The Oracle JDBC driver must be compatible with the Oracle server.

It is strongly recommended to locate the driver in your Oracle server installation. Contact your DBA for the files if required. Alternatively, download the Oracle JDBC driver from the [Oracle web site](#). Make sure the driver version is compatible with your Oracle server.

2. Proceed with the TeamCity setup.

Microsoft SQL Server

Supported versions

On MS SQL server side

 TeamCity uses the primary character set (char, varchar, text) for storing internal text data and the national character set (nchar, nvarchar, ntext) to store the user input and data from external systems, like VCS, NTLM, etc.

1. Create a new database. As the primary collation, it is recommended to use the collation corresponding to your locale. We also suggest using the case-sensitive collation (collation name ending with '_CS_AS'), which 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. It's recommended to setup not less then 1Gb.

On TeamCity server side (with MS SQL)

1. Download the MS **sqljdbc** package from the Microsoft Download Center and unpack it.
2. Copy the sqljdbc4.jar from the just downloaded package into the [TeamCity Data Directory](#)/lib/jdbc directory. Proceed with the TeamCity setup.

Additional connection settings:

To specify additional settings:

1. Stop the TeamCity server.
2. Configure additional settings:
 - If you use a named instance, you need to manually modify the <TeamCity Data Directory>\config\database.properties file and specify the instance name in the connection URL as follows:

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

- If you prefer to use **MS SQL integrated security (Windows authentication)**, you should also install sqljdbc_auth.dll from the driver package. Incorrect installation of the DLLs often results in "This driver is not configured for integrated authentication" error
 - a. Determine the bitness of [TeamCity server Java/JVM](#) in use. You might want to use JVM with the same bitness as Windows. Note that you will need to perform manual [switch to 64-bit JVM](#) as by default 32 bit JVM is bundled.
 - b. Copy the <sql_jdbc_home>/enu/auth/x86/sqljdbc_auth.dll (in case of 32-bit Java) or <sql_jdbc_home>/enu/auth/x64/sqljdbc_auth.dll (in case of 64-bit Java) file into a directory and specify the directory in "-Djava.library.path=DIRECTORY_WITH_DLL" [TeamCity server JVM option](#). Reportedly, as alternative to adding the JVM option, you can also copy the .dll into Windows/System32 directory and ensure that there are no other sqljdbc_auth.dll files in your system (in the PATH-listed directories).
 - c. In the <TeamCity Data Directory>\config\database.properties file, specify the connection URL (with **no user names or passwords**) as follows:

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

More details about setting up integrated security for MS SQL native JDBC driver can be found in Microsoft documentation for [MS SQL 2005](#) and [MS SQL 2008](#).

3. Start the TeamCity server.

Database Configuration Properties

The database connection settings are stored in <TeamCity Data Directory>\config\database.properties file. The file is a Java properties file. You can modify it to specify required properties for your database connections.

TeamCity uses Apache DBCP for database connection pooling. Please refer to <http://commons.apache.org/dbcp/configuration.html> for detailed description of configuration properties.

 For all supported databases there are template files with database-specific properties located in the <TeamCity Data Directory>/config directory. The files have the following naming format: database.<database_type>.properties.dist and can be used as a reference on the required settings.

See also:

[Installation and Upgrade: 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.



If you see the **TeamCity Maintenance** screen, click the "*I'm a server administrator, show me the details*" link and enter **Maintenance Authentication Token**. Follow the instructions to create a new TeamCity database.

Full Migration

These steps describe switching to another database preserving all data. The TeamCity migration tool, the `maintainDB` command line utility, is available for this purpose.

The `maintainDB.[cmd|sh]` shell/batch script is located in the <TeamCity Home>/bin directory and is used for migrating as well as for **backing up** and **restoring** TeamCity data.

The utility is only available in the TeamCity .tar.gz and .exe distributions. If you installed TeamCity using the .war distribution and need to perform data migration, use the tool from the .tar.gz distribution.

TeamCity supports **HSQLDB**, **MySQL**, **Oracle**, **PostgreSQL** and **Microsoft SQL Server**; the migration is possible between any of these databases.



The target database must be empty before the migration process (it must NOT contain any tables).



If an error occurs during migration, do not use the new database as it may result in the database data corruption or various errors that can uncover later. In case of an error, investigate the reason logged into the console or in the migration logs (see below), and, if a solution is found, clean the target database and repeat the migration process.

To migrate all your existing data to a new external database:

1. Create and configure an external database to be used by TeamCity and install the database driver into TeamCity. **Do not modify any TeamCity settings at this stage.**
2. Shut down the TeamCity server.
3. Create a temporary properties file with a custom name (for example, `database.<database_type>.properties`) for the target database using the corresponding template (<TeamCity Data Directory>/config/database.<database_type>.properties.dist). Configure the properties and place the file into any temporary directory. **Do not modify the original database.<database_type>.properties file.**
4. Run the `maintainDB` tool with the `migrate` command and specify the absolute path to the newly created target database properties file with the `-T` option:

```
maintainDB.[cmd|sh] migrate [-A <path to TeamCity Data Directory>] -T <path to database.properties file>
```



If you have the `TEAMCITY_DATA_PATH` environment set (pointing to the **TeamCity Data Directory**), you do not need the `-A <path to TeamCity Data Directory>` parameter of the tool.

Upon the successful completion of the database migration, the temporary file will be copied to (<TeamCity Data Directory>/config/database.properties) file which will be used by TeamCity. The temporary file can be safely deleted.

If you are migrating between external databases, the original `database.properties` file for the source database will be replaced with the file specified via the `-T` option. The original `database.properties` file will be automatically re-named to `database.properties.before.<timestamp>`.

5. Start the TeamCity server. This must be the same TeamCity version that was run last time (TeamCity `upgrade` must be performed as a separate procedure).

After you make sure the migration succeeded, you **may** delete the old HSQLDB files: `buildserver.*` to remove the no longer needed internal storage data.

Backup and Restore

You can **create a backup** and then **restore** it using different target database settings. You will probably need to specify restore options to restore only the database data.

Troubleshooting

- Extended information during migration execution is logged into the `logs\teamcity-maintenance.log` file. Also, `logs\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: `jdb:hsqldb:hsq1://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

- 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 [Changes](#),
- highlight such builds on the Projects page if the appropriate [option](#) is selected,
- notify you on such builds when the **Builds affected by my changes** option is selected in [notifications settings](#).

Customizing UI

On the **General** tab of the [My Settings & Tools](#) page, you can customize the following UI settings:

- Highlight my changes and investigations: Select to highlight builds that include your changes (changes committed by a user with the VCS username provided in the [Version Control Username Settings](#) section) and problems you were assigned to investigate on the [Projects](#) page, Project Home Page, Build Configuration Home Page.
- Show date/time in my timezone: Check the option, if you want TeamCity to automatically detect your time zone and show the date and time (for example, build start, vcs change time, etc.) according to it.
- Show all personal builds
- Add builds manually triggered by you to your [favorites](#).

Viewing your Roles and Permissions

1. In the top right corner of the screen, click the arrow next to your username, and select **My Settings & Tools** from the drop-down list.
 - To view the list of user groups you are in, go to the **Groups** tab.
 - To view your roles and permissions in different projects, go to the **Roles** tab. Note, that roles are assigned to the user by the system administrator.

See also:

[Concepts: Role and Permission](#)

[User's Guide: Viewing Your Changes | Subscribing to Notifications](#)

Subscribing to Notifications

TeamCity provides a wide range of notification possibilities to keep developers informed about the status of their projects. Notifications can be sent by e-mail, Jabber/XMPP instant messages or can be displayed in the IDE (with the help of TeamCity plugins) or the Windows system tray (using TeamCity Windows Tray Notifier).

- [Subscribing to 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 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:

1. In the top right corner of the screen, click the arrow next to your username, and select **My Settings & Tools** from the drop-down list. Open the **Notification Rules** tab.
2. Click the required notifications type:
 - **Email Notifier:** to be able to receive email notifications, your email address must be specified 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.
3. Click **Add new rule** and specify the rule in the dialog. The notification rules are comprised of two parts: [what you will watch](#) and [which events you will be notified about](#). See the details below.



Email and Jabber notifications are sent only if the System Administrator has configured 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. Since TeamCity 9.0 , you can use the branch filter here to receive alerts only on the builds from the specified branches.
Builds from the selected 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 . Since TeamCity 9.0 , you can use the branch filter here to receive alerts only on the builds from the specified branches; you can also limit notifications to your favorite builds .
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. Since TeamCity 9.0 , you can use the branch filter here to receive alerts only on the builds from the specified branches; you can also limit notifications to your favorite builds .
System wide events	Select to be notified about system wide events.

Which Events Will Trigger Notifications

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.
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.
Build starts	Check this option to receive notifications as soon as a build of the specified build configurations starts.
Build fails to start	Check this option to receive notifications when a build of the specified build configurations fails to start .
Build is probably hanging	Check this option to receive notifications when TeamCity identifies a build of the specified build configurations as hanging .
Investigation is updated	Check this option to receive notifications on changing a build configuration or test investigation status, e.g. someone is investigating the problem, or problems were fixed, or the investigator changed.

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.

 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

Monitoring the quality of the codebase is essential for a development team: a project developer needs to see whether his/her commit brought a build failure or not; for a project leader it is important to detect the code at fault for a build failure to be able to have the situation rectified early, so other members of the team are not inconvenienced.

The **Changes** page of the TeamCity Web UI allows you to review the commits made by all TeamCity users and see how they have affected builds. You can filter the results with the user selector on the page.



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

By default, the page does not show the commits to the build configurations hidden by the current user on the Projects dashboard. To remove this filter and view all build configurations, deselect the **Hide configurations excluded from my Projects** box.

Each change now has a new pie-chart icon with pie slices showing the relative size of pending, successful, as well as old and new problematic builds affected by the change. Hovering over/clicking the pie-chart icon gives a visual representation of how the user commit has affected different builds. Builds with new/critical problems are listed by default. Expanding the change or clicking the *See builds* link lists all builds with the change.

The screenshot shows the TeamCity 'Changes' page. At the top, there are tabs for 'Projects', 'Changes', 'Agents', 'File', and 'Build Queue'. Below the tabs, there's a search bar and a link to 'Add Available'. The main area is titled 'Changes' and shows a list of build changes. Each item includes a small icon, the change type (e.g., 'Build', 'File'), the date ('24 Jun 16 10:19'), and a detailed description. A vertical red bar highlights the third item in the list.

Change Type	Date	Description
Build	24 Jun 16 10:19	Message for build configuration template detection: A build status listener, 'Other build builds'
Build	24 Jun 16 10:22	User(s) trigger when project build changes: If user build X runs, 'Other build builds'
Build	24 Jun 16 10:49	Builds with noncritical problems: TeamCity: Helper file for Interactive Tests - Build Runner Tests - NFT Listener
Build	24 Jun 16 10:49	TeamCity: Helper file for Interactive Tests - Build Runner Tests - NFT Listener
Build	24 Jun 16 10:49	TeamCity: Helper file for Interactive Tests - Build Runner Tests - NFT Listener - Stable - Grade - Built by build tests
File	24 Jun 16 10:49	Period builds without new conditions
Comment/Issue	24 Jun 16 10:49	Planning availability
File	24 Jun 16 10:49	Pending builds
Build	24 Jun 16 10:49	Other builds

From this page you can:

- View all commits and changes included into personal builds.
- View how changes have affected the builds.
- See whether there are new failed tests caused by changes.
- Navigate to the issue tracker, if the [issue tracker integration is configured](#).
- Open possibly problematic files in your IDE: the option is available if the plugin for this IDE is installed and you are logged in to TeamCity from within this IDE.
- Navigate to change details.
- View detailed data for each change on the dedicated tabs. To switch between tabs for the currently selected change, use Tab/Shift+T ab or mnemonics: 'T' for tests, 'B' for builds, 'F' for files.
- View builds with any of the changes. By default, successful and pending build configurations are hidden from display. Uncheck the corresponding box to view all builds with the change.

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

See also:

Concepts: [Build State](#) | [Change](#)

Working with Build Results

In TeamCity a build goes through several states:

- upon some event the build trigger adds the build to the queue where the build stays waiting for a free agent
- the build starts on the agent and performs all configured build steps
- the build finishes and becomes a part of the build history of this build configuration.

In TeamCity all information about a particular build, whether it is queued, running or finished, is accumulated on the **build results** page.

Besides providing the build information, this page enables you to:

- [run a custom build](#) using the **Run...** button
- use the **Actions** menu to do the following:
 - add a build to [favorites](#)
 - add a comment
 - pin the build
 - tag the build
 - change the build status, marking the build as [failed](#) or [successful](#)
 - [label this build sources](#)
 - remove the build
- [edit the configuration settings](#)

Build Details

The build results page can be accessed the Build Configuration home page and from various places in the TeamCity web UI where a build number or build status appears as a link.

Some data is accessible only after the build is finished, some details like [Changes](#), [Build parameters](#), and [Dependencies](#) are also applicable to the build while it is waiting in the queue.

Results	Artifacts	Changes	Started
#53161 Success	None	Victor Bedrosyan [1]	01 Aug 13
#53160 Success	None	Victor Bedrosyan [1]	01 Aug 13
#53149 Success	None	Kirill Maximov [1]	01 Aug 13
#53148 Success	None	Victor Bedrosyan [1]	01 Aug 13

The following build information is available on the page:

- Build Overview
 - Tests
- Changes
- All Tests
 - Test History
 - Test Duration Graph
- Build Log
- Parameters
- Dependencies
- Related Issues
- Build Artifacts
- Code Coverage Results
- Code Inspection Results
- Duplicates
- Maven Build Info
- Internal Build ID

Depending on the build runners enabled as your build steps, the number of tabs on the page and the information on the **Overview** tab may vary.

Build Overview

The **Overview** Tab displays general information about the build, such as the build duration, the agent used, trigger, dependencies, etc. If a build is queued, the tab displays the position of the build in the queue, the time the build is supposed to start, etc.

If a build is running, the tab displays the build progress. You can also stop a running build using the corresponding link on the **Overview** tab or the appropriate option from the **Actions** button drop-down.

If another build of this same build configuration is concurrently running, a small window appears on the **Overview** tab with the following information:

- The build results of that build linking to the build results page
- A changes link with a drop-down list of changes included in that build
- The build number, time of start, and a link to the agent it is running on.

If a build is probably hanging, the corresponding notification is displayed at the top of the **Overview** tab. In this case TeamCity provides a link to view the process tree of the running build and the thread dumps of each Java or .Net process in a separate frame. If the process is not Java or .Net, its command line is retrieved. The possibility to view the thread dump is supported for the following platforms:

- Windows, with JDK 1.3 and higher
- Windows, with JDK 1.6 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.

Tests

Successful or failed tests in this build (if any) are displayed on the **Overview** tab:

For each failed test, you can view its stacktrace, the diff between the expected and actual values, jump to the test [history](#), assign a team member to investigate its failure, open the test in your IDE and/or start fixing it right away.

To view all tests related to the build, use the dedicated **Tests** tab. [Learn more](#).

Changes

From the **Changes** tab you can:

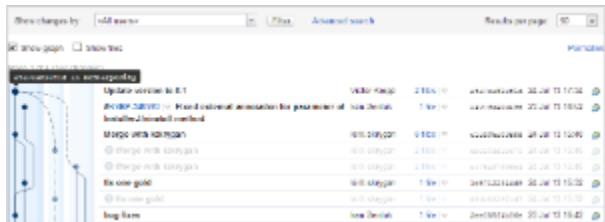
- Review all changes included in the build with the corresponding [revisions](#) in version control
- [Label the build sources](#) (prior to TeamCity 8.1)
- Configure the [VCS settings](#) of the build configuration (if you have enough permissions).

For each change on this page you can:

- Explore the change in details
- Navigate to the **Change Details** by clicking a changed file link
- [Trigger a custom build](#) with this change
- Download patch
- Download patch to your IDE
- Review the change in an [external change viewer](#), if configured by the administrator.

The **Changes** tab provides advanced filtering capabilities for the list of changes.

Enabling **Show graph** displays the changes as a graph of commits to the VCS roots related to this build.



The graph appears on the left of the list and allows you to get the view of the changes with a variable level of detailing. You can:

- View the VCS roots which were changed in this build, each of the roots being represented as a bar.
- Navigate to a graph node to display the VCS root revision number.
- Click a bar to select a single VCS root. The changes not pertaining to this root are grayed out.
- If there have been merges between the branches of the repository, the graph displays them. To collapse a bar, navigate to its darker area and click it to hide history of merges. The dotted line will indicate that the bar is expandable.
- If your VCS root has subrepositories (marked S in the list of changes), navigate to a node in the parent to see which commits in subrepositories are referenced by this revision in the parent.

You can select to view the modified files by checking the **Show files** box. Clicking a file name opens the diff viewer.

All Tests

To view all the tests for a particular build, open the build results page, and navigate to the **Tests** tab.
On this page:

Item	Description
------	-------------

Download all tests in CSV	Click the link to download a file containing all the build tests results.
Filtering options	<p>Use this area to filter the test list:</p> <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

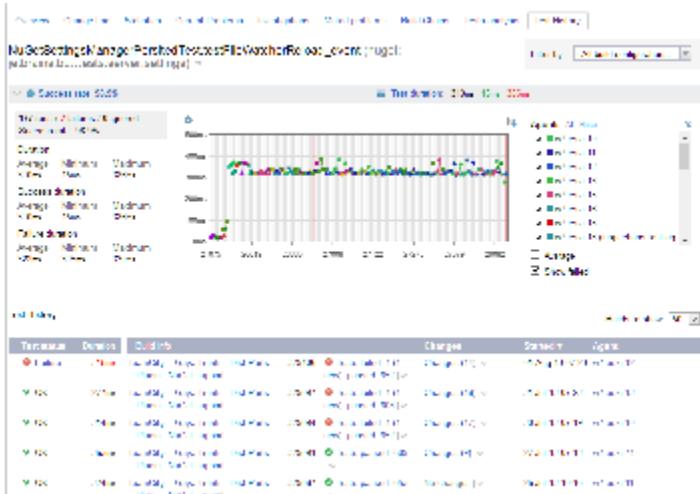
To navigate to the history of a particular test, click the arrow next to the test name and select **Test History** from the drop-down. There are several places where tests are listed and from where you can open Test History.

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:

By clicking the links in the **Coverage Breakdown** section, you can drill-down to display statistics for different scopes, e.g. Namespace, Assembly, Method, and Source Code.

Code Inspection Results

If configured, the results of the **Code Inspection** build step are shown on the **Code Inspection** tab.

Use the left pane to navigate through the inspection results; the filtered inspections are shown in the right pane.

- Switch from the **Total** to **Errors** option, if you're not interested in warnings.
 - Use the scope filter to limit the view to the specific directories. This makes it easier for developers to manage specific code of interest.
 - Use the inspections tree view under the scope filter to display results by specific inspection.
 - Note that TeamCity displays the source code line number that contains the problem. Click it to jump the code in your IDE.

Duplicates

If your build configuration has Duplicates build runner as one of the build steps, you will see the **Duplicates** tab in the build results.

The tab consists of:

- A list of duplicates found. The **new only** option enables you to show only the duplicates that appeared in the latest build.
 - A list of files containing these duplicates. Use the left and right arrow buttons to show selected duplicate in the respective pane in the lower part of the tab.
 - Two panes with the source code of the file fragments that contain duplicates.
 - Scope filter in the the upper-left corner lists the specific directories that contain the duplicates. This filtering makes it easier for developers to manage the code of interest.

Maven Build Info

For each Maven build the TeamCity agent gathers Maven specific build details, which are displayed on the **Maven Build Info** tab of the build results after the build is finished.

Internal Build ID

In the URL of the build result page you can find parameter "buildId" with a numeric value. This number is internal build id uniquely identifying the build in the TeamCity installation.

You might need this ID when constructing URL manually. For example for REST API, downloading artifacts.

See also:

Concepts: Build Log | Build Artifact | Change | Code Coverage

User's Guide: Investigating Build Problems | Viewing Tests and Configuration Problems

Administrator's Guide: Creating and Editing Build Configurations

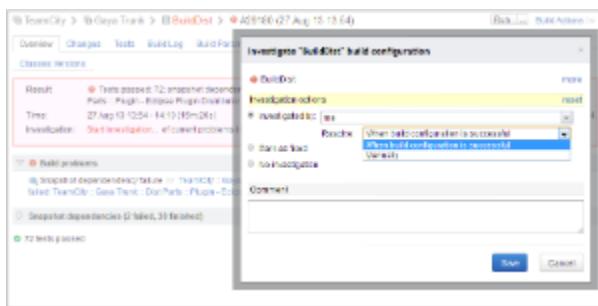
Investigating Build Problems

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.
3. Select the condition to remove the investigation: when the build configuration becomes green or the test passes (default) or specify that an investigation should be resolved manually.
 Manual resolution is useful with so called "flickering tests", i.e. when a test fails from time to time and the "green" status of a build is not an indicator of the problem resolution.
4. Save your investigation.

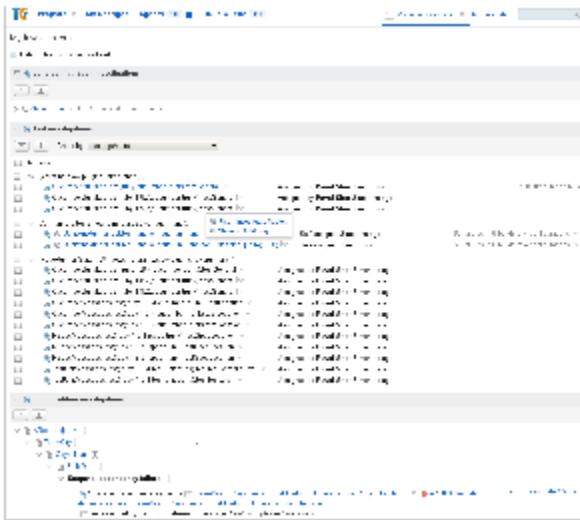


To investigate a problem in more than one build configuration, click **more** and select the desired build configurations.

To 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.

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

See also:

[Concepts: Build Configuration Status](#)

[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 and enables you to:

- run a custom build using the **Run...** button
- using the **Actions** menu
 - pause triggers
 - check for pending changes
 - enforce clean checkout
 - assign an investigation
- edit the configuration settings

The build configuration details are separated into several tabs whose number may vary depending on your server or project configuration, e.g. 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** of the build configuration, and, if applicable:
 - the number of [queued builds](#)
 - for a running build - the progress details with the [Stop](#) option to terminate the build
 - for a failed build - the number and [agent](#), etc.
- the **Recent History** section lists builds of the current build configuration

History

Displays [Build History](#) on a separate page and allows filtering builds by build agents, [tagging builds](#) and filtering them by tags (if available).

Change Log

By default, lists changes from builds finished during the last 14 active days. Use the [show all](#) link to view the complete change log.

The page shows the change log with its graph of commits to the monitored branches of all VCS repositories used by the current build configurations and the repositories used by the [dependencies](#) and [dependent configurations](#) of the current configuration.

Statistics

Displays the collected statistical data as [visual charts](#).

Compatible Agents

Lists all authorized agents. Agents meeting [Agent Requirements](#) are listed as compatible. For each incompatible agent, the reason is provided. The agents belonging to the [pool\(s\)](#) associated with the current project are listed first.

Pending Changes

Lists [changes](#) waiting to be included in the next build on a separate page.

Settings

Lists the current [build configuration settings](#) on a separate page.

Statistic Charts

To help you track the condition of your projects and individual build configurations over time, TeamCity gathers statistical data across all their history and displays it as visual charts. The statistical charts can be divided into the following categories:

- Project-level statistics available at the [Project home page | Statistics](#) tab.
- Build Configuration-level statistics available at the [Build Configuration home page | Statistics](#) tab.

Regardless of the selected statistics level in the Statistics tab, you can:

- Use the branch filter to view the results from the specified branches only (available since [TeamCity 9.0](#))
- 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. Note that **since TeamCity 9.0**, the way TeamCity counts tests **has changed**.
- **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. You can search builds by build number, tag, build configuration name and other different parameters specifying one or several keywords. You can also search for builds by text in build logs, and, **since TeamCity 9.1.**, by the [external id](#) of a build configuration.

On this page:

- [Search Query](#)
 - Differences from Lucene Syntax
 - Performing Fuzzy Search
 - Boolean Operators and Wildcards Support
- Complete List of Available Search Fields, Shortcuts, and Keywords
 - [Search Fields](#)
 - [Shortcuts](#)
 - Using Double-Colon

- "Magic" Keywords
- Search by Build Log

Search Query

In TeamCity you can search for builds using the [Lucene query syntax](#); however, a TeamCity search query has two major differences described [below](#).

To narrow your search and get more precise results, use the available search fields - indexed parameters of each build. For complete list of available search fields (keywords), refer to [this section of the page](#).

Differences from Lucene Syntax

When using a search query in TeamCity, mind the following major differences from the Lucene native syntax:

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

Performing Fuzzy Search

You also have a possibility to perform fuzzy search using the tilde (~) symbol at the end of a single word term to search for items with similar spelling.

Boolean Operators and Wildcards Support

You can combine multiple terms with Boolean operators to create more complex search queries. In TeamCity, you can use AND, "+", OR, NOT and "-".

 When using Boolean operators, type them ALL CAPS.

- AND (same as a plus sign). All words that are linked by the "AND" are included in the search results. This operator is used by default.
- NOT (same as minus sign in front of a query word). Exclude a word or phrase from search results.
- OR operator helps you to fetch the search terms that contain either of the terms you specify in the search field.

TeamCity also supports "*" and "?" wildcards in a query.

 It is not recommended to use the asterisk (*) at the beginning of the search term as it may require a significant amount of time for TeamCity to search its database. For example, the *onfiguration search term is incorrect.

Complete List of Available Search Fields, Shortcuts, and Keywords

Search Fields

When using search keywords, use the following query syntax:

```
<search field name>:<value to search>
```

Search Field	Shortcut	Description	Example
agent		Find all builds that were run on the specified agent.	agent:unit-77, or agent:agent14*
build		Find all builds that include changes with the specified string.	build:254 or build:failed
buildLog		Find all builds that include certain text in build logs. It is disabled by default.	buildLog: "NUnit report"
changes		Find all builds that include changes with the specified string.	changes:(fix test)
committers		Find all build that include changes committed by the specified developer.	committers:ivan_ivanov
configuration	c	Find all builds from the specified build configuration.	configuration:IPR c:(Nightly Build)

file_revision		Find all builds that contain a file with the specified revision.	file_revision:5
files		Find all builds that include files with the specified file name.	files:
labels	l	Find all builds that include changes with the specified VCS label.	label:EAP l:release
pin_comment		Find all builds that were pinned and have the specified word (string) in the pin comment.	pin_comment:publish
project	p	Find all builds from the specified project.	project:Diana p:Calcutta
revision		Find all builds that include changes with the specified revision (e.g., you can search for builds with a specific changelist from Perforce, or revision number in Subversion, etc.).	revision:4536
stamp		Find all builds that started at the specified time (search by timestamp).	stamp:200811271753
status		Find all builds with the specified text in the build status text.	status:"Compilation failed"
tags	t	Find all builds with the specified tag.	tags:buildserver t:release
tests		Find all builds that include specified tests.	tests:
triggerer		Find all builds that were triggered by the specified user.	triggerer:ivan.ivanov
vcs		Find builds that have the specified VCS.	vcs:perforce

Shortcuts

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

i Note that when you use these shortcuts, do not insert the colon after it. That is, the query syntax is as follows: <shortcut><value to search>

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

Using Double-Colon

You can use the double-colon sign (::) to search for a project and/or build configuration by name:

- pro::best — search for builds of configurations with the names starting with "best", and in the projects with the names starting with "pro".
- mega:: — search for builds in all projects with names starting with "mega"
- ::super — search for builds of build configurations with names starting with "super"

"Magic" Keywords

TeamCity also provides "magic" keywords (see table below for the complete list). These magic keywords are formed with the '\$' sign and a word itself. The word can be shortened up to one (first) syllable, that is, the \$labeled, \$1, and \$lab keywords will be equal in a query. For example, to search for pinned builds of the "Nightly build" configuration in the "Mega" project you can use any of the following queries:

- configuration:nightly project:Mega \$pinned
- c:nigh p:mega \$pin
- M::night \$pin

Magic word	Description
\$tagged	Search for builds with tags. For example, Calcutta::Master \$t query will result in a list of all builds marked with any tag of build configurations whose name starts with "Master" from projects with names beginning with "Calcutta".
\$pinned	Search for pinned builds.

\$labeled	Search for builds that have been labeled in VCS. For example, to find labeled builds of the Main project you can use following queries: <code>p:Main \$labeled</code> , or <code>project:Main \$l</code> , or <code>m:: \$lab</code> , etc.
\$commented	Search for builds that have been commented.
\$personal	Search for personal builds. For example, using <code>-\$p</code> expression in your query will exclude all personal builds from search results.

Search by Build Log

By default, TeamCity does not search for builds by a certain text in build logs.

To enable search by the build logs, perform the following:

1. Set the `tc.search.indexBuildLog=true` TeamCity internal property
2. Reset the search cache on the **Administration| Server Administration| Diagnostics, Caches** tab or manually delete files from `<TeamCity Data Directory>\system\caches\search`. Please note that you might have to restart the server to let it pick up the updated search cache. Since TeamCity 9.0.1 restart is not required.

After re-indexing, TeamCity will be able to perform searching by specified text in the build logs and will list the relevant builds.

Maven-related Data

Maven Project Data

In TeamCity you can find information about settings specified in your Maven project's `pom.xml` file on the dedicated **Maven** tab of build configuration. In addition to getting a quick overview of the settings, you can find **Provided parameters** in the upper section of this page, e.g. `maven.project.name`, `maven.project.groupId`, `maven.project.version`, `maven.project.artifactId`. You can use these parameters within your build. You can reference them within the build number pattern using %-notation. For example: `%maven.project.version%.{0}`.

Maven Build Information

For each Maven build TeamCity agent gathers Maven specific build details, that are displayed on the **Maven Build Info** tab of the build results after the build is finished.

This page can be useful for build engineers when adjusting build configurations.

Administrator's Guide

In this section:

- TeamCity Configuration and Maintenance
- 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

TeamCity Configuration and Maintenance



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

To edit the server configuration:

On the [Administration](#) page, select **Global Settings**.

The **TeamCity Configuration** section of the page displays the following information:

Setting	Description
Database:	The database used by the running TeamCity server.
Data directory:	The <code><TeamCity data directory></code> path with the ability to browse the directory.

Artifact directories:	<p>The list of the root directories used by the TeamCity server to store build artifacts, build logs and other build data. The default location is <TeamCity data directory>/system/artifacts.</p> <p>Since TeamCity 9.1 the list can be changed by specifying a new-line delimited list of paths. Absolute and relative (to TeamCity Data Directory) paths are supported.</p> <p>All the specified directories use the same structure. When looking for build artifacts, the specified roots are searched for the directory corresponding to the build. The search is done in the order the root directories are specified. The first found build artifacts directory is used as the source of build artifacts. Artifacts for the newly starting builds are placed under the first directory in the list.</p>
Caches directory:	The directory containing TeamCity internal caches (of the VCS repository contents, search index, other), which can be manually deleted to clear caches .
Server URL	The configurable URL of the running TeamCity server.

The **Build Settings section** allows configuring the following settings:

Setting	Description
Maximum build artifact file size:	Maximum size in bytes. KB, MB, GB or TB suffixes are allowed. -1 indicates no limit
Default build execution timeout:	Maximum time for a build. Can be overridden when defining build failure conditions .

The **Version Control Settings** controls the following:

Setting	Description
Default VCS changes check interval:	Set to 60 seconds by default. Specifies how often TeamCity polls the VCS repository for VCS changes. Can be overridden when configuring VCS roots .
Default VCS trigger quiet period:	Set to 60 seconds by default. Specifies a period (in seconds) that TeamCity maintains between the moment the last VCS change is detected and a build is added into the queue. Can be overridden when configuring VCS triggers .

This section also includes:

- [Configuring Authentication Settings](#)
- [TeamCity Data Backup](#)
- [Projects Import](#)
- [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](#)
- [Build Time Report](#)
- [TeamCity Monitoring and Diagnostics](#)

Configuring Authentication Settings

TeamCity can authenticate users via an internal database, or can integrate into your system and use external authentication sources such as Windows Domain or LDAP.

- [Configuring Authentication](#)
 - [Simple Mode](#)
 - [Advanced Mode](#)

- User Authentication Settings
 - Special User Accounts
- Credentials Authentication Modules
 - Built-in Authentication
 - Windows Domain Authentication
 - LDAP Authentication
- HTTP Authentication Modules
 - Basic HTTP Authentication
 - NTLM HTTP Authentication

Configuring Authentication

Authentication is configured on the [Administration | Authentication](#) page; the currently used authentication modules are also displayed here.

TeamCity provides several preconfigured authentication options (presets) to cover the most common use-case described [below](#). The presets are combinations of authentication modules supported by TeamCity: three credentials authentication modules and two HTTP authentication modules:

- Credentials Authentication Modules
 - Built-in
 - Windows Domain Authentication
 - LDAP Integration (separate page)
- HTTP Authentication Modules
 - Basic HTTP Authentication
 - NTLM HTTP Authentication (separate page)

When you first log in to TeamCity, the default authentication including the Built-in and Basic HTTP Authentication modules is enabled and editing authentication settings in the [simple mode](#) is active.

To modify the existing settings, click the [Edit](#) link in the table next to the description of the enabled authentication module.

To switch to a different preconfigured scheme, use the [Load preset](#) button. For more options, switch to the [Advanced mode](#).

 Any changes made to authentication in the UI will be reflected in the `<TeamCity data directory>/config/auth-config.xml` file, which can be used to configure authentication if editing via the Web UI is not suitable for some reason. The detailed description is available in the [previous documentation version](#).

Simple Mode

Simple mode (default) allows you to select presets created for the most common use cases. To override the existing authentication settings, use the [Load preset...](#) button, select one of the options and [Save](#) your changes. The following presets are available:

- Default ([built-in authentication - Basic HTTP](#))
- [LDAP](#)
- Active directory ([LDAP with NTLM](#))
- Microsoft Windows Domain ([Basic HTTP and NTLM](#))

Advanced Mode

TeamCity allows enabling several authentication modules simultaneously using the advanced mode in the TeamCity Web UI.

When a user attempts to log in, all the modules will be tried one by one. If one of them authenticates the user, the login will be successful; if all of them fail, the user will not be able to log into TeamCity.

 It is possible to use a combination of internal and external authentication. The recommended approach is to configure [LDAP Integration](#) for your internal employees first and then to add [Built-in](#) authentication for external users.

1. Switch to advanced mode with the corresponding link on the [Administration | Authentication](#) page.
2. Click **Add Module** and select a module from the drop-down.
3. Use the properties available for modules by selecting/deselecting checkboxes in the **Add Module** dialog.
4. Click **Apply** and [Save](#) your changes.

Also, TeamCity plugins can provide [additional authentication modules](#).

User Authentication Settings

The very first time TeamCity server starts with no users (and no administrator) so you will be prompted for the administrator account. If you are not prompted for the administrator account, please refer to [How To Retrieve Administrator Password](#) for a resolution.

The TeamCity administrator can modify the authentication settings for every user on their profile page.

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.

Care should be taken when modifying authentication settings: there can be a case when the administrator cannot login after changing authentication modules.

Let's imagine that the administrator had the "jsmith" TeamCity username and used the default authentication. Then the authentication module was changed to Windows domain authentication (i.e. Windows domain authentication module was added and the default one was removed). If, for example, the Windows domain username of that administrator is "john.smith", he/she is not able to login anymore: he/she cannot login using the default authentication since it is disabled, and cannot login using Windows domain authentication since his/her Windows domain username is not equal to TeamCity username. The solution nevertheless is quite simple: the administrator can login using the super user account and change his/her TeamCity username or specify his/her Windows domain username on his/her own profile page.

Special User Accounts

By default, TeamCity has a [Super User](#) account with maximum permissions and a [Guest User](#) with minimal permissions. These accounts have no personal settings such as the [Changes](#) page and Profile information as they are not related to any particular person but rather intended for special use cases.

Credentials Authentication Modules

Built-in Authentication

By default, TeamCity uses the built-in authentication, meaning that users and their passwords are maintained by TeamCity.

When logging to TeamCity for the first time, the user will be prompted to create the TeamCity username and password which will be stored in TeamCity and used for authentication. If you installed TeamCity and logged into it, it means that built-in authentication is enabled and all user data is stored in TeamCity.

In the beginning the user database is empty and new users are either [added by the TeamCity administrator](#) or users are self-registered: the default settings allow the users to register from the login page. All newly created users belong to the [All Users](#) group and have all roles assigned to this group. If some specific [roles](#) are needed for the newly registered users, these roles should [be granted](#) via the [All Users](#) group.

By default, the users are allowed to change their password on their profile page.

Windows Domain Authentication

Allows users login using Windows domain name and password.

The credential check is performed on the TeamCity server side, so the server should be aware of the domain(s) users use to log in.

The supported syntax for the username is `DOMAIN\user.name` as well as `<username>@<domain>`.

In addition to login using login form, you can enable [NTLM HTTP Authentication](#) single sign-on.

If you select "Microsoft Windows Domain" preset, in addition to login via Windows domain, [Basic HTTP](#) and [NTLM](#) authentication modules are enabled by default.

Specifying Default Domain

To enable users to enter the system using the login form without specifying the domain as a part of the user name, do the following:

1. Go to the [Administration|Authentication](#) page.
2. Click the [edit](#) link in the table next to the **Microsoft Windows domain** authentication description.
3. Set the name in the **Default domain:** field.
4. Click **Done** and **Save** your changes.

Registering New Users on Login

The default settings allow users to register from the login page and TeamCity user names for the new users will be the same as their Windows domain account.

All newly created users belong to the [All Users](#) group and have all roles assigned to this group. If some specific [roles](#) are needed for the newly registered users, these roles should [be granted](#) via the [All Users](#) group.

To disable new user registration on login:

1. Go to the [Administration|Authentication](#) page.
2. Click the [edit](#) link in the table next to the **Microsoft Windows domain** authentication description. Uncheck the **Allow user registration from the login page** box.
3. Click the [edit](#) link in the table next to the **NTLM HTTP** authentication description. Uncheck the **Allow user registration from the login page** box.

Linux-Specific Configuration

If your TeamCity server runs under Linux, JCIFS library is used for the Windows domain login. The library is configured using the properties

specified in the `<TeamCity data directory>/config/ntlm-config.properties` file. Changes to the file take effect immediately without the server restart.

If the default settings do not work for your environment, refer to <http://jcifs.samba.org/src/docs/api/> for all available configuration properties.

If the library does not find the domain controller to authenticate against, consider adding the `jcifs.netbios.wins` property to the `ntlm-config.properties` file with the address of your WINS server. For other domain services locating properties, see <http://jcifs.samba.org/src/docs/resolver.html>.

LDAP Authentication

Please refer to the [corresponding section](#).

HTTP Authentication Modules

Basic HTTP Authentication

Please refer to [Accessing Server by HTTP](#) for details about basic HTTP authentication.

 For information on configuring Basic HTTP Authentication directly in the `<TeamCity data directory>/config/auth-config.xml`, refer to the previous documentation version.

NTLM HTTP Authentication

Please refer to the [corresponding section](#).

See also:

Concepts: [Authentication Modules](#)

LDAP Integration

LDAP integration in TeamCity has two levels: authentication (login) and users synchronization:

- **authentication** allows to login into TeamCity using LDAP server credentials.
- once LDAP authentication is configured, you can enable LDAP **synchronization** which allows the TeamCity users set to be automatically populated with the user data from LDAP.

LDAP integration is generic and can be configured for Active Directory or other LDAP servers.



It is recommended to configure LDAP authentication on a test server before enabling it in the production one, because switching to an incorrectly configured authentication scheme may cause users' inability to log in to TeamCity.

LDAP integration might be not trivial to configure, so it might require some trial and error approach to get the right settings. Please review [Typical LDAP Configurations](#). If a problem occurs, [LDAP logs](#) should give you enough information to understand possible misconfigurations. If you are experiencing difficulties configuring LDAP integration after going through this document and investigating the logs, please [contact us](#) and let us know your LDAP settings with a detailed description of what you want to achieve and what you currently get.

On this page:

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

Authentication

To allow logging into TeamCity with LDAP credentials, you need to configure LDAP connection settings in the `ldap-config.properties` file and enable LDAP authentication in the server's [Authentication section](#).

If you need to configure authentication without access to web UI refer to the [corresponding section](#) in the previous documentation version.

When the Allow creating new users on the first login option is selected (this is the default) a new user account will be created on the first successful login. The TeamCity user names for the new users will be derived from their LDAP data based on the configured setting. All newly created users belong to the **All Users** group and have all roles assigned to this group. If some specific **roles** are needed for the newly registered users, these roles can be granted via the **All Users** group.

TeamCity stores user accounts and details in its own database. For information on automatic user creation and automatic population of user details from LDAP, refer to [Synchronization](#) section below.

`ldap-config.properties` Configuration

LDAP integration settings are configured in the `<TeamCity data directory>/config/ldap-config.properties` file on the server.

Create the file by copying `<TeamCity data directory>/config/ldap-config.properties.dist` file and renaming it to the `<TeamCity data directory>/config/ldap-config.properties`; follow the comments in the file to edit the default settings as required.

The file uses the standard Java properties file syntax, so all the values in the file must be properly [escaped](#). The file is re-read on any modification so you do not need to restart the server to apply changes in the file.

It is strongly recommended to back up the previous version of the file: if you misconfigure LDAP integration, you may no longer be able to log in into TeamCity. The users who are already logged in are not affected by the modified LDAP integration settings because users are authenticated only on login.

The mandatory property in the `ldap-config.properties` file is `java.naming.provider.url` that configures the server and root DN. The property stores the URL to the LDAP server node that is used in following LDAP queries. For example, `ldap://dc.example.com:389/CN=Users,DC=Example,DC=Com`. Please note that the value of the property should use URL-escaping if necessary. e.g. use `%20` if you need the space character.



For samples of `ldap-config.properties` file please refer to the [Typical LDAP Configurations page](#).

The supported configuration properties are documented in comments in the `ldap-config.properties.dist` file.

Configuring User Login

The general login sequence is as follows:

- based on the username entered in the login form by the user, an LDAP search is performed (defined by the `teamcity.users.login.filter` LDAP filter where the user-entered username is referenced via the `$login$` or `$capturedLogin$` substring within the users base LDAP node (defined by `teamcity.users.base`),
- if the search is successful, authentication (LDAP bind) is performed using the DN found during the search and the user-entered password,
- if the authentication is successful, TeamCity user is created if necessary and the user is logged in. The name of the TeamCity user is retrieved from an attribute of the found LDAP entry (the attribute name is defined via the `teamcity.users.username` property)

When users login via LDAP, TeamCity does not store the user passwords. On each user login, authentication is performed by a direct login into LDAP with the credentials based on the values entered in the login form.

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

The related external link: [How To Set Up Secure LDAP Authentication with TeamCity](#) by Alexander Groß.

Active Directory

The following template enables authentication against active directory:

Add the following code to the `<TeamCity Data Directory>/config/ldap-config.properties` file (assuming the domain name is "Example.Com" and domain controller is "dc.example.com").

```
java.naming.provider.url=ldap://dc.example.com:389/DC=Example,DC=Com  
java.naming.security.principal=<username>  
java.naming.security.credentials=<password>  
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)  
teamcity.users.username=sAMAccountName  
java.naming.security.authentication=simple  
java.naming.referral=follow
```

Advanced Configuration

If you need to fine-tune LDAP connection settings, you can add the `java.naming` options to the `ldap-config.properties` file: they will be passed to the underlying Java library. The default options are retrieved using `java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory`. Refer to the Java [documentation page](#) for more information about property names and values.

You can use an LDAP explorer to browse LDAP directory and verify the settings (for example, <http://www.jxplorer.org/> or <http://www.ldapbrowser.com/softerra-ldap-browser.htm>).

There is an ability to specify failover servers using the following pattern:

```
java.naming.provider.url=ldap://ldap.mycompany.com:389 ldap://ldap2.mycompany.com:389  
ldap://ldap3.mycompany.com:389
```

The servers are contacted until any of them responds. There is no particular order in which the address list is processed.

Synchronization

Synchronization with LDAP in TeamCity allows you to:

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

TeamCity supports one-way synchronization with LDAP: the data is retrieved from LDAP and stored in the TeamCity database. Periodically, TeamCity fetches data from LDAP and updates users in TeamCity.

When synchronization is enabled, you can review the related data and run on-demand synchronization in the [Administration|LDAP Synchronization](#) section of the [server settings](#).

Common Configuration

You need to have LDAP authentication configured for the synchronization to function.

By default, the synchronization is turned off. To turn it on, add the following option to `ldap-config.properties` file:

```
teamcity.options.users.synchronize=true
```

You also need to specify the following mandatory properties:

- `java.naming.security.principal` and `java.naming.security.credentials` - they specify the user credentials which are used by TeamCity to connect to LDAP and retrieve data,
- `teamcity.users.base` and `teamcity.users.filter` - these specify the settings to search for users
- `teamcity.users.username` - the name of the LDAP attribute containing the TeamCity user's username. Based on this setting LDAP entries are mapped to the TeamCity users.

No users are created or deleted on enabling user's synchronization. Check the section [Creating and Deleting Users](#) for the related configuration.

User Profile Data

When synchronization is properly configured, TeamCity can retrieve user-related information from LDAP (e-mail, full name, or any custom property) and store it as TeamCity user's details. If updated in LDAP, the data will be updated in the user's profile in TeamCity. If modified in user's profile in TeamCity, the data will no longer be updated from LDAP for the modified fields. All the user fields synchronization properties store the name of LDAP field to retrieve the information from.

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

The list of supported user settings:

- teamcity.users.username
- teamcity.users.property.displayName
- teamcity.users.property.email
- teamcity.users.property.plugin: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.

Since TeamCity 8.0, there is an **experimental** feature which allows mapping user profile properties in TeamCity to a formatted combination of LDAP properties rather than to a specific property on user synchronization.

To enable the mapping, add `teamcity.users.properties.resolve=true` into `ldap-config.properties`.

Then you can use the %-references to LDAP attributes in the form of `%ldap.userEntry.<attribute>%` in the user property definitions, e.g.

```
teamcity.users.property.plugin\:notifier\:jabber\:jabber-account=%ldap.userEntry.name%@jabber.my.domain.com
```

User Group Membership

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

To configure Group membership:

1. Create groups in TeamCity manually.
2. Specify the mapping of LDAP groups to TeamCity groups in the `<TeamCity data directory>/config/ldap-mapping.xml` file. Use the `ldap-mapping.xml.dist` file as an example: TeamCity user groups are determined by the **Group Key**, LDAP groups are specified by the group DN.
3. Set the required properties in the `ldap-config.properties` file, the **groups settings** section:
 - `teamcity.options.groups.synchronize` enables user group synchronization
 - `teamcity.groups.base` and `teamcity.groups.filter` specify where and how to find the groups in LDAP
 - `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 the groups that are configured in the mapping. TeamCity synchronizes membership only for users residing directly in the groups. To map nested LDAP groups to TeamCity, copy your LDAP groups structure in TeamCity groups, together with the group inclusions. Then configure the mapping between the TeamCity groups and corresponding LDAP groups. See the [related issue](#).

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

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

Creating and Deleting Users

TeamCity can automatically create users in TeamCity, if they are found in one of the mapped LDAP groups and groups synchronization is turned on via `teamcity.options.groups.synchronize` option.

By default, automatic user creation is turned off. To turn it on, set `teamcity.options.createUsers` property to `true` in `ldap-config.properties` file.

TeamCity can automatically delete users in TeamCity if they cannot be found in LDAP or do not belong to an LDAP group that is mapped to predefined "All Users" group. By default, automatic user deletion is turned off as well; set `teamcity.options.deleteUsers` property to turn it on.

Username migration

The username for the existing users can be updated upon first successful login. For instance, suppose the user previously logged in using 'DOMAIN\User' name, thus the string 'DOMAIN\User' was stored in TeamCity as the username. To synchronize the data with LDAP, the user can change the username to 'user' using the following options:

```
teamcity.users.login.capture=DOMAIN\\(\.*)
teamcity.users.login.filter=(cn=$login$)
teamcity.users.previousUsername=DOMAIN\\$login$
```

The first property allows you to capture the username from the input login and use it to authenticate the user (can be particularly useful when the domain 'DOMAIN' isn't stored anywhere in LDAP). The second property `teamcity.users.login.filter` allows you to fetch the username from LDAP by specifying the search filter to find this user (other mandatory properties to use this feature: `teamcity.users.base` and `teamcity.users.username`). The third property allows you to find the 'DOMAIN\user' username when login with just 'user', and replace it with either the captured login, or with the username from LDAP.

Note that if any of these properties are not set or cannot be applied, the username isn't changed (the input login name is used). More configuration examples are available [here](#).

Scrambling credentials in `ldap-config.properties` file

Since **TeamCity 9.0** `java.naming.security.credentials` property can be used to configure either plain-text or scrambled-form passwords. To get the scrambled password value, execute the following command (must be executed from **<TeamCity installation directory>/webapps/ROOT/WEB-INF/lib** directory):

For Windows:

```
java -cp common-api.jar:commons-codec-1.3.jar:log4j-1.2.12.jar  
jetbrains.buildServer.serverSide.crypt.ScrambleMain <text to scramble>
```

For Linux:

```
java -cp common-api.jar:commons-codec-1.3.jar:log4j-1.2.12.jar  
jetbrains.buildServer.serverSide.crypt.ScrambleMain <text to scramble>
```

Note that scrambling is not encryption: it protects the password from being easily remembered when seen occasionally, but it does not protect against getting the real password value when someone gets the scrambled password value.

Debugging LDAP Integration

Internal LDAP logs are stored in `logs/teamcity-ldap.log` 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, use the "debug-ldap" [logging preset](#).

Typical LDAP Configurations

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

- Basic LDAP Login
 - Windows Active Directory
 - Unix
 - Specifying Backup LDAP server
- Basic LDAP Login for Users in Specific LDAP Group Only
- Active Directory With User Details Synchronization
- Active Directory With User Details Synchronization and User Creation
- Active Directory With Group Synchronization
 - Limiting the number of groups to be synchronized

Basic LDAP Login

The examples of minimal working configurations are given below.

Windows Active Directory

```
java.naming.provider.url=ldap://dc.example.com:389/DC=example,DC=com
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName
```

Note that "sAMAccountName" is limited to 20 symbols. You might want to use another attribute which contains entire username.
Unix

```
java.naming.provider.url=ldap://dc.example.com:389/DC=example,DC=com
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
teamcity.users.login.filter=(uid=$capturedLogin$)
teamcity.users.username=uid
```

TeamCity does not store the user passwords in this case.

On each user login, authentication is performed by a direct login into LDAP with the credentials entered in the login form.
Specifying Backup LDAP server

You can specify a backup LDAP server in the `java.naming.provider.url` property as follows:

```
# The second URL is used when the first server is down.
java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
ldap://failover.example.com:389/DC=example,DC=com
```

Basic LDAP Login for Users in Specific LDAP Group Only

Only users from a specific user group are allowed to log in. The users need to enter the username only the without domain part to log in. The example is for Windows Active Directory:

```
java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
java.naming.security.principal=<username>
java.naming.security.credentials=<password>

# filtering only users with specified name and belonging to LDAP group "Group1" with
# DN "CN=Group1,CN=Users,DC=example,DC=com"
teamcity.users.login.filter=(&(sAMAccountName=$capturedLogin$)(memberOf=CN=Group1,CN=Users,DC=example,DC=com))

#teamcity.users.username=sAMAccountName

# Allow only username part without domain (optional)
teamcity.auth.loginFilter=[^/\\\\\\@]+

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

Active Directory With User Details Synchronization

Users can log in to TeamCity with their domain name without the domain part, there is an account "teamcity" with password "secret" that can read all Active Directory entries. The TeamCity user display name and email are synchronized from Active Directory.

```
java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
java.naming.security.principal=CN=teamcity,CN=Users,DC=example,DC=com
java.naming.security.credentials=secret
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName

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

Active Directory With User Details Synchronization and User Creation

Users can log in to TeamCity with their domain name without the domain part, there is an account "teamcity" with password "secret" that can read all Active Directory entries. The TeamCity user display name and email are synchronized from Active Directory.
The users not existing in the TeamCity database are created.

Users no longer existing in Active Directory are deleted from the TeamCity user database.

```
java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
java.naming.security.principal=CN=teamcity,CN=Users,DC=example,DC=com
java.naming.security.credentials=secret
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName

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

# Automatic user creation and deletion during user synchronization

teamcity.options.users.synchronize.createUsers=true
teamcity.options.createUsers=true

teamcity.options.users.synchronize.deleteUsers=true
teamcity.options.deleteUsers=true
```

Active Directory With Group Synchronization

There should be `ldap-mapping.xml` file with one or more group mappings defined.

`ldap-config.properties` file:

```

java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
java.naming.security.principal=CN=teamcity,CN=Users,DC=example,DC=com
java.naming.security.credentials=secret
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName

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

# Automatic user creation and deletion of obsolete users during users synchronization

teamcity.options.users.synchronize.createUsers=true
teamcity.options.createUsers=true
teamcity.options.users.synchronize.deleteUsers=true
teamcity.options.deleteUsers=true

# Groups synchronization is on
teamcity.options.groups.synchronize=true

# The group search LDAP filter used to retrieve groups to synchronize.
# The result includes all the groups configured in the ldap-mapping.xml file.

teamcity.groups.filter=(objectClass=group)

# The LDAP attribute of a group storing its members.
teamcity.groups.property.member=member

```

Limits the number of groups to be synchronized

The `teamcity.users.filter` property helps limit the number of processed user accounts during users synchronization.

It is recommended to create the "TeamCity Users" group in Active Directory, and include all your required groups into this group, e.g. you may have the following Active Directory structure:

- Group A with members User 1, User 2
- Group B with members User 3, User 4
- Group "TeamCity Users" with members Group A, Group B

Then update the `teamcity.users.filter` property, e.g.

```
teamcity.users.filter=(&(objectClass=user)(memberOf:1.2.840.113556.1.4.1941:=CN=TeamCity Users,OU=Accounts,DC=domain,DC=com))
```

In this case TeamCity creates accounts only if they are members of the corresponding Active Directory group. Nested groups are supported.

Alternatively, you can list several groups:

```
teamcity.users.filter=(&(objectClass=user)(|(memberOf=CN=GroupOne,OU=myou,DC=company,DC=tld)(memberOf=CN=GroupTwo,OU=myou,DC=company,DC=tld)))
```

To limit users who can log in to TeamCity you also need to change `teamcity.users.login.filter` property:

```
teamcity.users.login.filter=(&(sAMAccountName=$capturedLogin$)(memberOf:1.2.840.113556  
.1.4.1941:=CN=TeamCity Users,OU=Accounts,DC=domain,DC=com))
```

For more details on the filter syntax refer to the [Microsoft documentation](#).

For more details on the AD attributes refer to the [Microsoft documentation](#).

LDAP Troubleshooting

General advice: if you experience problems with LDAP configuration, turn on the debug logging (see [Reporting Issues](#)).

Cannot authenticate using LDAP

Check the `teamcity-ldap.log` file. For each unsuccessful login attempt there should be a reason specified. Most commonly these are:

- The login filter doesn't match the entered login ("User-entered login does not match `teamcity.auth.loginFilter=..., aborting`")
- The LDAP server rejected login with the "Invalid credentials" message ("Failed to login user '...' due to authentication error. Cause: Invalid credentials ([LDAP: error code 49 - 80090308: LdapErr: DSID-0C090334, comment: AcceptSecurityContext error, data 525, vece^@])")

The first reason means that the login can't be used for signing in because it doesn't match a certain filter. For example, by default you can't login with 'DOMAIN\username' - the filter forbids '/', '\' and '@' symbols. See the `teamcity.auth.loginFilter` property.

The second error can be caused by various things, e.g.:

- You are trying to login with your username, but LDAP server accepts only full DNs
If all users are stored in one LDAP branch, you should use the `teamcity.auth.formatDN` property. Otherwise see the section below.
- Check your DN and the actual principal from the logs, probably there is a typo or an unescaped sequence. Try to log in with this principal using another LDAP tool.
- Try changing the security level (`java.naming.security.authentication`): it can be "simple", "strong" or "none".

Users in LDAP are stored in different branches, so the `teamcity.auth.formatDN` property can't be applied. How can the users login with their usernames?

This feature is available from version 5.0. You should specify how you want to find the user (`teamcity.users.login.filter`), e.g. by the username or e-mail. On each login TeamCity finds the user in LDAP *before* logging in, fetches the user DN and then performs the bind. Thus you should also define the credentials for TeamCity to perform search operations (`java.naming.security.principal` and `java.naming.security.credentials`).

NTLM HTTP Authentication

The TeamCity NTLM HTTP authentication feature employs Integrated Windows Authentication and allows transparent/SSO login to the TeamCity web UI when using browsers/clients supporting TLMv1, NTLMv2, Kerberos and Negotiate HTTP authentications.

Generally, it allows users to log in into the TeamCity server wen UI using their NT domain account without the need to enter credentials manually.

- Configuration

- Requirements
- Enabling NTLM HTTP Authentication
 - NTLM login URLs
- Using NTLM HTTP Authentication Module with LDAP Authentication
- Configuring client
 - Internet Explorer
 - Google Chrome
 - Mozilla Firefox
- Troubleshooting

Configuration

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

 For information on configuring authentication the settings directly in the `<TeamCity data directory>/config/auth-config.xml` file, refer to the previous documentation version.

You can enable NTLM login with any login module once the TeamCity username is the same as the Windows domain username or the Windows domain username is specified on the user profile.

 NTLM HTTP authentication is supported only for TeamCity servers installed on Windows machines. If you need it under other platforms, feel free to [contact us](#) with details as to why.

Requirements

1. The authenticating user should be logged in to the 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 the NTLM HTTP authentication module is configured, users will see a link on the login screen which, when clicked, will force the browser to send the domain authentication data.

You can force the server to announce NTLM HTTP authentication by specifying protocols in the "Force protocols" setting.

This will make the server request domain authentication for any request to the TeamCity web UI. If the user's browser is run in the domain environment, the current user will be logged in automatically. If not, the browser will pop up a dialog asking for domain credentials.

Without this attribute, NTLM HTTP authentication will work only if the client explicitly initiates it (e.g. clicks the "Login using NT domain account" link on the login page), and in the usual case an unauthenticated user will be simply redirected to the TeamCity login page. The TeamCity server forces NTLM HTTP authentication only for Windows users by default. If you want to enable it for all users, set the following [internal property](#):

```
teamcity.ntlm.ignore.user.agent=true
```

NTLM login URLs

There are two more ways to force NTLM authentication for a certain connection (there is no need to set the `forceProtocols` attribute for this case):

- Send request to `<Your TeamCity server URL>/ntlmLogin.html` and TeamCity will initiate NTLM authentication and redirect you to the overview page.
- Send request to `<Your TeamCity server URL>/ntlmAuth/<path>` and TeamCity will initiate NTLM authentication and show you the `<path>` page (without redirect).

Using NTLM HTTP Authentication Module with LDAP Authentication

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

With this property set, a user will be able to log in via their NT domain account only if he/she already has an existing account in TeamCity (i.e. if he/she has already logged into TeamCity earlier via LDAP) with a TeamCity username which equals the Windows domain username or a custom NT domain username specified on the user's profile page.

Configuring client

Depending on your environment, you may need to configure your client to make NTLM authentication work.

Internet Explorer

1. Open **Tools | Internet Options**.
2. On the **Advanced** tab make sure the option **Security | Enable Integrated Windows Authentication** is checked.
3. On the **Security** tab select **Local Intranet | Sites | Advanced** and add your TeamCity server URL to the list.

Google Chrome

On Windows, Chrome normally uses IE's behaviour, see more information [here](#).

Mozilla Firefox

1. Type `about:config` in the browser's address bar.
2. Add your TeamCity server URL to the `network.automatic-ntlm-auth.trusted-uris` property.

Troubleshooting

Helpful links:

- <http://waffle.codeplex.com/wikipage?title=Frequently%20Asked%20Questions>
- <http://waffle.codeplex.com/discussions/254748>
- <http://waffle.codeplex.com/wikipage?title=Troubleshooting%20Negotiate&referringTitle=Documentation>

Enabling Guest Login

Logging in as a guest user is turned off by default.

To enable the guest login to TeamCity:

1. Navigate to the **Administration | Authentication** page.
2. Select the **Allow login as guest user** option.
3. Save your changes.

The **Log in as guest** link appears on the **Log in to TeamCity** page.

By default, the guest user can view all the projects. To customize which projects guest user has access to:

Click the **Configure guest user roles** link to configure the roles. The link appears when the **per-project authorization mode** is enabled.

See also:

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

Administrator's Guide: [Configuring Authentication Settings | Managing Users and User Groups](#)

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. To include all data, use the tool when the server is stopped. Restore is possible via the `maintainDB` console tool.
- **Manual backup:** is suitable if you want to manage the backup procedure manually.

You may need to back up the build agent's data only.

Restoring data from backup is performed using the `maintainDB` tool.



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

The recommended approach is either to perform the backup process described under [Manual Backup and Restore](#) or run a backup from the [Web UI](#) regularly (e.g. automated via REST API) with the "Basic" level - this will ensure backing up all important data except build artifacts and build logs.

Build artifacts and logs (if necessary) can be backed up manually by copying files under `.BuildServer/system/artifacts` and, prior to TeamCity 9.0, `.BuildServer/system/messages`. See [TeamCity Data Directory#artifacts](#) for details. Since TeamCity 9.1, if logs are selected for backup, TeamCity will be searching for them in all artifact directories currently specified on the server.

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 [Administration | Backup](#) page, specify backup parameters as described below, and start the backup process.

Option	Description
Backup file	<p>Specify the name for the backup file, the extension (.zip) will be added automatically. By default, TeamCity will store the backup file in the <code><TeamCity Data Directory>/backup</code> 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 <code><TeamCity Data Directory>/config/backup-config.xml</code> file. For example:</p> <div style="border: 1px solid #ccc; padding: 10px;"><pre><backup-settings> ... <general> <backup-dir path="C:/TC-Backups" /> </general> ... </backup-settings></pre></div>
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="border: 1px solid #2e8b57; padding: 5px; background-color: #e0f2e0;"><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 the 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 a backup created during server running. To include the builds, consider using another [creating backup](#) approach when 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 a specific structure that does not depend on the OS or database type you

use. Thus, you can use the backup file to restore your data even on a different Operating System, or with another database. If you change the contents of this file manually, TeamCity won't be able to restore your data.

On the **History** tab of the **Administration | 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. Backups created with the `maintainDB` utility are not displayed on the **History** tab.

See also:

- [Installation and Upgrade: Upgrade](#)
- [Concepts: TeamCity Data Directory](#)
- [Administrator's Guide: TeamCity Data Backup](#)

Creating Backup via `maintainDB` command-line tool

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

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

 This document describes some of the `maintainDB` options. For a complete list of all available 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 Home>`, including used server port number
- TeamCity application logs (they also reside under `<TeamCity 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 backing up data, it is recommended to shut down the TeamCity server to include all builds into the backup. If the backup process is started when the TeamCity server is up, running and queued builds are not included into the backup. If the TeamCity server is shut down, all builds are included into the backup.

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
```

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
- build artifacts by default, unless a different location is configured
- 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>/logs` directory.



You may also want to back up TeamCity Windows Service settings, if they were modified.

Manual Restoration of Server Backup

If you need to restore backup created with the web UI or `maintainDB` utility, please refer to [Restoring TeamCity Data from Backup](#). This section describes restoration of a manually created backup.

You should always restore both the data in the `<TeamCity Data Directory>` and data in the database. Both the database and the directory should be backed up/restored in sync.

TeamCity Data Directory Restoration

You can simply put the previously backed up files back to their original places. However, it is important that no extra files are present when restoring the backup.

The simplest way to achieve this is to restore the backup over a clean installation of TeamCity. If this is not possible, make sure the files created after the backup was done are cleared. Especially the newly created files under the artifacts, messages, changes directories under `<TeamCity Data Directory>/system`.

TeamCity Database Restoration

Database should be restored using database-specific tools and methods. You might also want to use database-specific methods to make the restore faster (like setting SQL Server "Recovery Model" to "Simple").

Prior to restoring make sure there are no extra tables in the schema used by TeamCity.

Restoration to New Server

If you want to run a copy of the server, make sure the servers use distinct data directories and databases. For an external database, make sure you modify settings in the `<TeamCity Data Directory>/config/database.properties` file to point to another database.

Restoring Build Logs

Since TeamCity 9.1 build logs (located in the `/logs` subdirectory of `/artifacts`) can be backed up from multiple artifact directories. In this case during the restore TeamCity will detect logs from several locations and will ask you to select a single directory where all build logs will be restored to.

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

This document describes some of the `maintainDB` options. For a complete list of all available options, run `maintainDB` from the command line with no parameters.

On this page:

- Performing full restore
- Restoring database only
- Resuming restore after interruption

You can restore backed up data into the same or a different database; from/to any of the supported databases, e.g. you can restore data from a HSQL database to a PostgreSQL database, as well as restore a backup of PostgreSQL database to a new PostgreSQL database.

During database restoration you might want to configure database-specific settings to make the bulk data changes faster (like setting SQL Server "Recovery Model" to "Simple").

Performing full restore

To perform full restore a TeamCity server from a backup file:

1. Install the TeamCity server from a `.tar.gz` or `.exe` installation package. Do not start the TeamCity server.
2. Create a new empty TeamCity Data Directory.
3. Select one of the options:
 - a. To restore the backup into a new external database, create and configure the database, placing the `database.properties` file into any directory other than the TeamCity Data Directory.
 - b. To restore the backup into the internal database, save the code below to the `database.properties` file and place the file into any directory other than TeamCity Data Directory:

```
# Database: HSQLDB (HyperSonic) version 2.x
connectionUrl=jdbc:hsqldb:file:$TEAMCITY_SYSTEM_PATH/buildserver
```

4. Place the required [database drivers](#) into the lib/jdbc sub directory.
5. Use the `maintainDB` utility located in the <TeamCity Home>/bin directory.
6. Use the `restore` command:

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

- The `-A` argument can be omitted if you have the `TEAMCITY_DATA_PATH` environment variable set.
- The `-F` argument can be an absolute path or a path relative to the <TeamCity Data Directory>/backup directory.

By default, if no other option except `-F` is specified, all of the backed up scopes will be restored from the backup file. To restore only specific scopes from the backup file, use the corresponding options of the `maintainDB` utility: `-D`, `-C`, `-U`, `-L`, and `-P`.



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

You can also copy the files that were not included into the backup into the data directory (most importantly, build artifacts, located in <TeamCity Data Directory>/system/artifacts by default), see details on the directories in the [TeamCity Data Directory](#) description.

Restoring database only

Before restoring a TeamCity database to an existing server, make sure the Teamcity server is not running.

To restore a TeamCity database only from a backup file to an existing server:

1. [Create and configure the database](#), placing the `database.properties` file into any directory other than the TeamCity Data Directory.
2. Ensure that the required database drivers are present in the TeamCity Data Directory/lib/jdbc sub directory.
3. Use the `maintainDB` utility located in the <TeamCity Home>/bin directory (only available in TeamCity .tar.gz and .exe distributions).
4. Use the `restore` command:

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

5. See the `maintainDB` utility console output. You may have to copy the `database.properties` file manually if requested.

Resuming restore after interruption

The restore may be interrupted due to the following reasons:

- Lack of space on the file system or in the database
- Insufficient permissions to the file system or the database.

The interruption occurs when one of tables or indexes failed to be restored, which is indicated in the `maintainDB` utility console output. **Before resuming the restore**, manually delete the incorrectly restored object from the database.

To resume the backup restore after an interruption:

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

```
maintainDB.[cmd|sh] restore <all previously used restore options> --continue
```

See also:

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

Projects Import

Since TeamCity 9.0, you can import projects with all their data and user accounts from a backup file to an existing TeamCity server.

To start importing projects:

- create a usual backup file on the source TeamCity server containing the projects to be imported (note that the **versions of the source and target TeamCity servers have to be the same**)
- put the backup file into the `<TeamCity Data Directory>/import` directory on the target server
- follow the import steps on the [Administration| Projects Import](#) page.

On this page:

- [Importing projects](#)
 - Defining import scope
 - Configuration files import
 - Importing users and groups
 - Conflicts
- [Data excluded from import](#)
 - Viewing Import Results
 - Moving artifacts and logs

Importing projects

After selecting a backup file, you need to specify which projects will be imported.

TeamCity will analyze the selected projects to see if they will be imported, merged or skipped.

- The project will be **imported** if it is new for the target server. All its entities (Build Configurations, Templates, Builds, etc) and their data will be created on the target server.
- The project will be **merged** if the same project already exists on the target server (if the source and target project have the same [UUID](#) and [external ID](#)).
During merging the existing entities will remain intact and only the entities new for the target will be imported with the related data.
The data for the existing entities will not be imported or merged: new data will not be added to the existing entities (e.g. changes will not be added to an existing VCS root), the existing files will not be changed (e.g. if the same template exists on both servers with different settings, the target file will be preserved).
It means, for example, that you cannot import missing builds to the existing build configuration. If you need to add the missing data to the existing entities, for example, import new builds into an already imported build configuration, then you should remove this build configuration using the UI and re-import its project.
- The project will be **skipped** if a [conflict](#) occurs: either the project's [UUID](#) is new but its external ID already exists on the target; or if the source and target projects have the same [UUID](#) but different external IDs.

Defining import scope

You can select the import scope: choose among project settings, builds and changes history, and user accounts or import all of them. Since an imported project can also use settings from its parent, TeamCity will also import all the vcs roots, templates, meta-runners and other project-related settings for parent projects. If the same project already exists on the target server, the existing objects will not be overwritten.

Configuration files import

For each imported or merged project, the configuration files are imported to the [data directory](#) on the target server, provided they are new for the target. The existing files will not be changed.

The following files are imported:

- Configuration xml files for the Project with its Build Configurations, Templates, and VCS Roots as well as its subprojects.
- All files from the `<TeamCity Data Directory>/plugins` directory.
- Build Numbers files for the newly added build configurations.

Importing users and groups

When users are selected for import, TeamCity will analyze the usernames to see if users will be **imported** or **merged**.

TeamCity users must have unique usernames.

- A user account whose username is new for the target server will be **imported**. Such users appear on the target server in a separate group marked *Imported <Import Date Time>*. All the related data (personal builds, changes, test mutes and investigations) will be created on the target server. The [user account settings](#) (roles, permissions, VCS names, notification settings, etc.: system-wide settings as well as the settings related to the imported projects) are preserved during import.
- User accounts with the same username on the source and the target servers will be **automatically merged**. During merging the existing data will remain intact and only the data new for the target will be added: all the new user-related data (personal builds, changes, test mutes and investigations) and the [user account settings](#) (roles, permissions, VCS names - **since TeamCity 9.0.1**, notification settings, etc.: system-wide settings as well as the settings related to the imported projects) will be added to the user on the target server.

 Automatic merging may cause a problem if the same username belongs to different users on the source and the target server: during

import the user information will be merged anyway.

-  Note that the scope of user permissions on the target may change after import, e.g.
- if a user has the system administrator role on the source, this role will be added to the user on the target after import,
 - if a user has several roles in several projects on the source, only the new roles for the projects within the import scope will be added on the target.

Import of user groups works the same way: new groups are imported, while the existing groups are automatically merged.

If a **conflict** occurs (the group exists on both the source and the target, but the group roles are different), after import the group on the target server may get additional roles. As a result, a member of this group on the target will get additional roles and permissions as well.



Since TeamCity 9.0.3, existing users are not automatically merged. All conflicting users are divided into two groups - those with the same email on both servers and those with different ones.

You can view basic information about the users from both groups and decide whether you want to merge them or not.

The same is true for user groups - you can view all the groups that have the same group key and decide if you want to merge them. Note that "**All Users**" group is always listed as a conflicting one because it is a default group on all TeamCity servers.

Conflicts

TeamCity does not import entities from the backup file if they conflict with some entity on the target server. **Since TeamCity 9.0.3**, TeamCity analyzes the backup file before import and displays all detected conflicts on the **Import Scope** configuration page.

It is **highly recommended to resolve all conflicts** before proceeding with the import, as unresolved conflicts may result in unpredictable behavior after the import, e.g.:

- Critical errors can be shown if, for example, some VCS Root was skipped, but a Build Configuration depending on it was imported.
- Imported Build Configurations may refer to the wrong Template if there was an unresolved conflict of **external IDs** between the templates from the source and target servers.

Data excluded from import

There is a number of limitations regarding the import of project-related data:

- audit records are imported only if users are selected in the scope;
- the backup files do not contain artifacts and logs (since version 9.0 build logs are stored under build artifacts), so these are not imported automatically, but TeamCity provides scripts to move them **manually**;
- running builds and the build queue are not included in the backup and not imported;
- global server settings (authentication schemes, custom roles, etc.) are not imported



Importing projects may take significant time.

There can be only one import process per server.

Viewing Import Results

Each import process creates the `projectsImport-<date>` directory under the TeamCity logs allowing you to view the import results.

The directory contains the following:

- conflicting files folder, containing all data which has been merged
- mappings, containing mapping of the fields in the source and target databases
- scripts for copying artifacts and logs (see the section [below](#))
- import report, listing import results including the information on the data which has not been imported (if any)

Moving artifacts and logs

Although artifacts and logs are not imported right from the backup file, you can copy/move them from the source to the target server using the `.bat` and `.sh` scripts from the `projectsImport-<date>` directory under TeamCity logs. These scripts accept the source and target data directories via the command line; **since TeamCity 9.1 EAP2** the scripts accept the source and target **artifact** directories. The rest is done automatically. The scripts can be executed while the server is running.

-  It may take some time for TeamCity to display the imported build artifacts.

TeamCity Startup Properties

Please see corresponding section:

[Configuring TeamCity Server Startup Properties](#)

[Configuring Build Agent Startup Properties](#)

Configuring Server URL

The server URL configured in the Administration UI (on **Administration | Global Settings** page) is used by the server to generate links to the server when the URL cannot be derived from any other parameter. These cases include Notifications (email, Jabber, etc.) and some other actions performed not within a web request. All generated links will be prefixed by this URL.

Please make sure the server is accessible by the URL specified.

In most cases TeamCity correctly autodetects its own URL and sets it as the **Server URL**. However, sometimes autodetection is not possible/correct (for example, when the TeamCity server is running behind the Apache proxy). For such cases you can specify the server URL on the **Administration | Global Settings** page, or in the <TeamCity data directory>/config/main-config.xml file using the following format (no server restart is required after the change):

```
<server rootURL="http://some.host.com:port">  
</server>
```

Configuring TeamCity Server Startup Properties

Various aspects of TeamCity behavior can be customized through a set options passed on a TeamCity server start. These options fall into two categories: affecting Java Virtual Machine (JVM) and affecting TeamCity behavior.



You do not need to specify any of the options unless you are advised to do by the TeamCity support team or you know what you are doing.

In this section:

- TeamCity internal properties
- JVM Options
 - Standard TeamCity Startup Scripts

TeamCity internal properties

TeamCity has some properties that are not exposed in the UI. If you need to set such a property (e.g. asked by TeamCity support), add it to 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**.

An alternative way to add an internal property is to pass it as a -D<name>=<value> JVM option (see below).

You can use custom internal properties settings to tweak various aspects of the TeamCity [Web UI](#), [notifications](#), etc. to your needs.

JVM Options

If you need to pass additional JVM options to a TeamCity server (e.g. -D options mentioned at [Reporting Issues](#) or any non--D options like -X...), the approach will depend on the way the server is run. If you are using the .war distribution, use the manual of your Web Application Server. In all other cases, please refer to the [#Standard TeamCity Startup Scripts](#) section below.

For general notes on the memory settings, please refer to [Setting Up Memory settings for TeamCity Server](#).

You will need to restart the server for the changes to take effect.

Standard TeamCity Startup Scripts

If you run the server using the runAll or teamcity-server scripts or as a Windows service, you need to set the options via the OS [environment variables](#) passed to the TeamCity server process:

- TEAMCITY_SERVER_MEM_OPTS — server JVM memory options (e.g. -Xmx750m -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 or as global environment variables. You might need to reboot the machine after the environment change for the changes to have effect.

See also:

Concepts: TeamCity Data Directory

Administrator's Guide: Configuring Build Agent Startup Properties

TeamCity Tweaks

This page lists some of the the [internal properties](#) which can be used to tweak certain aspects of TeamCity behavior.

It is not recommended to use any of these unless you face an issue which you expect to address by using the properties.

Please note that the support for any of these properties can be abandoned in the future versions of TeamCity without any notice. Thus, if you find a property useful in your environment please let us know about that: detail your case and the properties/values used in an email sent to our [support email address](#).

On this page:

- [Web Page Refresh Interval](#)

Web Page Refresh Interval

You can configure different polling intervals (in seconds) for different pages in the TeamCity Web UI:

Property	Default	Description
teamcity.ui.pollInterval	6	How often the server is queried for common events (like build statuses, agents counter and so on). Since TeamCity 9.0 this property works only if WebSocket connection is not available and polling is used instead.
teamcity.ui.events.pollInterval	6	the delay between an event (received via polling or WebSockets) and the ajax request to update the UI. <ul style="list-style-type: none">With WebSocket, a client receives the event immediately, but reacts to it after the specified interval; as a result, e.g. a started build appears on the Overview page with a delay.With polling, a client receives the event during the polling request determined by <code>teamcity.ui.events.pollInterval</code> and reacts to it after the delay defined by <code>teamcity.ui.events.pollInterval</code>: e.g. a started build appears on the Overview page after <code>teamcity.ui.events.pollInterval + teamcity.ui.events.pollInterval</code> seconds
teamcity.ui.systemProblems.pollInterval	20	
teamcity.ui.problemsSummary.pollInterval	8	
teamcity.ui.buildQueueEstimates.pollInterval	10	

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. The most common approach for this is to setup a middle proxying server like Nginx or 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 [Set Up TeamCity behind a Proxy Server](#) section.

For small servers you can also setup HTTPS by the internal [Tomcat means](#).

See also a feature request: [TW-12976](#).

Authenticating with server certificate (HTTPS with no client certificate)

If your certificate is valid (i.e. it was signed by a well known Certificate Authority like Verisign), then TeamCity clients should work with HTTPS without any additional configuration. All you have to do is to use `https://` links to the TeamCity server instead of `http://`.

If your certificate is not valid: (so it is self-signed, not signed by a CA)

- 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 the CA Root certificate of the server's certificate to a file in one of the [supported formats](#). This can be done in a browser by inspecting certificate data and exporting it as Base64 encoded X.509 certificate.
- locate the JRE used by the 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 commands 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-----  
MIIGUjCCBDqgAwIBAgIEAKmKxzANBgkqhkiG9w0BAQQFADBwMRUwEwYDVQQDEwxK  
...  
-----END CERTIFICATE-----
```

Let's assume its name is <path to root certificate>.

4. Now import the root certificate to the trusted keystore with the command:

```
keytool -importcert -trustcacerts -file <path to root certificate> -keystore <path to trust keystore file> -storepass <trust keystore password>
```

Here you can use new values for <trust keystore path> and <trust keystore password> (or use existing trust keystore).

Starting the connecting application JVM

Now you need to pass the following parameters to the JVM when running the application:

```
-Djavax.net.ssl.keyStore=<path to keystore file>  
-Djavax.net.ssl.keyStorePassword=<keystore password>  
-Djavax.net.ssl.trustStore=<path to trust keystore file>  
-Djavax.net.ssl.trustStorePassword=<trust keystore password>
```

For IntelliJ IDEA you can add the lines into 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.

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
 - Changing Logging Settings
- Reading Logs
 - General Logging Configuration

General Logging Description

TeamCity uses [log4j library](#) for the logging and its settings can be [customized](#).

By default, log files are located under the `<TeamCity Server home>/logs` directory.

The most important log files are:

<code>teamcity-server.log</code>	General server log
<code>teamcity-activities.log</code>	Log of user-initiated and main build-related events
<code>teamcity-vcs.log</code>	Log of VCS-related activity
<code>teamcity-cleanup.log</code>	contains clean-up-related log
<code>teamcity-notifications.log</code>	Notifications-related log
<code>teamcity-clouds.log</code>	(off by default) Cloud-integration-related log
<code>teamcity-sql.log</code>	(off by default) Log of SQL queries, see details
<code>teamcity-http-auth.log</code>	(off by default) Log with messages related to NTLM and other authentication for HTML requests
<code>teamcity-xmlrpc.log</code>	(off by default) Log of messages sent by the server to agents and IDE plugins via XML-RPC

vcs-content-cache.log	(off by default) Log related to individual file content requests from VCS
teamcity-rest.log	(off by default) REST-API related logging
teamcity-freemarker.log	(off by default) Notification templates processing-related logging
teamcity-agentPush.log	(off by default) Logging related to agent push operations
teamcity-remote-run.log	(off by default) Logging related to personal builds processing on the server
teamcity-svn.log	(off by default) SVN integration log
teamcity-tfs.log	(off by default) TFS integration log
teamcity-starteam.log	(off by default) StarTeam integration log
teamcity-clearcase.log	(off by default) ClearCase integration log
teamcity-ldap.log	LDAP-related log
teamcity-nuget.log	NuGet-related log
teamcity-maintenance.log	(off by default) logs of back-up/ restore/ migration performed with maintainDB tool
teamcity-maintenance-truncation.log	(off by default) contains extended information on possible data truncation during back-up/ restore/ migration performed with maintainDB tool
teamcity-versioned-settings.log	(off by default) contains information on synchronization of the project settings with the version control
teamcity-ws.log	logs related to communication between browsers and the TeamCity server using the WebSoc ket connection

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**.

The debug logging can be enabled via "logging preset" under **Administration | Diagnostics** page, **Troubleshooting, Debug logging** subsection . Choosing a preset changes logging configuration. Since TeamCity 9.1 the preset is preserved even after a server restart, until changed on the page again. Before 9.1 the preset was reset to default on the server restart (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.

Changing Logging Settings

If you want to fine-tune the log4j configuration, you can edit `<TeamCity Server home>/conf/teamcity-server-log4j.xml` file (for .war TeamCity distribution, see [the related section](#)). If the server is running, the log4j configuration file will be reloaded automatically and the logging configuration will be changed on the fly (some log4j restrictions still apply, so for a massive change consider restarting the server).

Most useful settings of log4j configuration:

To change the minimum log level to save in the file, tweak the "value" attribute of the "priority" element:

```
<category ...>
    <priority value="INFO" />
...

```

The logs are rotated by default. When debug is enabled, it makes sense to increase "value" attribute of "maxBackupIndex" element to affect the number of preserved log files. While doing so, please ensure there is sufficient free disk space available.

```
<appender ...>
    <param name="maxBackupIndex" value="10" />
...

```

For detailed description of maxBackupIndex and other supported attributes, see <http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/RollingFileAppender.html>.

Reading Logs

Each message has a timestamp and level (ERROR, WARN, INFO, DEBUG).

e.g.:

```
[2010-11-19 23:22:35,657] INFO - s.buildServer.SERVER - Agent vmWin2k3-3
has been registered with id 19, not running a build
```

ERROR means an operation failed and some data was lost or action not performed. Generally, there should be no ERRORS in the log.

WARNs generally means that an operation failed, but will be retried or the operation is considered not important. Some amount of WARNs is OK. But you can review the log for such warnings to better understand what is going OK and what is not.

INFO is an informational message that just reports on the current activities.

DEBUG is only useful for issue investigation. e.g. to be analyzed by TeamCity developers.

General Logging Configuration

By default TeamCity searches for log4j configuration in a `../conf/teamcity-server-log4j.xml` file (this resolves to `<TeamCity Server home|TeamCity Home Directory>/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.

See also:

[Administrator's Guide: Viewing Build Agent Logs](#)
[Troubleshooting: Reporting Issues](#)

Build Agents Configuration and Maintenance

The [Agents](#) page of the TeamCity Web UI provides the comprehensive information on the TeamCity agents.

Connected / Disconnected

The **Connected** and **Disconnected** tabs display the agents by Agent pool (default). Uncheck the **Group by agent pool** box to view the agents alphabetically.

For each pool TeamCity displays the status of its build agents. Clicking the arrow next to the pool displays the list of the pools agents with their statuses.

Enabling/Disabling Agents via UI

Teamcity distributes builds only among the enabled agents.

Agents can be manually enabled/disabled via the web UI by clicking the status icon (1) next to the agent's name.

Optionally, you can tell TeamCity to automatically disable/enable the agent after a period of time and enter your comment.

TeamCity will follow the instructions and show the comment icon (2). Hovering over the icons will display the related information (3).

Pools

Refer to [a separate page](#) for information on configuring agent pools in TeamCity.

Parameter report

Filter all available agents using a specified parameter.

Matrix and Statistics

Refer to a separate page for information on [viewing the agents workload](#).

Diff

Compare two agents and see their differences highlighted.

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` or `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` or `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` or `sh` script, set the `TEAMCITY_LAUNCHER_OPTS` environment variable.

Build Agent Is Run As Service

In the `<Agent Home>\launcher\conf\wrapper.conf` file, add the following lines (one per option, the `N` number should increase):

```
wrapper.java.additional.<N>
```



Make sure you re-number all the lines after the inserted ones.

See also:

Concepts: Agent Home Directory

Administrator's Guide: Configuring TeamCity Server Startup Properties

Viewing Agents Workload

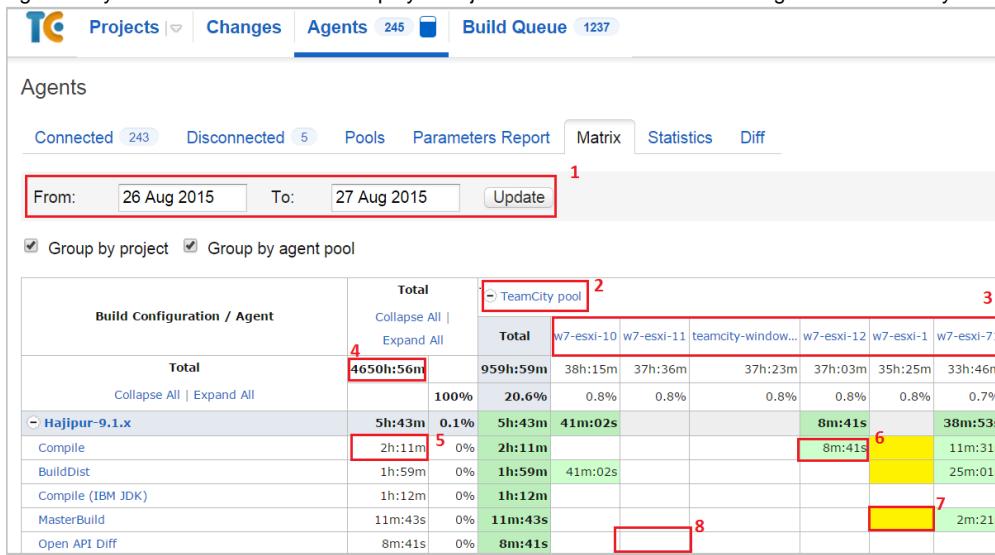
TeamCity provides handy ways to estimate build agents efficiency and help you manage your system:

- Load statistics matrix
- Build Agents' workload statistics

Load Statistics Matrix

The Matrix available at the **Matrix** tab on the **Agents** page provides you with a bird's-eye view of the overall Build Agents workload for all finished builds during the time range you selected.

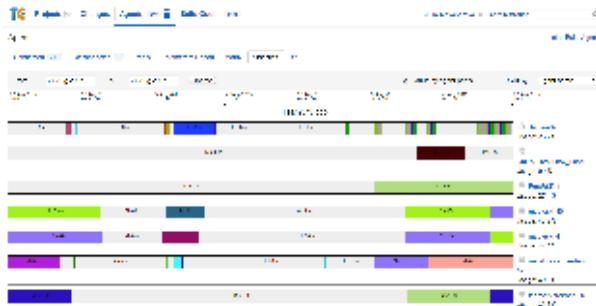
By taking a look at the build configurations compatible with a particular agent, you can [assign the build configuration to particular Build Agents](#) and significantly lower the idle time. This helps you adjust the hardware resources usage more effectively and fill the discovered productivity gaps.



1. Specified time range. Click **Update** to reload the matrix.
2. [Agent pool](#). Clicking a link opens the pool's page.
3. Individual agents. Clicking a link opens the [agent's details](#) page.
4. Total build duration during the specified time.
5. Total build duration for a particular build configuration during the specified time range.
6. Total duration for a particular build configuration of builds that were run on the particular build agent during the specified time period.
7. The build agent is compatible with the build configuration, but no builds were run during the specified time range.
8. The build agent is incompatible with the configuration, or disconnected.

Build Agents' Workload Statistics

TeamCity provides a number of visual metrics, namely, statistics of Build Agents' activity and their usage during a particular time period. These information is available at the **Statistics** tab on the **Agents** page.



You'll find this feature helpful in:

- your daily administration activities aimed at lowering your company's network workload
- locating and eliminating the gap between the most frequently used computers and those which are often idle
- reducing the cost of your hardware resources ownership

See also:

[Concepts: Build Agent](#)

[Installation and Upgrade: Installing and Running Additional Build Agents](#)

Viewing Build Agent Details

To view the state and information about an agent, click its name or navigate to the **Agents** page, find the agent in the list of connected, disconnected or authorized agents and click its name.

For each connected agent TeamCity provides the following information:

Agent Summary

- **Status:** [learn more about an agent's status](#).
 - **Details:** the agent's name, host name and IP address, port number, operating system, the agent's CPU rank, the pool the agent belongs to, and the current version number.
 - If there is a **build currently running** on the agent, the page displays information on the build with the link to the [build results](#).
 - **Miscellaneous:** this section provides the options to
 - [Clean sources on the agent](#)
 - Open Remote Desktop: available for agents running on the Windows operating system if the RDP client is installed and registered on the agent.
 - Reboot Agent Machine: available to users with *Reboot build agent machines* permissions. Click the link and confirm the reboot action. By default, the TeamCity agent will wait until the current build finishes. Deselect the checkbox and click Reboot to restart the agent immediately.
- Additional configuration of the reboot command is possible.
- ▼ [Click to view details](#).

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. Additional configuration might be required if the default reboot command fails.

To tweak agent reboot, add the `teamcity.agent.reboot.command` agent configuration parameter to the `buildagent.properties` file with the command to execute when reboot is required.

Example configuration:

```
teamcity.agent.reboot.command=sudo shutdown -r 60
or
teamcity.agent.reboot.command=shutdown -r -t 60 -c "TeamCity Agent reboot
command" -f
```

- Dump threads on agent.

Build History

Shows the builds that were run on the agent

Compatible Configurations

Displays compatible and incompatible build configurations with the reason of incompatibility.

Build runners

Lists build runners supported by build agent.

Agent logs

The page allows viewing and downloading the logs.

Agent Parameters

The tab lists system properties, environment variables, and configuration parameters. Refer to the [Configuring Build Parameters](#) page for more information on different types of parameters.

See also:

[Concepts: Build Agent | Run Configuration Policy](#)

[Installation and Upgrade: Installing and Running Additional Build Agents](#)

Viewing Build Agent Logs

- [Log Files](#)
- [Generic Debug Logging](#)
- [VCS Debug Logging](#)
- [Specific Debug Logging](#)
- [Advanced Logging Configuration](#)

To analyze agent-specific cases, there are internal log files saved by the TeamCity agent process into `<TeamCity agent home>/logs` directory on the agent machine.

When the agent is connected to TeamCity server, you can browse 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>teamcity-vcs.log</code>	VCS-related logging (for checkout mode "Automatically on 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

Generic Debug Logging

To enable general debug logging on agent, change "jetbrains.buildServer" category logging priority in the `<agent home>/conf/teamcity-agent-log4j.xml` file:

```
<category name="jetbrains.buildServer">
    <priority value="DEBUG"/>
    <appender-ref ref="ROLL"/>
</category>
```

Then, see `teamcity-agent.log*` files
To turn debug mode off, make the line "`<priority value="INFO"/>`".

VCS Debug Logging

To enable detailed VCS logging on agent, change VCS category logging priority in the `<agent home>/conf/teamcity-agent-log4j.xml` file:

```
<category name="jetbrains.buildServer.VCS">
    <priority value="DEBUG"/>
    <appender-ref ref="ROLL.VCS"/>
</category>
```

Then, see `teamcity-vcs.log*` files
To turn debug mode off, make the line "`<priority value="INFO"/>`".

Specific Debug Logging

To get dump of the data sent from the agent to the server, enable agent XML-RPC log, by uncommenting the line below in the `<agent home>/conf/teamcity-agent-log4j.xml` file.

```
<category name="jetbrains.buildServer.XMLRPC">
    <!--<priority value="DEBUG"/>-->
    <appender-ref ref="ROLL.XMLRPC"/>
</category>
```

Then, see `teamcity-xmlrpc.log`
To turn it off, make the line "`<priority value="INFO"/>`".

Advanced Logging Configuration

You can configure location of the logs by altering the value of **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.

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**.

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:

TeamCity		288 877 MB	1%	265 862 MB	10 005 MB	
↳	My Team	TeamCity Team (Builds 6.1)		314 978 MB	-%	
↳	Integration Tests			24 348 MB	+1%	
↳	UI Tests			55 699 MB	+1%	
↳	V3 Tests			5 554 MB	+1%	
↳	V2 Tests			1 740 MB	+1%	
↳	UI/UX Plugin Tests			2 448 MB	+1%	
↳	Server Code Analysis			108 MB	+1%	
↳	Commerce Live Payment Tests			247 MB	+1%	
↳	Plane Revenue Tests			245 MB	+1%	
↳	OD Tests			109 MB	+1%	
↳	Mobile Runtime Tests			32 MB	+1%	
Configuration Team City → Gazebo Trunk → More build details						
Builts on disk:	221	with artifacts:	118			
Average artifacts:	22.64 MB	avg. size:	1.13 MB			
Replies:	112 nodes	(43/62) CI errors on disk				
Top 10 causes for artifacts being out of date:	Stage					
↳	stage 01	23.0007	21.24 MB	>100%	↳ Incomplete	+1%
↳	stage 01	01.01.2017	1.81 MB	>100%	↳ Exp.	
↳	stage 01	02.02.2017	1.37 GB	>100%	↳ Exp.	
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
↳	stage 01	02.02.2017	1.36 GB	>100%	↳ Incomplete	+1%
Cleanup rules for TeamCity → Gazebo Trunk → More details						
↳	MaxBuildField	Clean everything more than 365 days older than the last build		→ 111		
↳	Old Images			→ 116		
↳	Old Artifacts			→ 116		
↳	Old Artifacts	Clean oldest more than 5 days older than the last build		→ 116		
↳	Old Artifacts	Clean artifacts more than 5 days older than the last build, except artifacts > 1 MB		→ 116		
↳	Old Artifacts	Do not prevent dependency artifacts cleaning		→ 116		
Go to the Change Log						
↳	Navigation					

Server Health

The **Server Health** report contains results of the server inspection for any configuration issues which impact or could potentially impact the performance. Such issues, the so called server health items, are collectively reported by TeamCity on the **Server Health** page in the **Administration** area.

The Project Administrator permissions at least are required to see the report.

Scope and Severity

The report enables you to define the analysis **scope**: you can select to analyze the global configuration or report on project-related items. The scope available to you depends on the level of the **View Project** permission granted to you. *Note that the report is not available for archived projects.*

The Server Health analysis also employs the **severity** rating, depending on the issue impact on the configuration of the system on the whole or an individual project.

Visibility Options

By default, the warning and error level results pertaining to the global configuration will be displayed on the report page as well as at the top of each page in TeamCity.

Besides those, some items will be displayed in-place: depending on the object causing the issue, the server health item will be reported on the Build Configuration, Template or VCS root settings page.

Only active items are displayed on the TeamCity pages. To remove an item from display, use the **hide** option next to an item on the report page. For global items, this option is available in every server health message.

Hidden items will be removed from the TeamCity pages, and will be displayed on the Server Health page below the active items. Their description will be greyed out.

To return an item to display, use the **Show** option

The visibility changes will be listed on the **Audit** page in the **Administration** area.

Issue Categories

Currently, TeamCity alerts you to the following configuration issue categories:

Global Configuration Items

TeamCity displays a notification on the availability of the new TeamCity version and a prompt to upgrade.

TeamCity displays a notification on the availability of the new TeamCity version and a prompt to upgrade. A warning is displayed if any of the licenses are incompatible with this new version. The notification is visible to system administrators only and they can use the link in the "Some Licenses are incompatible" message to quickly navigate to the [Licenses](#) page, where all incompatible licenses will have a warning icon.

WebSocket connection issues

Since TeamCity 9.0, the WebSocket protocol is used to get web UI updated for events, running builds updates and statistics counters.

In case of any problems preventing WebSocket connection from working, a warning will be displayed. TeamCity will automatically switch to the legacy update mode (usual HTTP request polling used by TeamCity before version 9.0) and you will be able to continue using TeamCity in this mode.

 However, it is **recommended** to make the following adjustments to benefit from faster web UI updates as well as reduced unnecessary network traffic and latency.

Proxy Server Configuration

If a reverse proxy is used in front of the TeamCity server, it needs to be [configured](#) to support the WebSocket protocol.

All URLs used by browsers that do not support the WebSocket connection are listed in the corresponding health report.

Bio Connector

If Tomcat is configured to use the BIO connector, the WebSocket protocol is [automatically disabled](#). It is recommended to change the Tomcat Connector settings [to use the NIO connector](#).

Critical Errors

This category shows the following errors:

- errors in project configuration files - occur if the server detects some inconsistency or a broken configuration while it loads configuration files from the disk
- errors raised by [the disk space watcher](#)
- warnings from the TeamCity Server [Memory Monitor](#)

Database Related Problems

TeamCity will warn you if the server currently uses the internal database. [A standalone database is recommended as the storage engine](#).

As [TeamCity does not support Sybase as an external database](#), a warning message will be displayed if you are using Sybase.

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

Since TeamCity 9.0 EAP1, TeamCity displays a warning if a rule of a VCS Trigger or Schedule Trigger references a VCS root which is not attached to any build configuration.

Redundant Trigger

The report will show cases when a build trigger is redundant, for example:

- There are two build configurations **A** and **B**
- **A** snapshot depends on **B**
- Both have VCS triggers, **A** with the [Trigger on changes in snapshot dependencies](#) option enabled.

In this case, the VCS trigger in **B** is redundant and causes builds of **A** to be put into the queue several times.

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

TeamCity analyzes the current settings of a build configuration and suggests additional options, e.g. adding a VCS trigger, a build step, etc. Besides the server health report, the suggestions for a specific build configuration are shown right on the configuration settings pages.

Extensibility

The default Server Health report provided by TeamCity might cover either too many, or not all the items required by you. Depending on your infrastructure, configuration, performance aspects, etc. that you need to analyze, a custom Server Health report can be needed. TeamCity enables you to write a [plugin](#) which will report on specific items.

Build Time Report

Since TeamCity 9.0, the **Build Time** report is available providing comparative statistics of the build time taken up by TeamCity projects and build configurations.

To see the report, open the TeamCity **Administration** area and select **Build Time** from Project-related Settings section.

Period Settings

The report displays total build duration for the server allowing you to select the period to be analyzed:

- the last 24 hours
- last week
- last month.

Grouping and Sorting

You can group the build configuration by project to display the duration of builds for individual projects with the percentage in relation to the parent project. The scope can be further drilled down to the build time of an individual build configuration.

Without grouping the report shows the build time for individual build configurations with the percentage in relation to the total server build time. It is

possible to include archived projects in the report.

Clicking the column headers allows sorting the report results.

TeamCity Monitoring and Diagnostics

TeamCity provides a variety of diagnostic tools and indicators to monitor and troubleshoot the server, which are accessible from the [Administration|Diagnostics](#) page.

The tools make it easier to identify and investigate problems and, if needed, [report issues](#) on your server.

The following options are available to you:

- [Troubleshooting](#)
 - [Debug Logging](#)
 - [Hangs and Thread Dumps](#)
 - [OutOfMemory and Performance](#)
 - [Java Configuration](#)
- [VCS status](#)
- [Server logs](#)
- [Internal Properties](#)
- [Logging Presets](#)
- [Caches](#)
- [Browse Data directory](#)
- [Search](#)

Troubleshooting

This section provides a number of indicators helping you to detect and address issues with the TeamCity server performance.

Debug Logging

By default, the debug logging is turned off. In this section you can enable [debug logging on the server](#) and select the debug logging level. After the server restart, the debug messages will appear in the corresponding TeamCity log files.

Hangs and Thread Dumps

The [server thread dump](#) can be viewed in the browser or saved to a file.

OutOfMemory and Performance

This section displays the data provided by the [TeamCity Memory Monitor](#), which regularly checks available memory and submits a warning if the [memory or CPU usage](#) grows too high.

If you experience memory problems, this section provides an option to dump a memory snapshot.

See also information on configuring [memory settings](#) for the TeamCity server.

Java Configuration

This section informs you on the Java [installed on your server](#) and the configured [JVM options](#).

VCS status

This tab displays the information on the monitored VCS roots, including their checking for changes status and duration. You can filter the available VCS roots by checking for changes duration.

Server logs

This tab allows you to view and download the available [TeamCity server logs](#), as well as saved thread dumps and memory dumps.

Internal Properties

This tab displays the internal properties affecting the TeamCity behavior and the JVM system properties. To fine-tune these aspects, see [this section](#).

Logging Presets

TeamCity uses the `log4j` library for the logging, and its settings can be customized. In this section you can view and download the available presets, as well as upload new presets, which can then be enabled on the [Diagnostics|Troubleshooting|Debug Logging](#).

It is also possible to change the logging configuration [manually](#).

Caches

This tab shows you the caches of the TeamCity processes stored in `TeamCity Data Directory/system/caches`. Resetting some caches is performed by the server during the clean-up automatically, but sometimes you might need to clear caches manually sign the reset link.

`vcsContentCache` - TeamCity maintains `vcsContentCache` cache for the sources to optimize communications with the VCS server. The caches are reset during the cleanup time. To resolve problems with sources update, the caches may need to be reset manually.

`search` - resetting this cache is required when enabling [search by build log](#).

`git` - contains the bare clone of the remote [Git](#) repository used by TeamCity.

`buildsMetadata` - resetting this cache is required to [reindex](#) the TeamCity NuGet feed.

Browse Data directory

This tab shows the files in the [TeamCity Data Directory](#) and allows you to upload new files.

Search

Displays information on the TeamCity data index related to the [search](#).

Managing Projects and Build Configurations

In this section:

- [Creating and Editing Projects](#)
- [Working with Meta-Runner](#)
- [Archiving Projects](#)
- [Creating and Editing Build Configurations](#)
- [Copy, Move, Delete Build Configuration](#)
- [Ordering Projects and Build Configurations](#)
- [Working with Feature Branches](#)
- [Triggering a Custom Build](#)
- [Ordering Build Queue](#)
- [Muting Test Failures](#)
- [Changing Build Status Manually](#)
- [Customizing Statistics Charts](#)
- [Storing Project Settings in Version Control](#)

See also:

[Concepts: Project | Build Configuration](#)

Creating and Editing Projects

On this page:

- Create project
 - Create Project from URL
 - Create Project
- Manage Project
 - Copy Project
 - Move Project
 - Archive Project
 - Delete Project

Create project

To create a new project:

On the **Administration | Projects** page select one of the options:

- Create project from URL
- Create project

To create a subproject, go to the parent project settings page and create a subproject manually or automatically from a URL.

Create Project from URL

1. Click the **Create project 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 Project** 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.

3. Click **Create**. An empty project is created.



To configure an existing project, select the desired project in the list, and click the **Edit Project Settings** link on the right.

4. Create build configurations (select build settings, configure VCS settings, and choose build runners) for the project.
5. Assign build configurations to specific build agents.

Manage Project

You can view all available projects and subprojects on the **Projects Overview** page listed in alphabetical order by default. **Starting from**

TeamCity 9.1., administrators can customize the default order.

To copy, move, delete or [archive](#) a project, use the **Actions** menu in the top right of the project settings page or **More** menu next to the project on the Root project settings page. These options are not available for the Root project.

Copy Project

Projects can be copied and moved to another project by project administrators.

A copy duplicates all the settings, subprojects, build configurations and templates of the original project, but no data related to builds is preserved. The copy is created with the empty build history and no statistics.

You can copy a project into the same or another parent.

On copying, TeamCity automatically assigns a new name and ID to the copy. It is also possible to change the name and ID manually. Selecting the **Copy project-associated user, agent and other settings** option makes sure that all the settings like notification rules or agent's compatibility are exactly the same for the copied and original projects for all the users and agents affected.

 When running TeamCity in the [Professional mode](#), the **Copy** option will not be displayed for a project if the number of build configurations on the server after copying will exceed the limit (20 unless you purchased additional Build Agent licenses).

Move Project

When moving a project, TeamCity preserves all its settings, subprojects, build configurations/templates and associated data, as well as the [build history](#).

Archive Project

Please refer to the dedicated [page](#).

Delete Project

When you delete a project, TeamCity will remove its .xml configuration files. After the deletion, the project is moved to the <TeamCity Data Directory>/config/_trash/.ProjectID.projectN directory. There is a [configurable](#) timeout (24 hours by default) before all project-related data (build history, artifacts, and so on) of the deleted project is completely removed during the next build history clean-up. You can [restore](#) a deleted project before the clean-up is run.

 If you attempt to delete a project with [dependent build configurations](#) from other projects, TeamCity will warn you about it. If you proceed with the deletion, the dependencies will no longer function.

See also:

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

Working with Meta-Runner

A *Meta-Runner* allows you to extract build steps, requirements and parameters from a build configuration and create a [build runner](#) out of them. This build runner can then be used as any other build runner in a build step of any other build configuration or template.

Basically, a meta-runner is a set of build steps from one build configuration that you can reuse in another; it is an xml definition containing build steps, requirements and parameters that you can utilize in xml definitions of other build configurations. TeamCity allows extracting meta-runners using the web UI.

With a meta-runner, you can easily reuse existing runners, create new runners for typical tasks (e.g. publish to FTP, delete directory, etc.), you can simplify your build configuration and decrease a number of build steps.

All meta-runners are stored on a project level, so they are available within this project and its subprojects only, and are not visible outside. If a meta-runner is stored on the **<Root project>** level, it is available globally (in all projects).

You can use the existing meta-runners from the TeamCity Meta-Runners Power Pack or create your own meta-runner.

On this page:

- [Using Meta-Runners Power Pack](#)
 - [Installing Meta-Runner](#)
- [Creating Meta-Runner](#)
 - [Preparing Build Configuration](#)
 - [Verifying Build Configuration Works Properly](#)

- Extracting and Using Meta-Runner
- Creating Meta-Runner from XML Definition of Build Configuration
- Creating Build Configuration from Meta-Runner

Using Meta-Runners Power Pack

Meta-runners Power Pack for TeamCity available on GitHub is a collection of meta-runners for various tasks like downloading a file, triggering a build, tagging a build, changing a build status, running PHP tasks, etc.

Each file called `*MRPP_<some text>.xml` contains a definition of a single Meta-runner. Download the required meta-runner (or copy its definition to a file) and install it as described in the section below.

Installing Meta-Runner

Since TeamCity 9.0, you can install a meta-runner using the TeamCity Web UI. Alternatively, you can do it directly via the file system.

- to install a Meta-runner via the Web UI, go to the Project Settings page, select Meta-Runners from the list of settings on the left, click **Upload Meta-Runner** and select the Meta-runner definition file. Save your changes.
- to install a Meta-runner directly to the file system, take the Meta-runner definition file and put it into the `TeamCity Data Directory\config\projects\<project ID>\pluginData\metaRunners` directory, where `\<Project ID\>` is the identifier of a project where you want to place the Meta-runner. If the `metaRunners` directory does not exist, it should be created. Once you place the file on the disk, TeamCity will detect it and load this Meta-runner; no server restart is required.

Creating Meta-Runner

Let us consider an example of creating a meta-runner.

To create a meta-runner, follow these steps (described below in more detail):

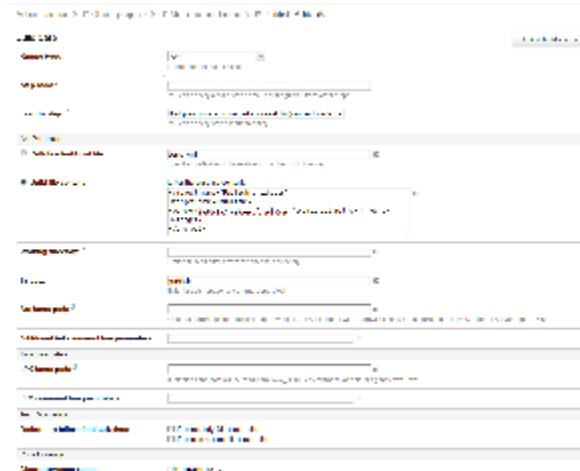
1. prepare a build configuration to test the build steps we are going to use in our meta-runner,
2. make sure the build configuration is working,
3. extract a meta-runner to the desired project.

In this example, we will create a meta-runner to publish some artifacts to TeamCity with the help of corresponding service message.

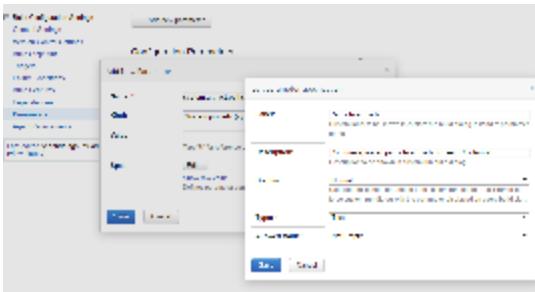
Usually artifacts configured in a build configuration are published when the build finishes. However, sometimes for long builds with multiple build steps we need artifacts faster. In this example, we will create a runner which can be inserted between any build steps and can be configured to publish artifacts produced by previous steps.

Preparing Build Configuration

The first step is to prepare a build configuration which will work the same way as the meta-runner we would like to produce. Let us use the configuration with a single Ant build step: Ant can be executed on any platform where the TeamCity agent runs; besides, Ant runner in TeamCity supports build.xml specified right in the runner settings. This is important because our build configuration must be self-contained, it cannot take build.xml from the version control repository. So in our case the Ant step settings will look like this:



where `artifact.paths` is a system property. We need to add it on the **Parameters** tab of the build configuration settings:

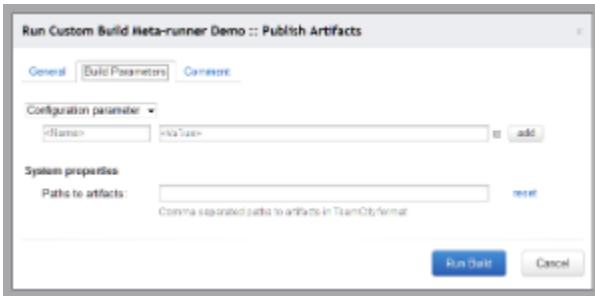


Note that each parameter can have a specification where we can provide the label, description, type of control and specify validation conditions. Before version 8.0 this specification was used by the custom build dialog only. Now this specification is used by a meta-runner too.

i Here the Ant build step is used just as an example. In the initial build configuration, you can use any of the available build runners (e.g. MSBuild, .Net process, etc.) and configure the settings, define the parameters for this build step. When you extract a meta-runner from this build configuration, all the settings defined in the build step, and all the build parameters will be added to the meta-runner.

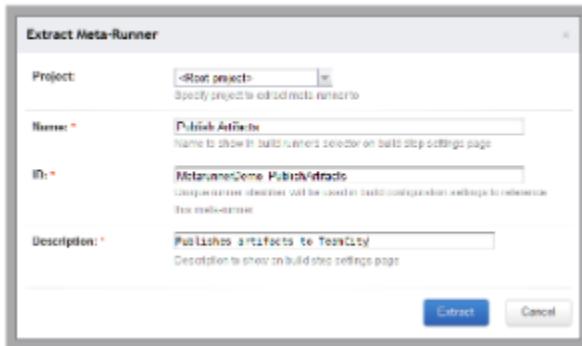
Verifying Build Configuration Works Properly

Once the build steps and parameters are defined, we need to make sure our build configuration works by running a couple of builds through the custom build dialog:



Extracting and Using Meta-Runner

If the build configuration works properly, we can create a meta-runner using the **Actions** button in the top right-hand corner, **Extract meta-runner...** option:



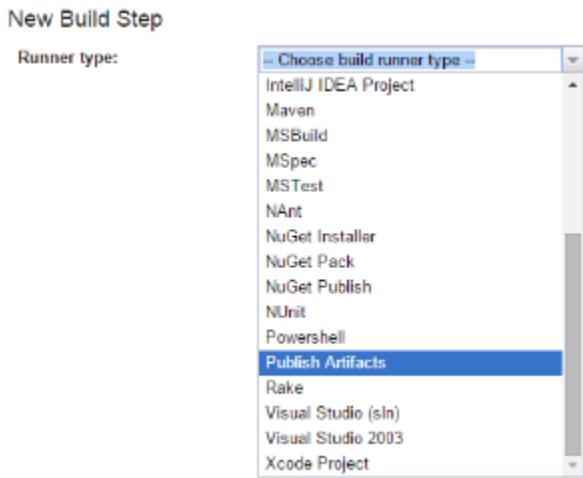
The **Extract Meta-Runner** dialog requires specifying the project where the meta-runner will be created. A meta-runner created in a project will be available in this project and all its subprojects. In our case the **<Root project>** is selected, so the meta-runner will be available in all projects.

We also need to provide the name, description and an ID for the meta-runner: the name and description will be shown in the web interface, an ID is required to distinguish this meta-runner from others.

Upon clicking the **Extract** button, TeamCity will take definitions of all build steps and parameters in this build configuration and create a build runner out of them.

i Besides build steps and parameters, a meta-runner can also have requirements: if requirements are defined in the build configuration, they will be extracted to the meta-runner automatically. Requirements can be useful if the tools used by meta-runner are available on specific platforms only.

Once the meta-runner is extracted, it becomes available in the build runners selector and can be used in any build step just like any other build runner:



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

When a meta-runner is extracted, all steps will be extracted. If you need to reorder parameters or make some quick fixes in the runner script, you can edit its raw xml definition in the web browser: go to the **Administration** page of the project -> **Meta-Runners** and use the **Edit** option next to the meta-runner. The parameters will be shown in the same order as the `<param>` elements in the xml definition. Definitions of meta-runners are stored in the `TeamCity Data Directory \config\projects\<project ID>\pluginData\metaRunners` folder.

Creating Meta-Runner from XML Definition of Build Configuration

Alternatively, you can use the xml definition of an existing build configuration as a meta-runner. To do it, save the definition of this build configuration to a file named as follows: `<runner id>.xml`, where `<runner id>` is the **ID** of this build runner. Install the meta-runner as described above.

Since a meta-runner looks and works like any other runner, it is also possible to create another meta-runner on the basis of an existing meta-runner.

Creating Build Configuration from Meta-Runner

Since TeamCity 9.0, if you need to fix a meta-runner and test your fix, you can create a build configuration from a meta-runner, can change its steps, adjust parameters and requirements, check how it works, and then use the **Extract meta-runner** action to apply the changes to the existing meta-runner with the same ID.

Archiving Projects

If a project is not in an active development state, you can *archive* it using the **Actions** menu in the top right of the project settings page. On the **Archive project** screen, you can choose whether to cancel the builds which are already in the queue using the corresponding option (enabled by default).

When *archived*:

- All the subprojects of the current project are archived as well.
- All project's build configurations are automatically **paused**.
- Automatic checking for changes in the project's **VCS roots** is not performed if the VCS roots are not used in other non-archived projects.
- As part of pausing, automatic build triggering is disabled. However, builds of the project can be triggered manually or automatically as a part of a build chain.
- All data (settings, build results, artifacts, build logs, etc.) of the project's build configurations are preserved - you can still edit settings of the archived project or its build configurations.
- Archived projects do not appear in most user-facing projects lists and in IDEs including the list of build configurations for remote run.

By default, permissions to archive projects are given to project and system administrators.

If you need to unarchive a project, you can do it using the **Actions** menu in the top right of the project settings page or the **More** menu next to the

project on the Root project settings page.

See also:

[Concepts: Project | Build Configuration](#)

Creating and Editing Build Configurations

TeamCity provides several ways to create a new build configuration for a project. You can:

- Create a new build configuration manually
- Create a new build configuration from URL (since **TeamCity 9.1**)
- Create a build configuration template, and then create a build configuration based on the template.
- **i Deprecated since TeamCity 8.1** Create a Maven build configuration by importing settings from `pom.xml`

Creating New Build Configuration

1. 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**.
2. Specify **general settings** for the build configuration, click **Create**.
3. Proceed with creating other settings:
 - **Create/edit VCS roots and specify VCS-specific settings**
 - On the **Build Steps** page, configure build steps either manually by selecting a desired build runner from the drop-down list, or using the automatic detection. [More information](#). Click **Save**. You can add as many build steps as you need within one build configuration. Note, that they will be executed sequentially. In the end, the build gets the merged status and the output goes into the same build log. If some step fails, the rest is executed or not depending on their **step execution policy**.
 - Additionally, configure **build triggering options, dependencies, properties and variables** and **agent requirements**.

Creating New Build Configuration from URL

Since **TeamCity 9.1**, you can create a build configuration using a VCS URL:

1. On the project settings page, click **Create build configuration from URL**.
2. Specify **settings** for the build configuration, click **Create**.

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 from the **Actions** menu at the top right corner of the screen.

Creating Build Configuration From Template

To create a build configuration based on a template:

1. On the Project settings page locate the desired template and click its name or use **Edit**.
2. Click the **Create Build Configuration** button available from the **Actions** button at top right corner of the screen.
3. Specify a name for the new configuration.
4. The settings set up in the template cannot be edited in a configuration created from this template. However, some of them can be [redefined in a build configuration](#). Note that modifying the settings of the template itself will affect all configurations based on this template.

Alternatively, you can create a build configuration based on a template by clicking **Create build configuration** and selecting a template from the **Based on template** drop-down on the configuration settings page.

Creating Maven Build Configuration (Deprecated)

i Deprecated. Since **Teamcity 8.1**, use the [create project from url](#) option.

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
```

Ordering Build Configurations

You can view all build configurations of a project on the [Project Overview](#) page listed in alphabetical order by default. **Starting from TeamCity 9.1**, administrators can [customize the default order](#).

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 <ul style="list-style-type: none"> • Hanging Build Detection • Allow Triggering Personal Builds • Enable Status Widget <ul style="list-style-type: none"> • HTML Status Widget • Limit Number of Simultaneously Running Builds 	Additional options for this build configuration. For more information, refer to the following sections below: <ul style="list-style-type: none"> • Hanging Build Detection • Allow Triggering Personal Builds • 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.

If you have a build finished on an agent, you can use the checkout directory browser and select artifacts from the 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**

p, .7z, .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 => symbols 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.

Allow Triggering Personal Builds

Since TeamCity 9.1, you can restrict running personal builds by unchecking the **allow triggering personal builds** option (on by default).

Enable Status Widget

This option enables retrieving the status and basic details of the last build in the build configuration without requiring any user authentication. Please note that this also allows getting the status of any specific build in the build configuration (however, builds cannot be listed and no other information except the build status (success/failure/internal error/cancelled) is available).

The status can be retrieved via the HTML status widget described below, or via a single icon with the help of [REST API](#).

HTML Status Widget

This feature allows you to get an overview of the current project status on your company's website, wiki, Confluence or any other web page. When the **Enable status widget** option is enabled, an HTML snippet can be included into an external web page and will display the current build configuration status.

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 :

- 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](#).
- 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.
- Define how project sources reach an agent via [VCS Checkout Mode](#).
- Additionally, you can add a label into the version control system for the sources used for a particular build by means of [VCS Labeling](#) build feature.

See also:

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

Configuring VCS Roots

VCS Roots in TeamCity

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

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

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

Common VCS Root Properties

Property	Description
Type of VCS	Type of Version control system supported by TeamCity, for example, Perforce, Subversion, etc
VCS root name	Unique name of VCS root across all VCS roots of the project.
VCS root ID	Unique 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. <div style="border: 1px solid #f0e68c; padding: 5px; margin-left: 20px;"> Some public servers may block access if polled too frequently.</div>
Belongs to project	Each VCS root belongs to some project, and in this section the name of this project is displayed. A VCS root can be moved to the common parent project of all subprojects, build configurations and templates where the root is currently used.

Please refer to the following pages for VCS-specific configuration details:

- [Guess Settings from Repository URL](#)
- [ClearCase](#)
- [CVS](#)
- [Git](#)
- [Mercurial](#)
- [Perforce](#)
- [StarTeam](#)
- [Subversion](#)
- [Team Foundation Server](#)
- [SourceGear Vault](#)
- [Visual SourceSafe](#)



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://• 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 +Maven-like urls: http://maven.apache.org/scm/subversion.html• svn://
TFS	<ul style="list-style-type: none">• http url: <code>http://<tfs server>:<port>/<collection name>\$/<project path></code>• https url for MS Visual Studio Online: <code>https://<url to visualstudio.com>/DefaultCollection\$/<project path></code>. Also see the related blog post.
Perforce	<ul style="list-style-type: none">• Maven-like urls: http://maven.apache.org/scm/perforce.html• same as Maven but without the '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

An installed ClearCase client on the TeamCity server is required to make TeamCity ClearCase integration work. You also need to create a ClearCase view on the TeamCity server machine (regardless of whether you plan to use the server-side or agent-side checkout). This view will be used for collecting changes in ClearCase VCS roots and for checkout in case of the server-side checkout mode. In case of the agent-side checkout mode, the config spec of this view will be also used to automatically create views on agents.



If you plan to use the agent-side [checkout mode](#), make sure the ClearCase client (cleartool) is also installed on the agents and is properly configured, i.e. the tool must be in system paths and have access to the ClearCase Registry.

ClearCase Settings

Common VCS Root properties [are described here](#). The section below contains description of the fields and options specific to the ClearCase Version Control System.

Option	Description
--------	-------------

ClearCase view path	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.  "lshistory" options can be customized, see TW-12390
Use ClearCase	Use the drop-down menu to select either UCM or BASE.
Global labeling	Check this option if you want to use global labels
Global labels VOB	Pathname of the VOB tag (whether or not the VOB is mounted) or of any file system object within the VOB (if the VOB is mounted)

 Make sure that the user that runs the TeamCity server process is also a ClearCase view owner.

See also:

[Administrator's Guide: VCS Checkout Mode](#)

CVS

Common VCS Root properties [are described here](#).

This page contains descriptions of CVS-specific fields and options available when setting up a VCS root. Depending on the selected access method, the page shows different fields that help you to easily define the [CVS settings](#):

- [CVS Root](#)
- [Checkout Options](#)
- [PServer Protocol Settings](#)
- [Ext Protocol Settings](#)
- [SSH Protocol Settings \(internal implementation\)](#)
- [Local CVS Settings](#)
- [Proxy Settings](#)
- [Advanced Options](#)

CVS Root

Option	Description
Module name	Specify the name of the module managed by CVS.
CVS Root	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 (described below): <ul style="list-style-type: none"> • pserver • ext • ssh • local

Checkout Options

Option	Description
<ul style="list-style-type: none"> • Checkout HEAD revision • Checkout from branch • Checkout by Tag 	Define the way CVS will fill in and update the working directory
Quiet period	Since there are no atomic commits in CVS, using this setting you can instruct TeamCity not to take (detect) a change until the specified period of time passed since the previous change was detected. It helps avoid situations when one commit is shown as two different changes in the TeamCity web UI.

PServer Protocol Settings

Option	Description
CVS Password	Click this radio button if you want to access the CVS repository by entering a password.
Password File Path	Click this radio button to specify the path to the <code>.cvspass</code> file.
Connection Timeout	Specify the connection timeout.

Ext Protocol Settings

Option	Description
Path to external rsh	Specify the path to the rsh program used to connect to the repository.
Path to private key file	Specify the path to the file that contains user authentication information.
Additional parameters	Enter any required rsh parameters that will be passed to the program. Multiple parameters are separated by spaces.

SSH Protocol Settings (internal implementation)

Option	Description
SSH version	Select a supported version of SSH.
SSH port	Specify SSH port number.
SSH Password	Click this radio button if you want to authenticate via an SSH password.
Private Key	Click this radio button to use private key authentication. In this case specify the path to the private key file.
SSH Proxy Settings	See proxy options above .

Local CVS Settings

Option	Description
Path to the CVS Client	Specify the path to the local CVS client.
Server Command	Type the server command. The server command is the command which makes the CVS client to work in server mode. Default value is <code>'server'</code>

Proxy Settings

Option	Description
Use proxy	Check this option if you want to use a proxy, and select the desired protocol from the drop-down list.
Proxy Host	Specify the name of the proxy.

Proxy Port	Specify the port number.
Login	Specify the login.
Password	Specify the password.

Advanced Options

Option	Description
Use GZIP compression	Check this option to apply gzip compression to the data passed between the CVS server and the CVS client.
Send all environment variables to the CVS server	Check this option to send environment variables to the server for compatibility with certain servers.
History Command Supported	Check this option to use the history file on the server to search for newly committed changes. Enable this option to improve the CVS support performance. By default, the option is not checked because not every CVS server supports keeping a history log file.

Git

Git support in TeamCity is implemented as a plugin. Git source control with Visual Studio Online is supported.

This page contains description of the Git-specific fields of the VCS root settings.
Common VCS Root properties [are described here](#).



Git needs to be installed on the agents if the [agent-side checkout](#) is used.



Important Notes

- [Remote Run](#) and [Pre-Tested Commit](#) are supported in the [IntelliJ IDEA](#) and [Eclipse](#) plugins; with the [Visual Studio Addin](#) use the [Branch Remote Run Trigger](#).
- Initial Git [checkout](#) may take significant time (sometimes hours), depending on the size of your project history, because the whole project history is downloaded during the initial checkout.

On this page:

- [Visual Studio Online](#)
 - [Enabling Alternate Credentials](#)
- [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](#)

Visual Studio Online

When configuring a VCS Root for a project hosted on Visual Studio Online, you have to enable alternate credentials support in your Visual Studio Online account.

Enabling Alternate Credentials

Applications that work outside the browser, such as the git client or the command-line TFS client, require basic authentication credentials to work. You can edit the VSO profile to set a secondary username that can be used in this case.

After navigating to your VSO account portal, use the menu at the top-right to edit your profile. The **Credentials** tab allows enabling alternate credentials. After clicking **Enable alternate credentials**, set a secondary username and password to use when configuring a VCS Root.

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 addition to the branch specification, or when the branch specification is empty. Note that parameter references are supported here.
Branch specification	In this area list all the branches you want to be monitored for changes in addition to the default one. 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). Read more . Note that only one asterisk is allowed and each rule has to start with a new line, see the section below .
Use tags as branches	Allows you to use git tags in branch specification (e.g. <code>+ -:refs/tags/<tag_name></code>). 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.somwhereElse.org/repos/test.git`, scp-like syntax: `git@git.somwhere.org:repos/test.git`)



Be Careful

The scp-like syntax requires a colon after the hostname, while the usual ssh url does not. This is a common source of errors.

- git: (e.g. `git://git.kernel.org/pub/scm/git/git.git`)
- http: (e.g. <http://git.somewhere.org/projects/test.git>)
- file: (e.g. `file:///c:/projects/myproject/.git`)



Be Careful

When you run TeamCity as a Windows service, it cannot access mapped network drives and repositories located on them.

Authentication Settings

Authentication Method	Description
Anonymous	Select this option to clone a repository with anonymous read access.
Password	Specify a valid username (if there is no username in the clone URL; the username specified here overrides the username from the URL) and a password to be used to clone the repository. For the agent-side checkout , it is supported only if git 1.7.3+ client is installed on the agent. See TW-18711 . For Git hosted from Team Foundation Server 2013, specify NTLM credentials here.

<p>Private Key</p> <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.
Use mirrors	Since TeamCity 9.0 When enabled (default), TeamCity clones the repository on each agent and uses the mirror as an alternate repository. As a result, this speeds-up clean checkout (because only the working directory is cleaned), and saves disk space (as there is only one clone of the given git repository on an agent).

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 <code>git gc</code> during the server cleanup (native git is used)
teamcity.server.git.executable.path	git	The path to the native git executable on the server
teamcity.server.git.gc.quota.minutes	60	Maximum amount of time to run <code>git gc</code>
teamcity.git.cleanupCron	0 0 2 * * ? *	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.buildPatchInSeparateProcess	true	Since 9.0 git-plugin builds patches in a separate process, set it to false to build patch in the server process. To build patch git-plugin has to read repository files into memory. To not run out of memory git-plugin reads only objects of size smaller than the threshold, for larger objects streams are used and they can be slow (TW-14947). With patch building in a separate process all objects are read into memory. Patch process uses the memory settings of the separate fetch process.
teamcity.git.mirror.expiration.timeout.days	7	The number of days after which an unused clone of the repository will be removed from the server machine. The repository is considered unused if there were no TeamCity operations on this repository, like checking for changes or getting the current version. These operations are quite frequent, so 7 days is a reasonably high value.
teamcity.git.commit.debug.info	false	Defines whether to log additional debug info on each found commit
teamcity.git.sshProxyType		Type of ssh proxy, supported values: http, socks4, socks5. Please keep in mind that socks4 proxy cannot resolve remote host names, so if you get an UnknownHostException, either switch to socks5 or add an entry for your git server into the hosts file on the TeamCity server machine.
teamcity.git.sshProxyHost		Ssh proxy host
teamcity.git.sshProxyPort		Ssh proxy port
teamcity.git.connectionRetryAttempts	3	Number of attempts to establish connection to the remote host for testing connection and getting a current repository state before admitting a failure

teamcity.git.connectionRetryIntervalSeconds	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.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

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: <code>+ -:branch_name</code> , where <code>branch_name</code> is specific to the VCS (with the optional <code>*</code> placeholder). Note that only one asterisk is allowed and each rule has to start with a new line. Bookmarks can also be used in the branch and branch specification fields. If a bookmark has the same name as a regular branch, a regular branch wins. More in the related TeamCity blogpost . Bookmarks support requires Mercurial 2.4 installed on the TeamCity server.
Use tags as branches	Allows you to use tags in branch specification. By default, tags are ignored.
Detect subrepo changes	By default, subrepositories are not monitored for changes.
Username for tags/merge	A custom username used for labeling
Use uncompressed transfer	Uncompressed transfer is faster for repositories in the LAN.
HG command path	The path to the hg executable. See more below .
Mercurial config	Since TeamCity 9.0. Specify the Mercurial configuration options to be applied to the repository, e.g. enter the following to enable the <code>largefiles</code> extension: [extensions] largefiles = The configuration format is described here .

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 the value specified in the root settings.

Agent Settings

These are the settings used in case of the [agent-side checkout](#), which requires Mercurial installed on all agents.

Option	Description
Purge settings	Defines whether to purge files and directories not being tracked by Mercurial in the current repository. You can choose to remove only unknown files and empty directories, or to remove ignored files as well. Added files and (unmodified or modified) tracked files are preserved. More details on mercurial.selenic.com
Use mirrors	Since TeamCity 9.0 (off by default) When enabled, TeamCity creates a local agent mirror first and then clones to the working directory from this local mirror. This option speeds up clean checkout, because only the build working directory is cleaned. Also, if a single root is used in several build configurations, a clone will be faster.

Internal Properties

This section describes hg-related internal properties. You can modify the defaults to adjust the Mercurial settings as needed.

Server-side internal properties:

Property	Default	Description
<code>teamcity.hg.pull.timeout.seconds</code>	3600	Maximum time in seconds for pull operation to run
<code>teamcity.hg.server.path</code>	hg	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.pull.timeout.seconds</code>	3600	Maximum time in seconds for pull operation to run
<code>teamcity.hg.agent.path</code>	hg	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. Common VCS Root properties [are described here](#).



A Perforce client must be installed on the TeamCity server. Check [TeamCity and Perforce compatibility](#).

If you plan to use the agent-side [checkout mode](#), note that a Perforce client must be installed on the agents, and the path to the p4 executable must be added to the PATH environment variable.

- P4 Connection Settings
- Checkout On Agent Settings
 - Perforce Workspace Parameters
 - Perforce Proxy Settings
- Other Settings
- Compatibility issues

P4 Connection Settings

Option	Description
Port	Specify the Perforce server address. The format is <code>host:port</code> .

Stream	<p>Click this radio button to specify an existing Perforce stream. TeamCity will use this stream to prepare the stream-based workspace, and will use the client mapping from such a workspace. The format should be <code>//streamdepot/streamname</code>. Parameters are supported. For <code>StreamAtChange</code> option you may use the Label to checkout field.</p> <div style="border: 1px solid red; padding: 10px;"> <p>⚠ Performance impact When this option is used with the checkout on the server mode, the 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> <div style="border: 1px solid orange; padding: 10px;"> <p>⚠ Checkout rules limitations When Perforce Streams are used with the checkout on the agent, simple checkout rules like <code>. => sub/directory</code> are supported. Exclude checkout rules, multiple include rules, or rules like <code>aaa=>bbb</code> are not supported.</p> </div>
Client	<p>Click this radio button to directly specify the client workspace name. The workspace must be already created by a Perforce client application like P4V or P4Win. Only the mapping rules from the configured client workspace are used. The client name is ignored.</p> <div style="border: 1px solid red; padding: 10px;"> <p>⚠ Performance impact When this option is used with the checkout on the server mode, the 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. If you have Client mapping selected, TeamCity handles file separators according to the OS/platform of the build agent where a build is run. To enforce specific line separator for all build agents, use Client or Stream having LineEnd option specified in Perforce instead of Client mapping. Alternatively, you can add an agent requirement to run builds only on a specific platform.</p> <div style="border: 1px solid blue; padding: 10px;"> <p>ℹ Tip Use <code>team-city-agent</code> 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> <div style="border: 1px solid yellow; padding: 10px;"> <p>⚠ Editing the client mapping for a Perforce VCS root will result in a clean checkout before the next build. See the available workaround.</p> </div>
Username	Specify the user login name.
Password	Specify the password.
Ticket-based authentication	Check this option to enable ticket-based authentication.

Checkout On Agent Settings

When the **agent-side checkout** is used, TeamCity creates a Perforce workspace for each **checkout directory/VCS root**. These workspaces are automatically created when necessary and are automatically deleted after some idle time.

It is possible to customize the name generated by TeamCity: add the `teamcity.perforce.workspace.prefix` configuration parameter at the [Parameters](#) page with the prefix in the value.

Option	Description
Workspace options	If needed, you can set here the following options for the <code>p4 client</code> command: <code>Options</code> , <code>SubmitOptions</code> , and <code>LineEnd</code> .

Run 'p4 clean' for cleanup	Enable this option to clean up your workspace from extra files before a build (since p4 2014.1) When enabled, the <code>p4 clean</code> command will be run before <code>p4 sync</code> command, unless <code>p4 sync -f</code> or <code>p4 sync -p</code> is used. See p4 sync command reference .
Don't update server on sync	Enable this option to use <code>clean checkout</code> and skip updating the Perforce db.have file (<code>p4 sync -p</code>)
Extra sync options	Specify additional 'p4 sync' options, like <code>--parallel</code> . See p4 sync command reference .

Perforce Workspace Parameters

With checkout on agent, TeamCity provides environment variables describing the Perforce workspace created during the checkout process. If several Perforce VCS Roots are used for the checkout, the variables are created for the first VCS root.

The variables are:

- **P4USER** - same as `vcsroot.<VCS root ID>.user` parameter
- **P4PORT** - same as `vcsroot.<VCS root ID>.port` parameter
- **P4CLIENT** - name of the generated P4 workspace on the agent

These variables can be used to perform custom p4 commands after the checkout.

Perforce Proxy Settings

To allow using Perforce proxy with the [agent-side checkout](#), specify the `env.TEAMCITY_P4PORT` environment variable on the build agent and the agent will take this value as the `P4PORT` value.

Other Settings

Path to P4 executable	Specify the path to the Perforce command-line client: <code>p4.exe</code> file). This path will be used for both the server-side checkout and the agent-side checkout. If you need different values for this path on different build agents when using the agent-side checkout, you can set the value using the <code>TEAMCITY_P4_PATH</code> environment variable in the <code>buildAgent.properties</code> file
Label/revision to checkout	If you need to check out sources not with the latest revision, but with a specific Perforce label (with selective changes), you can specify this label here. For instance, this can be useful to produce a milestone/release build, or a reproduce build. If the field is left blank, the latest sources revision will be used for checkout. You can also specify a changelist number here. <div style="border: 2px solid red; padding: 5px; margin-top: 10px;">! It is recommended to use the agent-side checkout if you use symbolic labels. With the server-side checkout on label, TeamCity will perform full checkout.</div>
Charset	Select the character set used on the client computer.
Support UTF-16 encoding	Enable this option if you have UTF-16 files in your project.

Compatibility issues

See also:

[Administrator's Guide: VCS Checkout Mode](#)

Perforce VCS Compatibility

Perforce server version	TeamCity version	Comment
2009.1 .. 2015.1	9.1.1+	
2010.2 .. 2015.1	9.1.0	TW-41876
2009.1 .. 2015.1	8.1.x..9.0.x	
2009.1 .. 2013.2	8.0.5	
2009.1 .. 2014.1 (might work with later versions, not tested)	8.0.6+	TW-34128

2009.1 .. 2012.1	7.1.2	TW-24046
2009.1 .. 2013.2	7.1.3..8.0.5	

StarTeam

This page describes the fields and options available when setting up VCS roots using StarTeam.

Common VCS Root properties are described here.

- [StarTeam Connection Settings](#)
- [Notes on Directory Naming Convention](#)

StarTeam Connection Settings

Option	Description
URL	Specify the StarTeam URL that points to the required sources
Username	Enter the login for the StarTeam Server
Password	Enter the corresponding password for the user specified in the field above
EOL Conversion	Define the EOL (End Of Line) symbol conversion rules for text files. Select one of the following: <ul style="list-style-type: none"> • 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 Subversion-specific fields and options available when setting up a VCS root.
Common VCS Root properties are described here.

- [SVN Connection Settings](#)
- [SSH settings](#)
- [Checkout on agent settings](#)
- [Labeling settings](#)
- [Authentication for SVN externals](#)
- [Subversion 1.8 support](#)
- [Timeouts](#)
 - [Connection timeout](#)

- Read timeout
 - Subversion server access via HTTP/HTTPS (both server/agent)
 - Subversion server access via svn:// or svn+ssh://
- Miscellaneous

You do not need Subversion client to be installed on the TeamCity server or agents. TeamCity bundles the Java implementation of SVN client ([SVNKit](#)).

SVN Connection Settings

Option	Description
URL	Specify the SVN URL that points to the project sources.
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 unchecked, 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 the sources from the externals) and will gather and display information about externals' changes on 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 can use this option if you have several SVN externals and do not want to get information about any changes made in the externals' source files.
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, the entered username will never be used to authenticate when accessing any of its subfolders. Anonymous access will be used instead. This rule only applies for `svn://` and `http(s)://` protocols; i.e. if you have a build configuration which uses a combination of this VCS Root + [VCS Checkout Rules](#) referencing a non-restricted path above the restricted one for another build configuration, changes under the restricted path will be ignored even if you specify correct username/password for the VCS Root itself.

SSH settings

Option	Description
Private Key File Path	Specify the full path to the file that contains the SSH private key. Since TeamCity 9.1 , you can also specify an SSH key uploaded to TeamCity
Private Key File Password	Enter the password to the SSH private key.
SSH Port	Specify the port that SSH is using.

i The Putty private key format (*.ppk) is not supported by TeamCity. A Putty private key has to be converted to the OpenSSH format (you can use PuTTYgen.exe: see [Conversions -> Export OpenSSH key](#)). Enter the path to the OpenSSH-formatted private key into the **Private Key File Path** field.

Checkout on agent settings

Option	Description
--------	-------------

Working copy format	Select the format of the working copy. Available values for this option are 1.4 through 1.8 (current default) This option defines the format version of Subversion files located in <code>.svn</code> directories, when the checkout on agent mode is used. The specified format is important in two cases: <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 the checkout on agent mode.
Revert before update	If the option is selected, then TeamCity always runs the " <code>svn revert</code> " command before updating sources; that is, it will revert all changes in versioned files located in the checkout directory. When the option is disabled and local modifications are detected during the update process, TeamCity runs the " <code>svn revert</code> " after the update. TeamCity does not delete non-versioned files in the working directory during the revert. For deleting non-versioned files, consider using Swabra

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 does not allow specifying SVN externals authentication parameters explicitly, in user interface. To authenticate on the SVN externals server, the following approaches are used:

- authenticate using the same credentials (username/password) as for the main repository
- authenticate without explicit username/password. In this case, the credentials should be already available to the `svn` process (usually, they stored in subversion configuration directory). So, this require setting correct "Configuration Directory" or "Default Config Directory" option under [SVN Connection Settings](#)

When TeamCity has to connect to a SVN external, it uses the following sequence:

- if the SVN external URL has the same prefix as the main repository (there is a match > 20 characters), TeamCity tries the main repository credentials first, and in case of a failure tries to connect without the username/password (so they picked up from SVN configuration directory)
- if the SVN external URL noticeably differs from the main repository, TeamCity tries to connect without the username/password, and in case of a failure, tries using the credentials from the main repository

Subversion 1.8 support

Since TeamCity 8.1, Subversion 1.8 is fully supported. See also [Subversion 1.8 support by previous TeamCity versions](#).

Timeouts

Sometimes, the SVN checkout operation for remote SVN servers may fail with a error like `svn: E175002: timed out waiting for server`.

Usually this can happen due to network slowness or the SVN server overload.

The timeout values for the connection and for read operations can be configured.

Connection timeout

Connection timeout is applied when TeamCity creates a connection to the SVN server. The default timeout for this operation is **60 seconds**, and can be specified via the TeamCity internal property `teamcity.svn.connect.timeout`, in seconds. The value of the property is set differently for server-side checkout and agent-side checkout:

- Server-side operations - [configure internal property](#)
- Agent-side checkout - [add start-up property](#)

Read timeout

The read timeout is used when a connection with the SVN server is established, and TeamCity is waiting for the data from the server. The value of the timeout depends on the SVN server access protocol.

Subversion server access via HTTP/HTTPS (both server/agent)

For HTTP read timeout TeamCity uses the `http-timeout` setting specified in the `servers` file in the Subversion configuration directory. On Win32 systems, this directory is typically located the Application Data area of the user's profile directory. On Unix/Mac, this directory is usually named `$HOME/.subversion` for the user account who runs the TeamCity server/agent.

If not specified, the default value for the timeout is 1 hour.

Subversion server access via svn:// or svn+ssh://

In this case the read timeout can be specified in seconds via the TeamCity internal property `teamcity.svn.read.timeout`. The default value is 30 minutes. The value of the property is set differently for server-side checkout and agent-side checkout:

- Server-side operations - [configure internal property](#)
- Agent-side checkout - [add start-up property](#)

Miscellaneous

Directories are not considered changed when they have the "svn:mergeinfo" Subversion property changes only. See [details](#).

See also:

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

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 and newer
1.8 working copy (checkout on agent) TW-20404	✗	✗	✓
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 to connect to Microsoft Team Foundation Server Version Control.

Common VCS Root properties [are described here](#).

When connecting to a Visual Studio Online Git repository, select [Git](#) as Type of VCS.



TFS integration is only supported for Windows machines. It is also required to have [Team Explorer](#) installed and correctly functioning on the TeamCity server.

If you want to use the [agent-side checkout mode](#), Team Explorer should also be installed on all the agents which will run the related builds.

See this [section](#) for the supported versions.

On this page:

- [TFS Settings](#)
- [Agent-Side Checkout](#)
- [Visual Studio Online](#)
 - [Enabling Alternate Credentials](#)
 - [Connecting to Visual Studio Online TFVC](#)

TFS Settings

Option	Description
URL	Team Foundation Server URL in the following format: TFS 2010+: <code>http[s]://<TFS Server>:<Port>/tfss/<Project Collection Name></code> TFS 2005/2008: <code>http[s]://<TFS Server>:<Port></code>

Root	Specify the root using the following format: \$<project name><project catalogue>
UserName	Specify a user to access Team Foundation Server. This can be a user name or DOMAIN\UserName string. Use blank to let TFS select a user account that is used to run the TeamCity Server (or Agent for the agent-side checkout)
Password	Enter the password of the user entered above.

Agent-Side Checkout

The [agent-side](#) checkout is only supported on Windows agent machine (for Linux and Mac agent you can use the [server-side checkout](#)). It is also required to have [Team Explorer](#) installed on the agent machine.

TeamCity automatically creates a TFS workspace for each [checkout directory](#) used. The workspace is created on behalf of the user account specified in the VCS root settings.

The created TFS workspaces are automatically removed after a two-week idle time.

TFS does not allow several workspaces on a machine mapped to the same directory. TeamCity TFS agent-side checkout may attempt to remove intersecting workspaces to create a new workspace that matches the specified VCS root and checkout rules. Note that it is unable to remove workspaces created by another user.

Option	Description
Enforce overwrite all files	When the option is enabled, TeamCity will call TFS to update workspace rewriting all files.

 Normally, there is no need to do forced update for every build. But, if you suspect that TeamCity is not getting the latest version from the repository, you can use this option.

Visual Studio Online

When configuring a VCS Root for a project hosted on Visual Studio Online, you have to enable alternate credentials support in your Visual Studio Online account.

Enabling Alternate Credentials

Applications that work outside the browser, such as the git client or the command-line TFS client, require basic authentication credentials to work. You can edit the VSO profile to set a secondary username that can be used in this case.

After navigating to your VSO account portal, use the menu at the top-right to edit your profile. The **Credentials** tab allows enabling alternate credentials. After clicking **Enable alternate credentials**, set a secondary username and password to use when configuring a VCS Root.

Connecting to Visual Studio Online TFVC

Configure the TFS Settings of the TeamCity VCS root:

Option	Description
URL	Enter the URL to your Visual Studio Online project's TFS path of the following format https://accountname.visualstudio.com/DefaultCollection/
Root	Define project root as \$/ProjectName
Username and Password	Enter your Microsoft Account credentials or use the alternate credentials created earlier, where the username must be prefixed with ##LIVE## .



TeamCity can also [automatically configure the project/VCS root](#) from a repository URL.

For step-by-step instructions, see the [related blog-post](#).

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](#)



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 <code>srcsafe.ini</code>	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 <code>vss-server</code> is a server name and <code>share</code> 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 check out a part of the VCS root configured and to map directories from the version control to subdirectories in the build checkout directory on a Build Agent. Thus, you can define a VCS root for the entire repository and make each build configuration checkout only the relevant part of it.

The Checkout Rules affect the changes displayed in the TeamCity for the build and the files checked out for the build on agent. To display changes, but not to trigger a build for a change, use [VCS Trigger Rules](#).

The general recommendation is to have a small number of VCS roots (pointing to the root of the repository) and define what is checked out by a specific build configuration via checkout rules.



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, go to the build configuration's **Version Control Settings** page, locate the VCS root in the list, and click the **Edit**

checkout rules link. A pop-up window will appear where you can enter the rule. Use the VCS repository browser to select a directory to check out.



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.

The general syntax of a single checkout rule is as follows:

```
+ | -: VCSPath [=> 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 `src/samples` path to the `samples` path in the Build Agent's default work directory. The second rule maps the contents of the `src` path to the `production/sources` on the build agent, except `src/help` which was excluded by the first rule and `src/samples` which was mapped to a different location by the third rule.

See also:

[Administrator's Guide: VCS Checkout Mode](#)

VCS Checkout Mode

The VCS Checkout mode is a configuration that sets how project sources reach an agent.

This mode affects only sources checkout. The current revision and changes data retrieving logic is executed by the TeamCity server and thus TeamCity server needs to access the VCS server in any mode.

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 the sources are exported rather than checked out, no administrative data is stored in the agent's file system and version control operations (like check-in, label or update) cannot be performed from the agent. TeamCity optimizes communications with the VCS servers by caching the sources and retrieving from the VCS server only the necessary changes (read more on that). Unless clean checkout is performed, the server sends to the agent incremental patches to update only the files changed since the last build on the agent in the given checkout directory.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p> Notes</p> <ul style="list-style-type: none"> The server side checkout simplifies administration overhead. Using this checkout mode, you need to install VCS client software on the server only (applicable to Perforce, Mercurial, TFS, Clearcase, VSS). Network access to VCS repository can also be opened to the server only. Thus, if you want to control who has access to the source repositories, the server side checkout is usually more effective. In some cases this checkout mode can lower the 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 the 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="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p> Notes</p> <ul style="list-style-type: none"> Agent checkout is usually more effective with regard to data transfers and VCS server communications. The agent side checkout 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" checkout rules are not supported for Git and Mercurial when using checkout on an agent due to these DVCS limitations. Integration with 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 build.vcs.number.* 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.

Build steps are configured on the **Build Steps** section of the [Build Configuration Settings](#) page: the steps can be auto-detected by TeamCity or added manually.

Each build step is represented by a [build runner](#) and provides integration with a specific build or test tool. You can add as many build steps to your build configuration as needed. For example, call a NAnt script before compiling VS solutions.

Build steps are invoked sequentially.

The decision whether to run the next build step may depend on the exit status of the previous build steps and the current build status.

The build step status is considered *failed* if the build process returned a non-zero exit code and the **Fail build if build process exit code is not zero** build failure condition is enabled (see [Build Failure Conditions](#)); otherwise build step is *successful*.

Note that the status of the build step and the build can be different. A build step can be successful, but the build can be failed because of another build failure condition, not based on the exit code (like failing a test or something else). On the other hand, if a build step has failed, the build will be failed too.

Execution policy

You can specify the step execution policy via the **Execute step** option:

- **Only if build status is successful** - before starting the step, the build agent requests the build status from the server, and skips the step if the status is failed.
- **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.
- **Even if some of previous steps failed**: select to make TeamCity execute this step regardless of the status of previous steps and status of the build.
- **Always, even if build stop command was issued**: select to ensure this step is always executed, even if the build was canceled by a user. For example, if you have two steps with this option configured, stopping the build during the first step execution will interrupt this step, while the second step will still run. Issuing the stop command for the second time will result in ignoring the execution policy: the build will be terminated.



- You can copy a build step from one build configuration to another from the original build configuration settings page.
- You can reorder build steps as needed. Note, that if you have a build configuration inherited from a template, you cannot reorder inherited build steps. However, you can insert custom build steps (not inherited) at any place and in any order, even before or between inherited build steps. Inherited build steps can be reordered in the original template only.
- You can disable a build step temporarily or permanently, even if it is inherited from a build configuration template.

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](#)
- [Ipr \(deprecated\)](#)
- [Maven](#)
- [MSBuild](#)
- [MSpec](#)
- [MSTest](#)
- [NAnt](#)
- [NuGet](#)
- [NUnit](#)
- [PowerShell](#)
- [Rake](#)
- [Simple Build Tool \(Scala\)](#)
- [Visual Studio \(sln\)](#)
- [Visual Studio 2003](#)
- [Visual Studio Tests](#)
- [Xcode Project](#)

See also:

Concepts: Build Runner

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> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">vs-addin\addin\addin.sln</div> <div style="background-color: #e0f2e0; padding: 10px; border: 1px solid #4CAF50; border-radius: 5px; 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>
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). <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> 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</div>
Run platform	From the drop-down list select the desired execution mode on a x64 machine.
Targets	A target is an arbitrary script for your project purposes. Enter targets separated by spaces. The available targets can be viewed in the Web UI by clicking the icon next to the field and added by checking the appropriate boxes.
Command line parameters	Specify any additional parameters for <code>msbuild.exe</code>

Reduce test failure feedback time	Use 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](#)

Ant

This is a runner for Ant build.xml files. TeamCity comes bundled with Ant 1.8.4., **since TeamCity 9.1** - with Ant 1.9.6.

In this section:

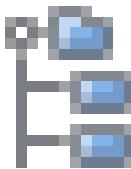
- [Support for Running Parallel Tests](#)
- [Ant Runner Settings](#)
 - [Ant Parameters](#)
 - [ant-net-tasks Tool](#)
 - [Java Parameters](#)
 - [Test parameters](#)
 - [Code Coverage](#)

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	<p>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.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;">   <p>Alternatively, click Choose file 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>
Build file content	<p>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.</p>

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. The available targets can be viewed in the Web UI by clicking the icon next to the field and added by checking the appropriate boxes. If this field is left empty, the default target specified in the build script file will be run.
Ant home path	Specify the path to the distributive of your custom Ant. You do not need to specify this parameter if you are going to use the Ant distributive bundled with TeamCity, Ant 1.8.4; since TeamCity 9.1- Ant 1.9.6. <div style="border: 1px solid #f0e68c; padding: 5px; margin-left: 20px;"> Please note, that you should use Ant 1.7 if you want to run JUnit4 tests in your builds.</div>
Additional Ant command line parameters	Optionally, specify additional command line parameters as a space-separated list. For example, you can specify the ant-net-tasks Tool here (see below).

ant-net-tasks Tool

The Ant build runner comes with a bundled tool, *ant-net-tasks*, which includes the jar files required for network tasks, such as FTP, sshexec, scp and mail.

It also contains [missing link Ant task](#) which can be used for REST requests.

To use the tool, specify `-lib "%teamcity.tool.ant-net-tasks%"` in [Additional Ant command line parameters](#) of the runner settings.

Java Parameters

Option	Description
JDK 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. <div style="border: 1px solid #f0e68c; padding: 5px; margin-left: 20px;"> 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: <code>%env.JDK_15%</code>.</div>
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 solid #f0e68c; padding: 10px; margin-left: 20px;">-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 fileset containing included tests only and processes it before all other filesets. It also patches other filesets to exclude tests added to the automatically generated fileset. After that JUnit starts and runs as usual.

Option	Description
Reduce test failure feedback time:	Use the following two options to instruct TeamCity to run some tests before others.
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>
	 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%%.

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 based on [ReSharper Command Line Tools](#) 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, 2012, 2013, and 2015 solutions.

 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. <pre>NLog.Logger.GetInstance().Log("abcd"); A.Log.Logger.GetInstance().Log("abcd");</pre>
Discard literals	If this option is checked, similar lines of code with different literals will be recognized as duplicates. <pre>myStatusBar.SetText("Not Logged In"); myStatusBar.SetText("Logging In...");</pre>
Discard local variables	If this option is checked, similar code fragments with different local variable names will be recognized as duplicates. <pre>int a = 5; a += 6; int b = 5; b += 6;</pre>
Discard class fields name	If this option is checked, the similar code fragments with different field names will be recognized as duplicates. <pre>Console.WriteLine(myFoo); Console.WriteLine(myBar); ... where myFoo and myBar are declared in the containing class</pre>

Discard types	If this option is checked, similar content with different type names will be recognized as duplicates. These include all possible type references (as shown below): <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 inspections on your code, TeamCity requires either an IntelliJ IDEA project file\directory, Maven pom.xml or Gradle build.gradle to be specified here.

Path to the project	<p>Depending on the type of project selected in the Project file type, specify here:</p> <ul style="list-style-type: none"> • 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. • For Gradle project: the path to the .gradle file. <p>This information is required by this build runner to understand the structure of the project.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">  The specified path should be relative to the checkout directory. </div>
Detect global libraries and module-based JDK in the *.iml files	<p><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.</p> <div style="border: 2px solid red; padding: 5px; 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><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.</p> <div style="border: 1px solid orange; padding: 5px; margin-top: 10px;">  If you update your project settings in IntelliJ IDEA (e.g add new jdk, libraries), remember to update the build runner settings by clicking Check/Reparse Project. </div>
Working directory	<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 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	<p>Use this field to specify the JDK home for the project.</p> <div style="border: 1px solid orange; padding: 5px; 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</pre> <pre>../Classes/*.jar</pre>
IDEA Home	If your project uses the IDEA JDK, specify the location of the IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

 You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a requirement for the corresponding property.

Java Parameters

Option	Description
JDK 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	If this option is checked, the test sources will be included in the duplicates analysis.  Tests may contain the data which is duplicated intentionally, and verifying tests for duplicates may yield a lot of results creating long builds and "spamming" your reports. We recommend you not select this option.

Include / exclude patterns| Optional, specify to restrict the sources scope to run duplicates analysis on. For details, refer to the section below [#IdeaPatterns]]

Detailization level	Use these options to define which elements of the source code should be distinguished when searching for repetitive code fragments. Code fragments can be considered duplicated if they are structurally similar, but contain different variables, fields, methods, types or literals. Refer to the samples below:
---------------------	--

Distinguish variables	If this option is checked, the similar contents with different variable names will be recognized as different. If this option is not checked, such contents will be recognized as duplicated: <pre>public static void main(String[] args) { int i = 0; int j = 0; if (i == j) { System.out.println("sum of " + i + " and " + j + " = " + i + j); } long k = 0; long n = 0; if (k == n) { System.out.println("sum of " + k + " and " + n + " = " + k + n); } }</pre>
Distinguish fields	If this option is checked, the similar contents with different field names will be recognized as different. If this option is not checked, such contents will be recognized as duplicated: <pre>myTable.addSelectionListener(new SelectionListener() { public void widgetDefaultSelected(SelectionEvent e) { } /*....*/ }); myTree.addSelectionListener(new SelectionListener() { public void widgetDefaultSelected(SelectionEvent e) { } /*....*/ });</pre>

Distinguish methods

If this option is checked, the methods of similar structure will be recognized as different. If this option is not checked, such methods will be recognized as duplicated. In this case, they can be extracted and reused.
Initial version:

```
public void buildCanceled(Build build, SessionData data) {
    /* ... */
    for (IListener listener : getListeners()) {
        listener.buildCanceled(build, data);
    }
}

public void buildFinished(Build build, SessionData data) {
    /* ... */
    for (IListener listener : getListeners()) {
        listener.buildFinished(build, data);
    }
}
```

After analysing code for duplicates without distinguishing methods, the duplicated fragments can be extracted:

```
public void buildCanceled(final Build build, final SessionData data)
{
    enumerateListeners(new Processor() {
        public void process(final IListener listener) {
            listener.buildCanceled(build, data);
        }
    });
}

public void buildFinished(final Build build, final SessionData
data) {
    enumerateListeners(new Processor() {
        public void process(final IListener listener) {
            listener.buildFinished(build, data);
        }
    });
}

private void enumerateListeners(Processor processor) {/* ... */

for (IListener listener : getListeners()) {
    processor.process(listener);
}
}

private interface Processor {
    void process(IListener listener);
}
```

Distinguish types	If this option is checked, the similar code fragments with different type names will be recognized as different. If this option is not checked, such code fragments will be recognized as duplicates. <pre>new MyIDE().updateStatus() new TheirIDE().updateStatus()</pre>
Distinguish literals	If this option is checked, similar line of code with different literals will be considered different. If this option is not checked, such lines will be recognized as duplicates. <pre>myWatchedLabel.setToolTipText("Not Logged In");</pre> <pre>myWatchedLabel.setToolTipText("Logging In...");</pre>
Ignore duplicates with complexity lower than	Complexity of the source code is defined by the amount of statements, expressions, declarations and method calls. Complexity of each of them is defined by its cost. Summarized costs of all these elements of the source code fragment yields the total complexity. Use this field to specify the lowest level of complexity of the source code to be taken into consideration when detecting duplicates. For meaningful results start with value 10.
Ignore duplicate subexpressions with complexity lower than	Use this field to specify the lowest level of complexity of subexpressions to be taken into consideration when detecting duplicates.
Check if Subexpression Can be Extracted	If this option is checked, the duplicated subexpressions can be extracted.

 Include / exclude patterns are newline-delimited set of rules of the form:

```
[ +: | -:]pattern
```

Where the pattern must satisfy these rules:

- must end with either ** or * (this effectively limits the patterns to only the directories level, they do not support file-level patterns)
- references to modules can be included as [module_name] / <path_within_module>

Some notes on patterns processing:

- excludes have precedence over includes
- if include patterns are specified, only directories matching these patterns will be included, all other directories will be excluded
- "include" pattern has a special behavior (due to underlying limitations): it includes the directory specified and all the files residing directly in the directories above the one specified.

Example:

```
+: testData/tables/**
-: testData/*
-: testdata/*
-: [testData]/**
```

 For the file paths to be reported correctly, "References to resources outside project/module file directory" option for the project and all

modules should be set to "Relative" in IDEA project.

FxCop

The [FxCop Build Runner](#) is intended for inspecting .NET assemblies and reporting possible design, localization, performance, and security improvements.

If you want TeamCity to display FxCop reports, you can either configure the corresponding build runner, or import XML reports by means of service messages if you prefer to run the FxCop tool directly from the script.

 The FxCop build runner requires FxCop installed on the build agent.

On this page:

- The description of the [FxCop build runner settings](#)
- Details on using [dedicated service messages](#).

For the list of supported FxCop versions, see [Supported Platforms and Environments](#).

FxCop Build Runner Settings

FxCop Installation

Option	Description
FxCop detection mode	When a build agent is started, it detects automatically whether FxCop is installed. If FxCop is detected, TeamCity defines the <code>%system.FxCopRoot%</code> agent system property. You can also use a custom installation of FxCop or the use FxCop checked in your version control. Depending on the selection, the settings displayed below will vary.
Autodetect installation	Select to use the FxCop installation on an agent.
FxCop version	<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	The option is available when <code><Custom></code> is selected above. Use this field to specify the path to your custom JDK used to run the build. If the field is left blank, the path to JDK Home is read either from the <code>JAVA_HOME</code> environment variable on agent the computer, or from <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: <div style="border: 1px solid #ccc; padding: 5px; width: fit-content;">-Xmx512m -Xms256m</div>

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}'"
    }
}
```

Since **TeamCity 9.1.2** these properties can also be referenced as `System.properties["<property_name>"]`.

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](#)
- [Getting the same results in IntelliJ IDEA and TeamCity Inspections Build](#)

IntelliJ IDEA Project Settings

Option	Description
Project file type	To be able to run IntelliJ IDEA inspections on your code, TeamCity requires either an IntelliJ IDEA project file\directory, Maven pom.xml or Gradle build.gradle to be specified here.
Path to the project	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.• For Gradle project: the path to the .gradle file. This information is required by this build runner to understand the structure of the project. <div style="border: 1px solid #ccc; padding: 5px; 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 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.



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.

Check/Reparse Project	<p>This option is available if you use an IntelliJ IDEA project to run the inspections. Click this button to reparse your IntelliJ IDEA project and import the build settings right from the project, for example the list of JDks.</p>
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 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> <p>! When building with the Ipr runner, this JDK will be used to compile the sources of corresponding IDEA modules. For Inspections and Duplicate Finder builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process, the JDK specified in the <code>JAVA_HOME</code> environment variable will be used.</p>
JDK Jar File Patterns	<p>Click this link to open a text area where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns.</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 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> <pre>-Xmx512m -Xms256m</pre> <p>i To modify the size of memory allocated to your Maven project in IDEA (set to <code>-Xmx512m</code> by default), use the <code>idea.maven.embedder.xmx jvm</code> property.</p> <p>Example:</p> <pre>-Didea.maven.embedder.xmx=512m</pre>

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.

i Include / exclude patterns are newline-delimited set of rules of the form:

```
[ +: | -: ]pattern
```

where the pattern must satisfy the following rules:

- must end with either `**` or `*` (this effectively limits the patterns to only the directories level, they do not support file-level patterns);

- references to modules can be included as [`<IDEA_module_name>`]/`<path_within_module>`. If you have a Maven project configured, you can use Maven module's artifactId as `<IDEA_module_name>`;
- the configured paths are treated as relative paths within content roots of the IDEA project modules. That is, the paths should be relative to the module's roots.

Some notes on patterns processing:

- excludes have precedence over includes
- if include patterns are specified, only directories matching these patterns will be included, all other directories will be excluded
- the include pattern has a special behavior (due to underlying limitations): it includes the directory specified and all the files residing directly in the directories above the one specified.

Example:

```
+:testData/tables/**
-:testData/*
-:testdata/*
-:[ testData]/**
```

 For the file paths to be reported correctly, the "References to resources outside project/module file directory" option for the project and all modules should be set to "Relative" in IDEA project.

For Maven inspections run, to ensure correct Java is used for the project JDK, define `env.JAVA_HOME` configuration parameter pointing to the JDK to be used as the project JDK.

Getting the same results in IntelliJ IDEA and TeamCity Inspections Build

The code inspections reported by IntelliJ IDEA and TeamCity Java Code Inspections build depend on a number of factors. You would need to ensure equal settings in IntelliJ IDEA and the build to get the same reports.

The relevant settings include:

- [inspections profile](#) used in IntelliJ IDEA and TeamCity build;
- environment-specific project dependencies (files not in version control, etc.);
- IDE-level settings, like defined SDKs, path variables, etc.;
- generated files: should be present in the TeamCity agent if they are present when working with the project in IntelliJ IDEA;
- IntelliJ IDEA version. It is recommended to use the same IntelliJ IDEA version that is used in the TeamCity build. TeamCity bundled an installation of IntelliJ IDEA. The version is written in the Inspections build log;
- the set and versions of the IntelliJ IDEA plugins that the project relies on.

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> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Specified path should be relative to the checkout directory. </div>
Detect global libraries and module-based JDK in the *.iml files	<p>If this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDKs when saved. This helps you ensure all references will be properly resolved.</p> <div style="border: 2px solid red; padding: 10px; margin-top: 10px;"> Warning When this option is selected, the process of opening and saving the build runner settings may become time-consuming, because it involves loading and parsing all project module files. </div>
Check/Reparse Project	<p>Click to reparse the project and import build settings right from the IDEA project, for example the list of JDKs.</p> <div style="border: 1px solid orange; padding: 10px; margin-top: 10px;"> 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	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	<p>Use this field to specify JDK home for the project.</p> <div style="border: 1px solid #ccc; padding: 5px; 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. The default value is used for Linux and Windows operating systems:</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>jre/lib/*.jar</pre> </div> <p>For Mac OS X, use the following lines:</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>lib/*.jar ../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	<p>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.</p>
JVM command line parameters	<p>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.</p> <p>Example:</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>-Xmx512m -Xms256m</pre> </div>

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:

[Administrator's Guide: IntelliJ IDEA Code Coverage](#)

Maven

Note that you can create a new Maven-based build configuration [automatically from URL](#), and set up a [dependency build trigger](#), if a specific Maven artifact has changed.



Remote Run Limitations related to Maven runner

As a rule, a personal build in TeamCity doesn't affect any "regular" builds run on the TeamCity server, and its results are visible to its initiator only. However, in case of using Maven runner, this behavior may differ.

TeamCity doesn't interfere anyhow with the Maven dependencies model. Hence, if your Maven configuration deploys artifacts to a remote repository, **they will be deployed there even if you run a personal build**. Thereby, a personal build may affect builds that depend on your configuration.

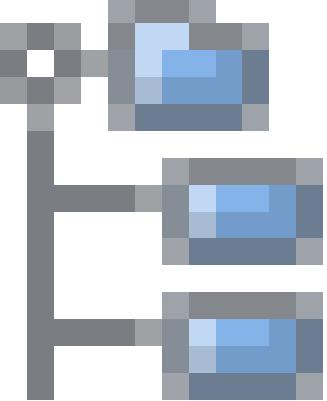
For example, you have a configuration A that deploys artifacts to a remote repository, and these artifacts are used by configuration B. When a personal build for A has finished, your personal artifacts will appear in B. This can be especially injurious, if configuration A is to produce release-version artifacts, because proper artifacts will be replaced with developer's ones, which will be hard to investigate because of Maven versioning model. Plus these artifacts will become available to all dependent builds, not only to those managed by TeamCity.

To avoid this, we recommend not using remote run for build configurations which perform deployment of artifacts.

On this page:

- [Maven runner settings](#)
 - [Maven Home](#)
 - [User Settings](#)
 - [Java Parameters](#)
 - [Local Artifact Repository Settings](#)
 - [Incremental Building](#)
 - [Code Coverage](#)
- [Using Maven Release with Perforce](#)

Maven runner settings

Option	Description
Goals	<p>In the Goals field, specify the sequence of space-separated Maven goals that you want TeamCity to execute. Some Maven goals can use version control systems, and, thus, they may become incompatible with some <i>VCS checkout modes</i>. 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. <p>i To use the <code>release:prepare</code> goal with Perforce VCS, see the section below.</p>
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="text-align: center;">  </div> <p>Alternatively, click here to choose the file using the VCS repository browser (Currently the VCS repository browser is only available for Git, Mercurial, Subversion and Perforce).</p>
Additional Maven command line parameters	<p>Specify the list of command line parameters.</p> <div style="border: 1px solid #f0e68c; padding: 5px; margin-top: 5px;"> ⚠ The following parameters are ignored: <code>-q</code>, <code>-f</code>, <code>-s</code> (if User settings path is provided) </div>
Working directory	<p>Specify the Build Working Directory, if it differs from the build checkout directory.</p>

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	<p>If there are settings files uploaded to the TeamCity server via the administration UI, you can select one of the available options here. To upload settings file to TeamCity, click <i>Manage settings files</i>.</p> <p>Maven settings are defined on the project level. You can see the settings files defined in the current project or upload files on the Project Settings page using Maven Settings. The files will be available in the project and its subprojects. The uploaded files are stored in the <TeamCity Data Directory>/config/projects/%projectId%/pluginData/mavenSettings directory. If necessary, they can be edited right there. The uploaded files are used both for the agent and server-side Maven functionality.</p> <p>If Custom or Predefined settings are used, the path to the effective user settings file is available inside the maven process as the <code>teamcity.maven.userSettings.path</code> system property.</p>

Java Parameters

JDK Home Path	The path to JDK Home is read from the *JAVA_HOME* environment variable or *JDK home* 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	<p>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:</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> -Xmx512m -Xms256m </div>

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.

Using Maven Release with Perforce

To run the `release:prepare` maven task with Perforce VCS, do the following:

Check the following:

1. Make sure you're using at least 2.0 version of [Maven Release Plugin](#).
2. Use [ticket-based authentication](#) for Maven Release plugin.
3. Make sure that your `release:prepare` maven task works when it is run from the command line without TeamCity.

In the [Perforce VCS Root Settings](#) of your build configuration in TeamCity:

1. Enable the [checkout on agent](#).
2. Enable [Use ticket-based authentication](#) in Perforce VCS root settings.
3. Make sure your build agent environment doesn't have any occasional P4 variables which can interfere with the execution of Maven Release Plugin.
4. Specify `release:prepare` in the **Goals** field of the Maven build step and run the build.

See also:

Concepts: Build Runner

Administrator's Guide: [Maven Artifact Dependency Trigger](#) | [Creating Maven Build Configuration](#)

MSTest

This page describes MSTest runner options, for details on MSTest support, please refer to the [corresponding page](#).



Since TeamCity 9.1 the MSTest runner is merged into the [Visual Studio Tests runner](#).

After upgrade to TeamCity 9.1, MSTest build steps are automatically configured by the new runner. VSTest steps have to be configured manually.

Option	Description
Path to MSTest.exe	A path to <code>MSTest.exe</code> file. By default Build Agent will autodetect the MSTest installation. Prior to TeamCity 9.1 the MSTest location was reported as system properties: <code>%system.MSTest.8.0%</code> , <code>%system.MSTest.9.0%</code> , <code>%system.MSTest.10.0%</code> , <code>%system.MSTest.11.0%</code> , <code>%system.MSTest.12.0%</code> , <code>%system.MSTest.14.0%</code> that referred to MSTest 2008, 2010, 2012, 2013, 2015 correspondingly. Since TeamCity 9.1 system parameters of the <code>%system.MSTest.xx.yy%</code> format were changed to configuration parameters of the <code>%teamcity.dotnet.mstest.xx.yy%</code> format. If system properties are required for the build, the mstest-legacy-provider plugin can be used.
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:</code> arguments 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 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:

```

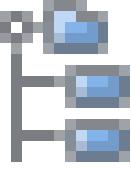
<target name="build">
    <nunit2 verbose="true" haltonfailure="false" failonerror="true"
failonfailureatend="true">
        <formatter type="Plain" />
        <test haltonfailure="false">
            <assemblies>
                <include name="dll1.dll" />
                <include name="dll2.dll" />
            </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> <div style="display: flex; align-items: center;"> ✓  </div> <p>Alternatively, click here 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. The available targets can be viewed in the Web UI by clicking the icon next to the field and added by checking the appropriate boxes.
Working directory	Specify the path to the Build Working Directory . By default, the build working directory is set to the same path as the build checkout directory .
NAnt home	Enter a path to the NAnt.exe .

Target framework	Sets <code>-targetframework</code> : option to 'NAnt' and generates appropriate agent requirements (<i>mono-2.0</i> target framework will require <i>Mono</i> system property, <i>net-2.0</i> — <i>DotNetFramework2.0</i> property, and so on). Selecting unsupported in TeamCity framework (<i>sscli-1.0</i> , <i>netcf-1.0</i> , <i>netcf-2.0</i>) won't impose any agent requirements.
	<p> This option has no effect on framework which used to run <code>NAnt.exe</code>. <code>NAnt.exe</code> will be launched as ordinary exe file if .NET framework was found, and through mono executable, if not.</p>
Command line parameters	<p>Specify any additional parameters for <code>NAnt.exe</code></p> <p> TeamCity passes automatically to NAnt all defined system properties, so you do not need to specify all of the properties here via '-D' 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](#)

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, 2.6.3**. Since Teamcity 9.1, NUnit 3.0 is also supported.

NUnit Test Settings

NUnit runner	Select the NUnit version to be used to run the tests. The number of settings available will vary depending on the selected version.
Path to nunit-console.exe	<i>Available if NUnit 3.0 is selected.</i> Specify the path to nunit-console.exe
Additional command line parameters	<i>Available if NUnit 3.0 is selected.</i> Enter additional command line parameters to pass to nunit-console.exe
.NET Runtime	<p>From the Platform drop-down, select the desired execution mode on a x64 machine. Supported values are: <code>Auto (MSIL)</code>, <code>x86</code> and <code>x64</code>.</p> <p>From the Version drop-down, select the desired .NET Framework version. Supported values are: <code>v2.0</code>, <code>v4.0</code>; and <code><auto></code>, <i>available if NUnit 3.0 is selected</i>.</p> <p> For NUnit 3.0, if <code><auto></code> is selected, tests will run under the framework they are compiled with.</p>

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> <pre>***.dll</pre> <p> All these wildcards are specified relative to path that contains the solution file.</p>
Do not run tests from	<p>Specify .NET assemblies that should be excluded from the list of assemblies to test. Multiple entries are comma-separated; usage of MSBuild wildcards is enabled.</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>
NUnit categories include	<p>Specify NUnit categories of tests to be run. Multiple entries are comma-separated.</p> <p>Category expressions are supported here as well; commas, semicolons, and new-lines are treated as global or operations (prior to the expression parsing).</p>
NUnit categories exclude	<p>Specify NUnit categories to be excluded from the tests to be run. Multiple entries are comma-separated.</p> <p>Category expressions are supported here as well; commas, semicolons, and new-lines are treated as global or operations (prior to the expression parsing).</p>
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. Currently not supported for NUnit 3.0.

Code Coverage

To learn about configuring code coverage options, please refer to the [Configuring .NET Code Coverage](#) page.

 **For NUnit 3.0**, only JetBrains dotCover is supported as a coverage tool.

See also:

[Administrator's Guide: Configuring Unit Testing and Code Coverage | NUnit Support](#)

Simple Build Tool (Scala)

The Simple Build Tool (Scala) runner natively supports **SBT** builds: you can build your code, run tests and see the results in a handy way in TeamCity. The supported SBT version 0.13.+.

The runner, formerly provided as a [standalone plugin](#), is bundled **since TeamCity 9.1**.

SBT runner settings

SBT parameters

Option	Description
SBT commands	Commands to execute, e.g. <code>clean "set scalaVersion:="2.11.6"" compile test or ;clean;set scalaVersion:="2.11.6";compile;test</code> .

SBT installation mode	When the default <Auto> option is selected, the latest SBT version will be installed on every TeamCity agent your build will be running. To specify an existing installation, use the <Custom> mode. The <code>sbt-launch.jar</code> from the <code>\bin</code> directory of the SBT home will be launched.
SBT home path	<i>Available if <Custom> is selected in the option above.</i> The path to the existing SBT installation directory.
Working directory	Optional. Specify the build working directory if it differs from the build checkout directory .

Java Parameters

Option	Description
JDK	When <Default> is selected, the JDK specified in the <code>JAVA_HOME</code> environment variable on the agent or the agent's own Java is used to run the build process. Set to <Custom> to use a custom JDK.
JDK home path	<i>Available if <Custom> is selected in the option above.</i> Specify the path to your custom JDK which will be used to run the build.
JVM command line parameters	Specify the desired Java Virtual Machine parameters, such as maximum heap size or parameters that enable remote debugging. These settings are passed to the JVM used to run your build. Example: <pre>-Xmx512m -Xms256m</pre>

Rake



TeamCity Rake runner supports the [Test::Unit](#), [Test-Spec](#), [Shoulda](#), [RSpec](#), [Cucumber](#) test frameworks. It is compatible with Ruby interpreters installed using Ruby Version Manager (MRI Ruby, JRuby, IronRuby, REE, MacRuby, etc.) with rake 0.7.3 gem or higher.

In this section:

- [Prerequisites](#)
- [Important Notes](#)
- [Rake Runner Settings](#)
 - [Rake Parameters](#)
 - [Ruby Interpreter](#)
 - [Launching Parameters](#)
 - [Tests Reporting](#)
- [Known Issues](#)
- [Additional Runner Options](#)
- [Development Links](#)

Prerequisites

Make sure to have Ruby interpreter (MRI Ruby, JRuby, IronRuby, REE, MacRuby, or etc) with rake 0.7.3 gem or higher (mandatory) and all necessary gems for your Ruby (or ROR) projects and testing frameworks installed on at least one build agent.

You can install several Ruby interpreters in different folders. On Linux/MacOS it is easier to configure using [RVM](#) or [rbenv](#). It is possible to install Ruby interpreter and necessary Ruby gems using the [Command Line](#) build runner step. If you want to automatically configure agent requirements for this interpreters, you need to register its paths in the build agent configuration properties and then refer to such property name in the [Rake build runner configuration](#).

To install a gem, execute:

```
gem install <gem's name>
```

You can refer to the [Ruby Gems Manuals](#) for more information.

Instead of the `gem` command, you can install gems using the [Bundler](#) gem.



If you use Ruby 1.9 for Shoulda, Test-Spec and Test::Unit frameworks to operate, the 'test-unit' gem must be installed.

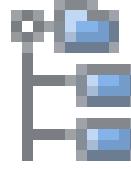
 To use the `minitest` framework, '`minitest-reporters`' gem must be installed. See details in the [RubyMine webhelp](#).

Important Notes

- Ruby's *pending specs* are shown as **Ignored Tests** in the **Overview** tab.
- Rake Runner uses its own unit tests runner and loads it using the `RUBYLIB` environment variable. You need to ensure your program doesn't clear this environment variable, but you may append your paths to it.
- If you run RSpec with the '--color' option enabled under Windows OS, RSpec will suggest you install the `win32console` gem. This warning will appear in your build log, but you can ignore it. TeamCity Rake Runner doesn't support coloured output in the build log and doesn't use this feature.
- Rake Runner runs spec examples with a custom formatter. If you use additional console formatter, your build log will contain redundant information.
- `Spec::Rake::SpecTask.spec_opts` of your `rakefile` is affected by `SPEC_OPTS` command line parameter. Rake Runner always uses `SPEC_OPTS` to 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	<p>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.</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> 
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: <pre>ruby rake <Additional Rake command line parameters> <TeamCity Rake Runner options, e.g TESTOPTS> <tasks></pre>

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 <code>PATH</code> .

Ruby interpreter path	The path to Ruby interpreter. The path cannot be empty. This field supports values of environment and system variables. For example: %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](#).

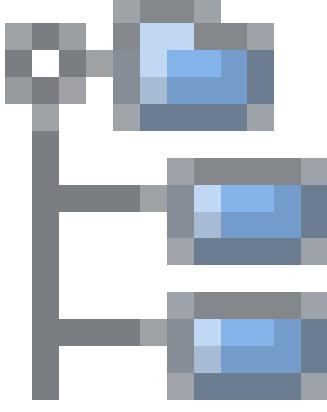
Visual Studio (sln)

This page contains reference information for the Visual Studio(sln) Build Runner that builds Microsoft Visual Studio 2005, 2008, 2010, 2012, 2013,

and **since Teamcity 9.1**. Visual Studio 2015 solution files.
To build Microsoft Visual Studio 2003 solution files, use the [Visual Studio 2003 runner](#).

 The Visual Studio (.sln) build runner requires the proper version of Microsoft Visual Studio installed on the build agent.

General Build Runner Options

Option	Description
Solution file path	<p>Specify the path to the solution to be built relative to the Build Checkout Directory. For example:</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">vs-addin\addin\addin.sln</div> <div style="background-color: #e0f2e0; padding: 10px; margin-top: 10px; border: 1px solid #ccc;"> </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, 2012, 2013, 2015 .
Targets	Specify the Microsoft Visual Studio targets specific for the previously selected Visual Studio version. The possible options are B uild, R ebuild, C lean, P ublish 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 Parameters page .

See also:

Troubleshooting: Visual Studio logging

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	A path to the solution to be built is relative to the build checkout directory . For example: <code>vs-addin\addin\addin.sln</code> 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).
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	Click this link and specify the desired map in the text area. Use the following format: <code>http://localhost:8111=myProjectPath/myProject</code> where <ul style="list-style-type: none">http://localhost:8111 is a host where the project will be uploadedmyProjectPath/myProject is the project root
Resolve URLs via WebDAV	Click this radio button, if you want the URLs to be resolved via WebDav. Make sure that all the necessary files are properly updated. The build agent may not update information from VCS implicitly.

MS Visual Studio reference path	Check this option, if you want to automatically include reference path of MS Visual Studio to the build.
NAnt home	Specify path to the NAnt executable to run builds of the build configuration. The path can be absolute, relative to the build checkout directory; also you can use an environment variable.
Command line parameters	<p>Specify any additional parameters for <code>NAnt.exe</code></p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> ✔ TeamCity passes automatically to NAnt all defined system properties, so you do not need to specify all of the properties here via '-D' option. You can create necessary properties at the Build Parameters section of the build configuration settings. </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 solid #ccc; padding: 10px; margin-top: 10px;"> <pre>***.dll</pre> </div> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> ⓘ All these wildcards are specified relative to path that contains the solution file. </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> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <pre>**\obj***.dll</pre> </div> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> ⓘ All these wildcards are specified relative to path that contains the solution file. </div>
Reduce test failure feedback time	Use following option to instruct TeamCity to run some tests before others.
Run recently failed tests first	If checked, in the first place TeamCity will run tests failed in previous finished or running builds as well as tests having high failure rate (a so called <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="border: 1px solid #ccc; padding: 5px; margin-top: 10px;">  Specified path should be relative to the checkout directory. </div>
Detect global libraries and module-based JDK in the *.iml files	<p>If this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDKs when saved. This helps you ensure all references will be properly resolved.</p> <div style="border: 2px solid red; padding: 10px; margin-top: 10px;">  Warning When this option is selected, the process of opening and saving the build runner settings may become time-consuming, because it involves loading and parsing all project module files. </div>
Check/Reparse Project	<p>Click to reparse the project and import build settings right from the IDEA project, for example the list of JDKs.</p> <div style="border: 1px solid #ccc; 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>	This field appears, if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In the Set value to field , specify a path to project resources, to be used on different build agents.

Project JDKs

This section provides the list of JDKs detected in the project.

Option	Description
JDK Home	<p>Use this field to specify JDK home for the project.</p> <p>! When building with the Ipr runner, this JDK will be used to compile the sources of corresponding IDEA modules. For Inspections and Duplicate Finder builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process itself the JDK specified in the <code>JAVA_HOME</code> environment variable will be used.</p>
JDK Jar File Patterns	<p>Click this link to open a text area, where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns. The default value is used for Linux and Windows operating systems:</p> <pre>jre/lib/*.jar</pre>
	<p>For Mac OS X, use the following lines:</p> <pre>lib/*.jar ../Classes/*.jar</pre>
IDEA Home	If your project uses the IDEA JDK, specify the location of IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

! You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a requirement for the corresponding property.

Java Parameters

Option	Description
JDK home path	<p>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.</p>
JVM command line parameters	<p>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.</p> <p>Example:</p> <pre>-Xmx512m -Xms256m</pre>

Additional Pre/Post Processing (Ant)

Option	Description
Run before build	In the appropriate fields, enter the Ant scripts and targets (optional) that you want to run prior to starting the build. The path to the Ant file should be relative to the project root directory.
Run after build	In the appropriate fields, enter the Ant scripts and targets (optional) that you want to run after the build is completed. The path to the Ant file should be relative to the project root directory.

JUnit Test Runner Settings



JUnit test settings map to the attributes of JUnit task. For details, refer to <http://ant.apache.org/manual/OptionalTasks/junit.html>

Option	Description
Test patterns	<p>Click the Type test patterns link, and specify the required test patterns in a text area. These patterns are used to generate parameters of the <code>batchtest</code> JUnit task section. Each pattern generates either <code>include</code> or <code>exclude</code> section. These patterns are also used to compose classpath for the test run. Each module mentioned in the patterns adds its classpath to the whole classpath.</p> <p>Each pattern should be placed on a separate line and has the following format:</p> <pre>[-]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 <code>testFileNamePattern</code> 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>*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	By default the whole classpath is composed of all classpaths of the modules used to get tests from. The following two options define whether you will use the default classpath, or take it from the specified module.
Override classpath in tests	If this option is checked, you can define test classpath from a single, explicitly specified module.
Module name to use JDK and classpath from	If the option Override classpath in tests is checked, you have to specify the module, where the classpath to be used for tests is specified.
JUnit Fork mode	Select the desired fork mode from the combo box: <ul style="list-style-type: none"> • 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	Check this option, if you want a new classloader to be instantiated for each test case. This option is available only if Do not fork option is selected.
Include Ant runtime	Check this option to add Ant classes, required to run JUnit tests. This option is available if fork mode is enabled (Fork per test or Fork once).
JVM executable	Specify the command that will be used to invoke JVM. This option is available if fork mode is enabled (Fork per test or Fork once).
Stop build on error	Check this option, if you want the build to stop if an error occurs during test run.
JVM command line parameters for JUnit	Specify JVM parameters to be passed to JUnit task.
Tests working directory	Specify the path to the working directory for tests.
Tests timeout	Specify the lapse of time in milliseconds, after which test will be canceled. This value is ignored, if Do not fork option is selected.
Reduce test failure feedback time	<p>Use following two options to instruct TeamCity to run some tests before others.</p> <div style="border: 1px solid #ccc; padding: 5px;"> <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> </div>
Run recently failed tests first	If checked, in the first place TeamCity will run tests failed in previous finished or running builds as well as tests having high failure rate (a so called <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.
Verbose Ant	<p>Check this option, if the generated JUnit task has to produce verbose output in ant terms.</p> <div style="border: 1px solid #ccc; padding: 5px;"> <p> 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.</p> </div>

Code Coverage

To learn about configuring code coverage options, please refer to the [Configuring Java Code Coverage](#) page.

MSpec

The *MSpec Test Runner* is designed specifically to run [MSpec](#) tests.

 To run tests using MSpec, you need to install it on at least one build agent.

MSpec Settings

Option	Description
Path to MSpec.exe	A path to <code>mspec.exe</code> file.

.NET Runtime	From the Platform drop-down, select the desired execution mode on a x64 machine. Supported values are: Auto (MSIL) (default), x86 and x64. From the Version drop-down, select the desired .NET Framework version.
	 If you have MSpec as an MSIL .NET 2.0/3.5 executable, TeamCity can enforce it to execute under any required runtime: x86 or x64, and .NET2.0 or .NET 4.0 runtime.
Run tests from	Specify the .NET assemblies where the MSpec tests to be run are stored.
Do not run tests from	Specify the .NET assemblies that should excluded from the list of found assemblies to test.
Include specifications	Specify comma- or newline separated list of specifications to be executed.
Exclude specifications	Specify comma- or new line separated list of specifications to be excluded.
Additional commandline parameters	Enter additional commandline parameters for <code>mspec.exe</code> .

Code Coverage

Learn about configuring code coverage options.

See also:

[Administrator's Guide: Configuring .NET Code Coverage](#)

.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](#)

PowerShell

The [PowerShell build runner](#) is specifically designed to run PowerShell scripts.

PowerShell Settings

Option	Description
Version	List of PowerShell versions supported by TeamCity. It is passed to <code>powershell.exe</code> as the <code>-Version</code> command line argument.
PowerShell run mode	Select the desired execution mode on a x64 machine.
Error Output	Specify how the error output is handled by the runner: <ul style="list-style-type: none">• error: any output to <code>stderr</code> is handled as an error• warning: default; any output to <code>stderr</code> is handled as a warning <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"><p> To fail a build if "an error message is logged by build runner" (see Build Failure Conditions), change the default setting of the Error Output selector from warning to error.</p></div>
Working directory	Specify the path to the build working directory .
Script	Select whether you want to enter the script right in TeamCity, or specify a path to the script: <ul style="list-style-type: none">• File: Enter the path to a PowerShell file. The path has to be relative to the checkout directory.• Source: Enter the PowerShell script source. Note that TeamCity parameter references will be replaced in the code.
Script execution mode	Specify the PowerShell script execution mode. By default, PowerShell may not allow execution of arbitrary <code>.ps1</code> files. TeamCity will try to supply the <code>-ExecutionPolicy ByPass</code> argument. If you've selected Execute .ps1 script from external file , your script should be signed or you should make PowerShell allow execution of arbitrary <code>.ps1</code> files. If the execution policy doesn't allow running your scripts, select Put script into PowerShell stdin mode to avoid this issue. The <code>-Command</code> mode is deprecated and is not recommended for use with PowerShell of version greater than 1.0
Script arguments	<i>Available if "Script execution mode" option is set to "Execute .ps1 script from external file".</i> 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 the console with commands like `Write-Output`, `Write-Error` and similar (see [TW-15080](#)). To avoid this behavior, either use the `Write-Host` command, or adjust the buffer length manually:

```
function Set-PSConsole {  
    if (Test-Path env:TEAMCITY_VERSION) {  
        try {  
            $rawUI = (Get-Host).UI.RawUI  
            $m = $rawUI.MaxPhysicalWindowSize.Width  
            $rawUI.BufferSize = New-Object Management.Automation.Host.Size ([Math]::max($m,  
500), $rawUI.BufferSize.Height)  
            $rawUI.WindowSize = New-Object Management.Automation.Host.Size ($m,  
$rawUI.WindowSize.Height)  
        } catch {}  
    }  
}
```

Error Handling

Due to [this issue](#) in PowerShell itself which causes zero exit code to be always returned to a caller, TeamCity cannot always detect whether the script has executed correctly or not. We recommend several approaches that can help in detecting script execution failures:

- *Manually catching exceptions and explicitly returning exit code*

Since TeamCity 9.0 the PowerShell plugin does not use the cmd wrapper around `powershell.exe`. It makes returning the explicit exit code possible.

```
try {
    # your code here
} Catch {
    $ErrorMessage = $_.Exception.Message
    Write-Output $ErrorMessage
    exit(1)
}
```

- *Setting Error Output to Error and adding build failure condition*

In case syntax errors and exceptions are present, PowerShell writes them to `stderr`. To make TeamCity fail the build, set **Error Output** option to `Error` and add a **build failure condition** that will fail the build on any error output.

- *Failing build on certain message in build log* Add a **build failure condition** that will fail the build on a certain message (say "POWERSHELL ERROR") in the build log.

```
$ErrorMessage = "POWERSHELL ERROR"
try {
    # your code here
} Catch {
    Write-Output $ErrorMessage
    exit(1)
}
```

Development Links

The PowerShell support is implemented as an open-source plugin. For development links refer to the [plugin's page](#).

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 <code>.sln</code> 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.

NuGet

TeamCity integrates with [NuGet](#) package manager and provides:

- [NuGet feed](#) based on the builds' published artifacts
- A set of NuGet runners to be used in the build on Windows OS:
 - 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 Dependency Trigger](#) which allows triggering builds on NuGet feed updates



- NuGet build runners are only supported on build agents running Windows OS.
- To use packages from an authenticated feed, see the [NuGet Feed Credentials](#) build feature.

On this page:

- [Typical Usage Scenarios](#)
- [Installing NuGet to TeamCity agents](#)
- [Using TeamCity as NuGet Server](#)

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.

Installing NuGet to TeamCity agents

The NuGet trigger and the NuGet-related build runners require the NuGet command line binary configured on the server. They are automatically distributed to agents once configured.

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.

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 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 (available to server system administrators). Two different links will be displayed on the 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.

When the NuGet Server is enabled, all NuGet packages published as TeamCity [build artifacts](#) will be indexed and will appear in NuGet feed. The feed will include the packages from the build configurations where the currently authenticated user has permission to view build artifacts ("View project" permission).

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), only specify NuGet packages as TeamCity build artifacts.
- 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](#).

If the build artifacts are changed under TeamCity Data Directory manually, you need to instruct TeamCity to reindex NuGet feed. For that, click "reset" link for "buildsMetadata" under Administration > Diagnostic > Caches.



Internet Explorer settings may need to be set to trust the TeamCity Server when working in a mixed authentication environment. For a step-by-step example of NuGet setup see blog post: [Setting up TeamCity as a native NuGet Server](#).

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. If left blank, http://nuget.org is used to search for your packages. You can also specify your own NuGet repository. If you are using TeamCity as a NuGet repository, specify <code>%teamcity.nuget.feed.server%</code> here. If you use packages from an authenticated feed specify <code>%teamcity.nuget.feed.auth.server%</code> instead and use the NuGet Feed Credentials build feature.
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 the solution. The package versions and constraints are taken from <code>packages.config</code> files.
Update Mode	Select one of the following: <ul style="list-style-type: none">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 <code>packages.config</code> - Select to update packages via calls to <code>NuGet.exe update Packages.Config</code> for each <code>packages.config</code> file under the solution.
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

The **NuGet Pack** build runner allows building a NuGet package from a given specification file. If you want to publish this package, add a [NuGet Publish](#) build step.



NuGet Pack is only supported on build agents running Windows OS.

Configure the following options of the NuGet Pack runner:

Option	Description
NuGet.exe	Select a NuGet version to use from the drop-down list (if you have installed NuGet beforehand), or specify a custom path to <code>NuGet.exe</code> .

Specification files	Enter path(s) to <code>csproj</code> or <code>nuspec</code> file(s). You can specify as many specification files here as you need. Wildcards are supported. If you specify here a <code>csproj</code> file, you won't have to redefine the version number and copyright information in the spec file.
Prefer project files to <code>.nuspec</code>	Check the box to use the project file (if exists, i.e. <code>.csproj</code> or <code>.vbproj</code>) for every matched <code>.nuspec</code> file.
Version	Specify the package version. Overrides the version number from the <code>nuspec</code> file. You can use the TeamCity variable <code>%build.number%</code> here.
Base Directory	Select an option from the drop down list to specify the directory where the files defined in the <code>nuspec</code> file are located (the directory against which the paths in <code><files></files></code> from <code>nuspec</code> are resolved, usually some <code>bin</code> directory). If Use explicit directory is set and the field is left blank, TeamCity will use the build checkout directory as the base directory.
Output Directory	Specify the path where the generated NuGet package is to be put.
Clean output directory	Check the box to clean the directory before packing.
Publish created packages to build artifacts	Check the box if you're using TeamCity as a NuGet repository to publish packages to the TeamCity's NuGet server and be able to use them as regular TeamCity artifacts.
Exclude files	Specify one or more wildcard patterns to exclude when creating a package. Equivalent to the <code>NuGet.exe -Exclude</code> argument.
Properties	Semicolon or new-line separated list of package creation properties. For example, to make a release build, you define here <code>Configuration=Release</code> .
Options	Create tool package - check the box to place the output files of the project to the tool folder. Include sources and symbols - check the box to create a package containing sources and symbols. When specified with a <code>nuspec</code> , it creates a regular NuGet package file and the corresponding symbols package (needed for publishing the sources to <code>SymbolsSource</code>).
Command line parameters	Set additional command line parameters to be passed to <code>NuGet.exe</code> .

See also:

[Administrator's Guide: NuGet Installer | NuGet Publish](#)

NuGet Publish

The **NuGet Publish** build runner is intended to publish (`push`) your NuGet packages to a given feed (custom or default).

 If you're using TeamCity as a 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 the path to it in the [General Settings](#) of your build configuration.

 NuGet Publish is only supported on build agents running Windows OS.

This page describes the NuGet Publish runner options:

Option	Description
NuGet.exe	Select a NuGet version to use from the drop-down list (if you have installed NuGet beforehand), or specify a custom path to <code>NuGet.exe</code> .
Packages	Specify a newline-separated list of NuGet package files (<code>.nupkg</code>) to publish to the NuGet feed. List packages individually or use wildcards.
Only upload packages but do not publish them to feed	Select the checkbox if you want to upload your package to the TeamCity server but keep it invisible in the feed. Works for NuGet versions 1.5 and older.
API key	Provide your API key.

Package Source	Specify the destination NuGet packages feed URL to push packages to, by default it is <code>nuget.org</code> . If you have your own NuGet server, specify its address here.
----------------	---

See also:

[Administrator's Guide: NuGet Installer | NuGet Pack](#)

Xcode Project

The *Xcode Project Build Runner* supports Xcode 3 (target-based build), Xcode 4 (scheme-based build), Xcode 5, 6 and **since TeamCity 9.1**, Xcode 7.0 (see the related [section below](#)).

The runner provides structured build log based on Xcode build stages, detects compilation errors, reports tests from the `xcodebuild` utility, adds automatic agent requirements for the appropriate version of tools installed (Xcode, SDKs, etc.) and reporting tools via agent properties.



To run an Xcode build, you need to have one or more build agents running Mac OS X with installed Xcode.

Xcode 7 Support

Xcode 7 support has been tested with Xcode7 beta.

To make a TeamCity agent work with the beta version of Xcode:

- if Xcode 7 beta is the only installed Xcode on the agent machine, it will be used by default. Agent restart is required if Xcode was installed/updated.
- if Xcode 6.x and 7 beta are both installed, you need to either specify **the path to the beta version** in the **Path to Xcode** of the Xcode Project build step settings or **select the default XCode distribution** using the `xcode-select` tool (details on [Apple.com](#)).

The path to the release version of Xcode: `/Applications/Xcode.app/Contents/Developer`

The path to the beta version of Xcode: `/Applications/Xcode-beta.app/Contents/Developer`

Command to switch: `sudo xcode-select -s path_to_xcode_distribution`

Note that if you use `xcode-select`, the agent must be restarted, as it only detects Xcode distributions and reports them to the server during startup.

Xcode Project Runner Settings

Setting	Build	Description
Path to the project or workspace		The path to a (<code>.xcodeproj</code>) project file or a (<code>.xcworkspace</code>) workspace file, should be relative to the checkout directory. For Xcode 3 build, only the path to 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 (for project) or scheme-based (for project and workspace) build. Depending on the selection, the settings displayed will vary.
Scheme	Scheme-based	Xcode scheme to build. The list of available schemes is formed by parsing your project/workspace files in the VCS. Make sure your Path to the project or workspace is set correctly and click the Check/Reparse Project button to show/refresh the schemes list. Note that a scheme must be shared to be shown in the list (to check if your scheme is shared, verify that it is located under the <code>xcshareddata</code> folder and not under the <code>xcuserdata</code> one, and that the <code>xcshareddata</code> folder is committed to your VCS; to check the latter 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 are sure you want to change the number or order of actions. <div style="border: 1px solid #ccc; padding: 5px; background-color: #f9f9f9;"> If your project is built under Xcode 5, 6, 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).</div>
Run tests		Check this option if want to run tests after you project is built.
Additional command line parameters		Other command line parameters to be passed to the "xcodebuild" utility.

See also:

Concepts: Build Runner

Visual Studio Tests

The Visual Studio Tests runner, available **since TeamCity 9.1**, integrates MSTest runner and VSTest console runner formerly provided as an external plugin. The Visual Studio Tests runner provides support for both frameworks enabling TeamCity to execute tests and automatically import their test results.



After upgrade to TeamCity 9.1, MSTest build steps are automatically converted to the Visual Studio Tests runner steps.

The VSTest steps created using the [VSTest.Console Runner](#) plugin will not be converted and will have to be configured manually with the Visual Studio Tests runner. If you choose to continue using the [VSTest.Console Runner](#) plugin with TeamCity 9.1, no changes are required.

On this page:

- Visual Studio Tests runner settings
 - VSTest Settings
 - Custom test logger
 - MSTest settings

Visual Studio Tests runner settings

Option	Description
Test engine type	Select the tool used to run tests: MSTest or VSTest
Test engine version	Select the version of the tool from the drop-down. By default, the available VSTest and MSTest installations are autodetected by TeamCity. MSTest versions 2005 through 2015 are supported, VSTest 2012, 2013 and 2015 are supported. You can specify a custom path to the test runner here as well. TeamCity parameters are supported.
Test file names	Specify the new-line separated list of assemblies to run tests on in the included assemblies list. Exclude assemblies from test run by specifying them in the corresponding field. Wildcards are supported.

Run configuration file	(Optional) Specify the run configuration file to use. You can use the file browser
Additional command line parameters	<p>Enter additional command line parameters for the selected tool.</p> <p>Note that tests run with MSTest are not reported on-the-fly.</p> <p>The Microsoft Developer Network lists the available options for VSTest and MSTest</p>

The rest of settings will vary depending on the engine to run tests with:

- [VSTest Settings](#)
- [MSTest settings](#)

VSTest Settings

Option	Description
Target platform	Select the platform bitness. Note that specifying x64 target platform will force the <code>vstest.console</code> process to be run in the isolated mode
Framework	Select the .Net platform
Test names	Newline-separated list of test names
Test case filter	Run tests that match the given expression. See more in Microsoft documentation
Run in isolation	Run the tests in an isolated process
Use real-time test reporting	Use custom TeamCity test logger for real-time reporting. See the next section

Custom test logger

VSTest.Console supports custom loggers, i.e. libraries that can handle events that occur when tests are being executed.

TeamCity 9.0+ has a custom logger that provides real-time test reporting.

The logger must be installed manually on the agent machine, as it requires dlls to be copied to the `Extensions` folder of the VSTest.Console. No agent restart is needed when the custom logger is installed.

To install the custom logger:

1. Download the [custom logger](#)
2. Extract the contents of the downloaded archive on the agent machine:
 - for VisualStudio 2015 - to `PROGRAM_FILES\Microsoft Visual Studio 14.0\Common7\IDE\CommonExtensions\Microsoft\TestWindow\Extensions`
 - for VisualStudio 2013 - to `PROGRAM_FILES\Microsoft Visual Studio`

- 12.0\Common7\IDE\CommonExtensions\Microsoft\TestWindow\Extensions
 • for VisualStudio 2012 - to PROGRAM_FILES\Microsoft Visual Studio
 11.0\Common7\IDE\CommonExtensions\Microsoft\TestWindow\Extensions
- Check that the custom logger was installed correctly by executing `vstest.console.exe /ListLoggers` in the console on agent machine. If the logger was installed correctly, you will see the logger with FriendlyName TeamCity listed:

```
VSTest.TeamCityLogger.TeamCityLogger
Uri: logger://TeamCityLogger
FriendlyName: TeamCity{info}
```

MSTest settings

Option	Description
MSTest metadata	Enter a value for <code>/testmetadata:file</code> argument.
Testlist from metadata to run	Edit the Testlist. Every line will be translated into <code>/testlist:line</code> argument.
Test	Names of individual tests to run. This option will be translated to the series of <code>/test: arguments</code>
Unique	Run the test only if one unique match is found for any specified test in test section
Results file	Enter a value for <code>/resultsfile:file</code> command line argument.

Adding Build Features

A "build feature" is a piece of functionality that can be added to a build configuration to affect running builds or reporting build results.

Build features are configured on the dedicated page of the **Build Configuration Settings** available from the list on the left.

 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:

- VCS Labeling
- Ruby Environment Configurator
- Build Files Cleaner (Swabra)
- XML Report Processing
- AssemblyInfo Patcher
- Free disk space
- Performance Monitor
- Shared Resources
- Automatic Merge
- NuGet Feed Credentials
- SSH Agent
- File Content Replacer

VCS Labeling

TeamCity can label (tag) sources of a particular build (automatically or manually) in your version control. The list of labels applied and their application status is displayed on the [Changes tab](#) of the build results page.

On this page:

- [Automatic VCS labeling](#)
- [Manual VCS labeling](#)
- [Subversion Labeling Rules](#)
- [Labeling Rule Examples](#)

Automatic VCS labeling

You can set TeamCity to label the sources of a build depending on the build status automatically. The process takes place in the background after the build finishes and does not affect the build status, which means that a labeling failure is not a standard [notification event](#). However, the users subscribed for [notifications about failed builds](#) of the current build configuration will be notified about a labeling failure.

Any errors encountered during labeling are reported on the [Changes tab](#) of the build results page.

Labeling is configured for a build configuration/template.

Automatic VCS labeling is configured on the [Build Features](#) page of the Build Configuration settings.

To configure automatic labeling, you need to specify the root to label and the labeling pattern. If you have branches configured for your build configuration, you can label builds from [branches you select](#).

You can override the labeling settings inherited from a template completely; you can also apply different labels to different VCS roots.



Labeling uses the credentials specified for the VCS root and the write access to the sources repository is required.

Note that if you change the VCS settings of a build configuration, they will be used for labeling only in the new builds.

"Moving" labels (a label with the same name for different builds, e.g. "SNAPSHOT") are currently supported only for CVS.

For an example of using the Teamcity VCS labeling feature to automate tag creation, refer to this [external posting](#).

Manual VCS labeling

To label the sources manually:

Navigate to the [build results](#) page, click **Actions** and select **Label this build sources** from the drop-down.

Manual labeling uses the VCS settings actual for the build.

Subversion Labeling Rules

To label Subversion VCS roots, additional configuration - [labeling rules](#) defining the SVN repository structure - is required.

Labeling rules are specified as newline-delimited rules in the following format:

```
TrunkOrBranchRepositoryPath => tagDirectoryRepositoryPath
```

The repository paths can be relative and absolute (starting from "/"). Absolute paths are resolved from the SVN repository root (the topmost directory you have in your repository), relative paths are resolved from the TeamCity VCS root.

When creating a label, the sources residing under `TrunkOrBranchRepositoryPath` will be put into the `tagDirectoryRepositoryPath/tagName` directory, where `tagName` is the name of the label as defined by the labeling pattern of the build configuration.

If no sources match the `TrunkOrBranchRepositoryPath`, no label will be created.

The `tagDirectoryRepositoryPath` path must already exist in the repository.

If the `tagDirectoryRepositoryPath` directory already contains a subdirectory with the current label name, the labeling process will fail, and the old tag directory won't be deleted or affected.

For example, there is a VCS root with the URL `svn://address/root/project` where `svn://address/root` is the repository root, and the repository has the structure:

```
-project
--trunk
--branch1
--branch2
--tags
```

In this case the labeling rules should be:

```
/project/trunk=>/project/tags
/project/branch1=>/project/tags
/project/branch2=>/project/tags
```

Labeling Rule Examples

You can use variables substitution in both labeling rules and labeling patterns. See a labeling rule example in a VCS root used in different configurations:

```
/projects/%projectName%/trunk => /projects/%projectName%/tags
```

This will require you to set the `%projectName%` configuration parameter in the build configuration settings.

By default, TeamCity will append the label name to the end of the specified target path. If you want to have a different directory structure and put the label in the middle of the target path, you can use the following syntax:

```
/project/trunk => /tagged_configurations/%%system.build.label%%/project  
/modules/module1/trunk => /tagged_configurations/%%system.build.label%%/module1  
/modules/module2/trunk => /tagged_configurations/%%system.build.label%%/module2
```

Thus, `%%system.build.label%%` will be replaced with the tag name (please note the double `%%` sign at the beginning - it is important).

Ruby Environment Configurator

The *Ruby environment configurator* build feature passes Ruby interpreter to all build steps. The build feature adds the selected Ruby interpreter and gems bin directories to the system PATH environment variable and configures other necessary environment variables in case of the [RVM](#) interpreter. E.g. in the [Command Line](#) build runner you will be able to directly use such commands as `ruby`, `rake`, `gem`, `bundle`, etc. Thus if you want to install gems before launching the [Rake](#) build runner, you need to add the [Command Line](#) build step which launches a custom script, e.g.:

```
gem install rake --no-ri --no-rdoc  
gem install bundler --no-ri --no-rdoc
```

Ruby Environment Configurator Settings

Option	Description
Ruby interpreter path	the path to Ruby interpreter. If not specified, the interpreter will be searched in the <code>PATH</code> . In this field you can use values of environment and system variables. For example: <code>%env.I_AM_DEFINED_IN_BUILDAGENT_CONFIGURATION%</code>
RVM interpreter	Specify here the RVM interpreter name and optionally a gemset configured on a build agent. Note, that the interpreter name cannot be empty. If gemset isn't specified, the default one will be used. (i) This option can be used if you don't want to use the <code>.rvmrc</code> settings, for instance to run tests on different ruby interpreters instead of those hard-coded in the <code>.rvmrc</code> file.
RVM with <code>.rvmrc</code> file	Specify here the path to a <code>.rvmrc</code> file relative to the checkout directory. If the file is specified, TeamCity will fetch environment variables using the rvm-shell and will pass it to all build steps.
Fail build if Ruby interpreter wasn't found	Check the option to fail a build if the Ruby environment configurator cannot pass the Ruby interpreter to the step execution environment because the interpreter wasn't found on the agent.

See also:

[Administrator's Guide: Configuring Build Steps | Command Line | Rake](#)

Build Files Cleaner (Swabra)

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 and until explicitly excluded 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 as 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 dir folders and their content, or - :some/dir, + :some/dir/inner to exclude some/dir folder and all its content except inner subfolder and its content.
	<div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> Note that after removing some exclude rules, it is advisable to run a clean checkout.</div>
Locking processes	Select whether you want Swabra to inspect the checkout directory for processes locking files in this directory, and what to do with such processes. Note that 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](#).

XML Report Processing

The [XML Report processing build feature](#) allows using report files produced by an external tool in TeamCity. TeamCity will parse the specified files on the disk and report the results as the build results.

The report parsing can also be initiated from within the build via [service messages](#).

XML Report Processing supports the following testing frameworks:

- JUnit Ant task
- Maven Surefire/Failsafe plugin
- NUnit-Console XML reports
- MSTest TRX reports (for MSTest 2005/2008/2010/2012/2013)
- Google Test XML reports

and the following code inspection tools:

- FindBugs (code inspections only): only FindBugs native format is supported (see a [sample](#)). The XML report generated by the FindBugs Maven Plugin is **NOT** supported: it has completely different schema layout and elements.
- PMD
- Checkstyle
- JSLint XML reports

and the following code duplicates tools:

- PMD Copy/Paste Detector XML reports

The bundled XML Report Processing plugin monitors the specified report paths, and when the matching files are detected, they are parsed according to the report type specified. For some report types, parsing of partially saved files is supported, so reporting is started as soon as first data is available and more data is reported as it is written on the disk.

The plugin takes into account only the files updated since the build start (determined by means of the last modification file timestamp). [Configuring XML Report Processing](#)

Add XML Report Processing as [a build feature](#) and configure its settings:

- Choose the report type and specify monitoring rules in the form of + | - :path separating them by a comma or new line.



To be processed, report XML files (or a directory) must be located in the checkout directory, and the path must be relative to [this directory](#).

Paths without the + | : prefix are treated as including. Ant-style wildcards are supported, e.g. dir/**.xml means all files with the .xml extension under the "dir" directory).



TeamCity loads generated reports once when they are created, make sure your build procedure generates files with unique names for each tests set without overwriting report files.

- Check the **Verbose output** option to enable detailed logging to the build log.
- For FindBugs report processing, it is necessary to specify the path to the FindBugs installation on the agent. It will be used for retrieving actual bug patterns, categories and their messages.
- For FindBugs, PMD and Checkstyle code inspections reports processing you can specify maximum errors and warnings limits, exceeding which will cause a build failure. Leave these fields blank if there are no limits.

Development Links

See plugin page at [XML Test Reporting](#).

See also:

[Concepts: Build Runner | Testing Frameworks](#)

AssemblyInfo Patcher

The [AssemblyInfo Patcher](#) build feature allows setting a build number to an assembly automatically, without having to patch the `AssemblyInfo.cs` files manually. When adding this build feature, you only need to specify the version format. Note that you can use TeamCity parameter

references here.

When configured, TeamCity searches for all AssemblyInfo (including GlobalAssemblyInfo) files: .cs, .cpp, .fs, .vb in their standard locations under the checkout directory and replaces the parameter for the `AssemblyVersion`, `AssemblyFileVersion` and `AssemblyInformationalVersion` attributes with the build number you have specified in the TeamCity web UI.

At the end of the build the files are reverted to the initial state.

Note that this feature will work only for standard projects, i.e. created by means of the Visual Studio wizard, so that all the `AssemblyInfo` files and content have a standard location.

For replacing a wider range of values in a larger number of files, consider using the [File Content Replacer](#) build feature.

Free disk space

The "Free disk space" build feature allows ensuring the build gets enough disk free space on the build agent.

On this page:

- [Analyzing and freeing disk space](#)
- [Other ways to set the free disk space value](#)
- [Configuring artifacts cache](#)

Analyzing and freeing disk space

Before the build and before each build preparation stage, the agent will check the currently available free disk space. 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`

Configuring artifacts cache

TeamCity build agent maintains a cache of the published and downloaded build artifacts for the purpose of reducing network transfers to the same agent. The cache is stored in `<Build Agent home>\system\artifacts_cache` directory and should be cleaned automatically once [Free disk space](#) build feature is configured correctly.

If caching the artifacts is undesirable (for example when the artifacts are large and not used within TeamCity, or if the artifacts cache directory is located not on the same disk as the build checkout directory, or if the builds do not define the [Free disk space](#) build feature and the default 3Gb is not sufficient for a build), caching artifacts on the agent can be [turned off](#) by adding the `teamcity.agent.filecache.publishing.disabled=true` configuration parameter to a project or one of the build configurations of a project. However, the agent will still cache artifacts downloaded as artifact dependencies.

See also:

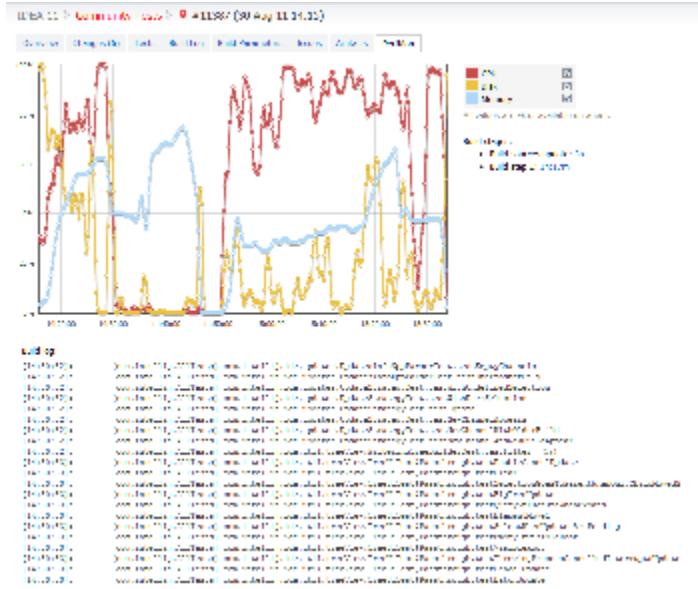
[Administrator's Guide: TeamCity Server Disk Space Watcher](#)

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.

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](#).

On this page:

- [Adding and Editing Shared Resources](#)
 - [Types of Shared Resources](#)
- [Using Shared Resources in Build Configurations](#)
 - [Locks for Resources with Quotas](#)
 - [Lock Fairness](#)
 - [Locks for Resources with Custom Values](#)
- [Development Links](#)

Adding and Editing Shared Resources

You can add, edit shared resources, and explore their details (origin of the resource, its usage, etc.) on the **Shared Resources** tab of the project configuration page.

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

Shared resources defined at a project level are available in all its subprojects and build configurations.

On the subproject level, it is not possible to edit a resource inherited from a project.

However, it is possible to redefine a resource inherited from a project by creating a resource with the same name but with different settings in a subproject.

For example:

1. Create a resource A with the infinite quota in project Parent.
2. Go to its subproject Child 1, and created a resource A (the same name as the resource in Parent) with the quota of 5.

TeamCity will treat this resource as redefined in subproject Child 1: i.e. the settings of the resource A defined at the project level (infinite

quota) will be propagated to all the other subprojects, with the exception of subproject Child 1 where resource A will have the quota of 5.

The usage of a resource is calculated by scanning the subtree of the current project; thus, if several projects use the same resource, only the usages in the current one will be counted.

Types of Shared Resources

When you click **Add new resource**, three types of resources are available:

- **Infinite resource** is a shared resource with an unlimited number of read locks.
- **Resource with quota**: quota is a maximum number of read locks that can be acquired on the resource.
- **Resource with custom values**: a custom value (e.g. a URL) is passed to the build that has acquired a lock on such a resource.

Using Shared Resources in Build Configurations

To define which build configuration(s) will use the resources, [add this build feature](#) to the build configuration settings.

When adding a shared resource, you need to select a resource available to this build configuration and define a lock.

The following locks are available:

Locks for Resources with Quotas

For this resource type, two types of locks are supported:

- **Read locks** - shared (multiple running builds with read locks are allowed).
- **Write locks** - exclusive (only a single running build with a write lock is allowed).

A resource with a quota will allow concurrent access to multiple builds for reading but will restrict access to a single build for writes to the resource. Until all read locks are released, the build requiring a write lock will not start and will wait in the queue while new readers are able to acquire the lock.

 A build with a resource quota set to zero will not run.
Lock Fairness

While read locks enable multiple builds to concurrently access a shared resource with a quota > 1, lock fairness ensures that the build queue is not violated. It means, that if there is a shared resource with a quota > 1, and there are several queued builds with read locks and a build with a write lock in between the readers, the build with the write lock will wait until all builds with read locks **earlier** in the queue finish and release the lock. Then the build requiring a write lock will be executed, and only after that the other readers can acquire the lock. Lock fairness does not allow builds with read locks interfere with build queue ordering and 'slip' past the build that is waiting for a write lock to become available.

Locks for Resources with Custom Values

Resources with custom values support three types of locks:

- Locks on **any available value**: a build that uses the resource will start if at least one of the values is available. If all values are being used at the moment, the build will wait in the queue.
- Locks on **all** values: a build will lock all the values of the resource. No other builds that use this resource will start until the current one is finished.
- Locks on **specific** value: only a specific value of the resource will be passed to the build. If the value is already taken by a running build, the new build will wait in the [queue](#) until the value becomes available.

When the resource is defined and the locks are added, the build gets a configuration parameter with the name of the lock and with the value of the resource string (`teamcity.locks.readLock.<lockName>` or `teamcity.locks.writeLock.<lockName>`), e.g. the parameter name can be: `teamcity.locks.readLock.databaseUrl`.

Development Links

See the Shared Resources plugin page at [TeamCity Shared Resources](#).

See also:

[JetBrains TV TeamCity Shared Resources Screencast](#)

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](#). TeamCity also allows merging branches [manually](#).

Automatic Merge Settings

Check [Adding Build Features](#) for notes on how to add a build feature.

All branches that are used in this feature **must** be present in a repository and included into the **Branch Specification**.

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 (with an optional * placeholder)</code> . Parameter references are supported here.
Merge into branch	A logical name of the destination branch the sources will be merged to. 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 also a related [TeamCity blog post](#).

NuGet Feed Credentials

When using NuGet packages from an authenticated feed during a build on TeamCity, the credentials for connecting to that feed have to be specified.

Adding this information to source control is not a secure practice, so TeamCity provides the **NuGet Feed Credentials** build feature which allows interacting with feeds that require authentication.

When editing the build configuration, from the list of available [Build Features](#) select **NuGet Feed Credentials**. In the dialog that is opened, specify the feed URL and the credentials to connect to the feed.

You can add this build feature to any build configuration, one for every feed that requires authentication.

SSH Agent

The [SSH Agent](#) build feature, available **since TeamCity 9.1**, runs an SSH agent with the selected [uploaded](#) SSH key during a build. When your build script runs an SSH client, it uses the SSH agent with the loaded key.

The first time you connect to a remote host, the SSH client asks if you want to add a remote host's fingerprint to the known hosts database at `~/.ssh/known_hosts`.

To avoid such prompts during a build, you need to configure the known hosts database beforehand. If you trust the hosts you are connecting to, you can disable known hosts checks:

- either **for all connections** by adding something like this in `~/ssh/config`:

```
Host *
  StrictHostKeyChecking no
```

- or **for an individual command** by running an ssh client with the `-o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no` options.

See more information in the man pages for `ssh`, `ssh-agent` and `ssh-add` commands.

 The SSH Agent build feature works on Windows if OpenSSH is installed. You can install it as a part of cygwin, or a part of Git distribution for Windows.

File Content Replacer

This [build feature](#), available **since TeamCity 9.1**, is a general purpose file content replacer allowing you to change a wide range of values in files specified by the user. The replacement is configurable per attribute and can be performed in any files. The changes are valid for the duration of the build and are reverted when the build is finished.

The common case of using File Content Replacer is replacing one attribute at a time choosing particular files for replacement. You can add more than one File Content Replacer build feature if you wish to:

- replace more than one attribute,
- replace one and the same attribute in different files/projects with different values.

This feature extends the capabilities provided by [AssemblyInfo Patcher](#).

Check the [Adding Build Features](#) section for notes on how to add a build feature.

On this page:

- [File Content Replacer Settings](#)
 - [Templates](#)
 - .NET templates
 - MFC templates
 - Xcode templates
 - [Examples](#)
 - Extending an attribute value with a custom suffix
 - Patching only specified files
 - Specifying path patterns which contain spaces
 - Changing only the last version part / build number of the `AssemblyVersion` attribute:

File Content Replacer Settings

You can specify the values manually or use value presets for replacement, which can be edited if needed.

Option	Description
Template (optional)	File content replacer provides a template for every attribute to be replaced. Clicking the Load Template... button displays the combobox with templates containing value presets for replacement. The templates can be filtered by <i>language</i> (e.g. C#), <i>file</i> (e.g. <code>AssemblyInfo</code>) or <i>attribute</i> (e.g. <code>AssemblyVersion</code>) by typing in the combobox. When a template is selected, the settings are automatically filled with predefined values. See the section below for template details.
Look in	Specify the newline- or comma-separated list of files where the values to be replaced will be searched. Ant-like wildcards are supported. <i>If a pre-defined template is selected, the files associated with that template will be used.</i>
File encoding	By default, TeamCity will auto-detect the file encoding. To specify the encoding explicitly, select it from the drop-down. When specifying a custom encoding, make sure it is present on the server and the agents. <i>If a pre-defined template is selected, the file encoding associated with that template will be used.</i>
Find what	Specify a value to be replaced. Regular expressions are accepted in your search criteria. The (?m) multiline flag is on by default. <i>If a pre-defined template is selected, the pattern associated with that template will be used.</i>
Match case:	By default, the comparison is case-sensitive. Uncheck for case-insensitive languages. <i>If a pre-defined template is selected, the comparison associated with that template will be used.</i>

Replace with	Type the values to be used to replace the characters in the Find what box. To delete the characters in the Find what box from your file, leave this box blank.
<p> </p> <ul style="list-style-type: none"> • Reference capturing groups when necessary, e.g. \$1replacement\$2. • Use a backslash to escape each digit immediately following a numbered capturing group (e.g. \$1\1.0.0.0\$2 vs. \$11.0.0.0\$2). • Use a backslash to escape special characters (e.g. \\$). 	

Templates

This section lists the available replacement templates.

.NET templates

The templates for replacing the following [Assembly attributes](#) are provided (listed in comparison with [AssemblyInfo Patcher](#)):

Assembly attribute	Supported by AssemblyInfo Patcher	Supported by File Content Replacer
• <code>CLSCCompliant</code>	No	C# only
• <code>ComVisible</code>	No	C# only
• <code>Guid</code>	No	C# only
• <code>AssemblyTitle</code>	No	C# only
• <code>AssemblyDescription</code>	No	C# only
• <code>AssemblyConfiguration</code>	No	C# only
• <code>AssemblyCompany</code>	No	C# only
• <code>AssemblyProduct</code>	No	C# only
• <code>AssemblyCopyright</code>	No	C# only
• <code>AssemblyTrademark</code>	No	C# only
• <code>AssemblyCulture</code>	No	C# only
• <code>AssemblyVersion</code>	Yes	Yes
• <code>AssemblyFileVersion</code>	Yes	Yes
• <code>AssemblyInformationalVersion</code>	Yes	Yes
• <code>AssemblyKeyFile</code>	No	C# only
• <code>AssemblyKeyName</code>	No	C# only
• <code>AssemblyDelaySign</code>	No	C# only
• <code>InternalsVisibleTo</code>	No	C# only
• <code>AllowPartiallyTrustedCallers</code>	No	C# only
• <code>NeutralResourcesLanguageAttribute</code>	No	C# only

MFC templates

The templates for replacing the following [MFC C++ resource keys](#) are provided:

- `FileDescription`
- `CompanyName`

- ProductName
- LegalCopyright
- FileVersion*
- ProductVersion*

⚠ In MFC *.rc files, both FileVersion and ProductVersion occur twice, once in a *dot-separated* (e.g. 1.2.3.4) and once in a *comma-separated* (e.g. 1,2,3,4) form. If your %build.number% parameter has a format of 1.2.3.{0}, using two build parameters with different delimiters instead of a single %build.number% is recommended.

Xcode templates

The templates for replacing the following **Core Foundation Keys** in Info.plist files are provided:

- CFBundleVersion
- CFBundleShortVersionString
- or both CFBundleVersion and CFBundleShortVersionString at the same time

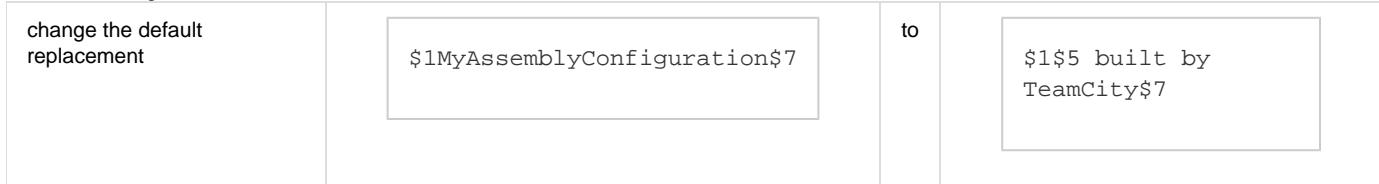
Examples

Extending an attribute value with a custom suffix

Suppose you do not want to replace your AssemblyConfiguration with a fixed literal, but want to preserve your AssemblyConfiguration from AssemblyInfo.cs and just extend it with a custom suffix, e.g.:

```
[assembly: AssemblyConfiguration("${AssemblyConfiguration} built by TeamCity")]
```

Do the following:



For changing complex regex patterns, [this external tool](#) might be useful.

Patching only specified files

The default AssemblyInfo templates follow the usual *Visual Studio* project/solution layout; but a lot of information may be shared across multiple projects and can reside in a shared file (e.g. CommonAssemblyInfo.cs).

Suppose you want to patch this shared file only; or you want to patch AssemblyInfo.cs files on a per-project basis.

Do the following:

1. Load the AssemblyInfo template corresponding to the attribute you are trying to process (e.g. AssemblyVersion)
2. Change the list of file paths in the **Look in** field from the default */Properties/AssemblyInfo.cs to */CommonAssemblyInfo.cs or list several files comma- or new-line separated files here, e.g. myproject1/Properties/AssemblyInfo.cs, myproject2/Properties/AssemblyInfo.cs .

Specifying path patterns which contain spaces

Spaces are usually considered a part of the pattern, unless they follow a comma, as the comma is recognised as a delimiter.

Note that the TeamCity server UI trims leading and trailing spaces in input fields, so a single-line pattern like <spaces>foo.bar will become foo.bar upon save. The following workarounds are available:

For a <i>single</i> pattern containing leading spaces	For [a <i>single</i> pattern containing] trailing spaces	Alternatively, to add a single pattern containing leading spaces, use an explicit inclusion rule:
'<spaces>foo.bar'	foo.bar<spaces>,	+:<spaces>foo.bar

Changing only the last version part / build number of the AssemblyVersion attribute:

Suppose, your AssemblyVersion in AssemblyInfo.cs is Major.Minor.Revision.Build (set to 1.2.3.*), and you want to replace the Build portion (following the last dot (the *) only).

1. Load the AssemblyVersion in AssemblyInfo (C#) template and change the default pattern:

```
(^\\s*\\[\\s*assembly\\s*:\\s*( (System\\s*\\. )?\\s*Reflection\\s*\\. )?\\s*AssemblyVersion(At  
tribute)?\\s*\\(\\s*@?\\")(([0-9\\*])+\\.?) +("\\\\s*\\)\\s*\\])
```

to

```
(^\\s*\\[\\s*assembly\\s*:\\s*( (System\\s*\\. )?\\s*Reflection\\s*\\. )?\\s*AssemblyVersion(At  
tribute)?\\s*\\(\\s*@?\\")(([0-9\\*]+\\.+) +([0-9\\*]+("\\\\s*\\)\\s*\\])
```

and change the default replacement:

```
$1\\%build.number%$7
```

to

```
$1$5\\%build.number%$7
```



%build.number% format

Make sure your %build.number% format contains just a decimal number without any dots, or you may end up with a non-standard version like 1.2.3.4.5.6.2600 (i.e. %build.counter% is a valid value here while 4.5.6.%build.counter% is not).

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, 2.6.3**. Since Teamcity 9.1, NUnit 3.0 is also supported.

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, 2.6.3**.
- 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 the TeamCity build agent environment, the results are reported to the TeamCity agent.



- If you need to access the path to the TeamCity NUnit launcher from some process, you can add the `%system.teamcity.dotnet.nunitlauncher%` environment variable.
- Values surrounded with "%" within custom scripts in the Commandline runner are treated as TeamCity references.

You can pass the following command line options to the TeamCity NUnit Test Launcher:

```
 ${teamcity.dotnet.nunitlauncher} <.NET Framework> <platform> <NUnit vers.>
 [ /category-include:<list> ] [ /category-exclude:<list> ] [ /addin:<list> ] <assemblies to
 test>
```

Option	Description
<.NET Framework>	Version of .NET Framework to run tests. Acceptable values are v1.1 , v2.0 , v4.0 or ANY .
<platform>	Platform to run tests. Acceptable values are x86 , x64 and MSIL . For .NET Framework 1.1 only MSIL option is available.
<NUnit vers.>	Test framework to use. The value has to be specified in the following format: NUnit-<version> . Supported NUnit versions: 2.2.10 , 2.4.1 , 2.4.6 , 2.4.7 , 2.4.8 , 2.5.0 , 2.5.2 , 2.5.3 , 2.5.4 , 2.5.5 , 2.5.6 , 2.5.7 , 2.5.8 , 2.5.9 , 2.5.10 , 2.6.0 , 2.6.1 , 2.6.2 , 2.6.3 . Since Teamcity 9.1, NUnit 3.0 is also supported.
/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 [Parameters](#) page of the Build Configuration.

Run tests from an assembly:

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.2.10 Assembly.dll
```

Run tests from an assembly with NUnit categories filter

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.2.10 /category-include:C1  
/category-exclude:C2 Assembly.dll
```

Run tests from assemblies:

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.5.0 /addin:Addin1.dll;Addin2.dll  
Assembly.dll Assebly2.dll
```

NUnit for NAnt Build Runner

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 follwing options:

- Use NUnit console with TeamCity Addin for NUnit.
- Import xml tests results via XML Test Report plugin.
- Use command line TeamCity NUnit Test Launcher.
- Configure reporting tests manually via service messages.
- To disable `nunit2` task replacement set `teamcity.dotnet.nant.replaceTasks` system property with value `false`.

TeamCity `nunt2` task implementation supports additional options that can be specified either as NAnt `<property>` tasks in the build script, or as **System Properties** under **Build Configuration -> Build Parameters**.

The following options are supported for TeamCity `<nunit2>` task implementation:

Property name	Description
<code>teamcity.dotnet.nant.nunit2.failonfailureatend</code>	Run all tests regardless of the number of failed ones, and fails if at least one test has failed.
<code>teamcity.dotnet.nant.nunit2.platform</code>	Sets desired runtime execution mode for .NET 2.0 on x64 machines. Supported values are x86 , x64 and ANY (default).
<code>teamcity.dotnet.nant.nunit2.platformVersion</code>	Sets desired .NET Framework version. Supported values are v1.1 , v2.0 , v4.0 . Default value is equal to NAnt target framework
<code>teamcity.dotnet.nant.nunit2.version</code>	Specifies which version of the NUnit runner to use. The value has to be specified in the following format: NUnit-<version>  Supported NUnit versions: 2.2.10 , 2.4.1 , 2.4.6 , 2.4.7 , 2.4.8 , 2.5.0 , 2.5.2 , 2.5.3 , 2.5.4 , 2.5.5 , 2.5.6 , 2.5.7 , 2.5.8 , 2.5.9 , 2.5.10 , 2.6.0 , 2.6.1 , 2.6.2 , 2.6.3 . Since Teamcity 9.1, NUnit 3.0 is also supported.
<code>teamcity.dotnet.nant.nunit2.addins</code>	Specifies the list of third-party NUnit addins used for NAnt build runner.

```
teamcity.dotnet.nant.nunit2.runProcessPerAssembly
```

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:

1. Make sure, the `teamcity_dotnet_nunitlauncher` system property is accessible on build agents. Build Agents running windows should automatically detect these properties as environment variables. If you need to set them manually, see defining **agent specific** properties for more information.
2. Configure your MSBuild build script with `NUnitTeamCity` task using the following syntax:

```
<UsingTask TaskName="NUnitTeamCity"
AssemblyFile="$(teamcity_dotnet_nunitlauncher_msbuild_task)" />
```

```
<NUnitTeamCity Assemblies="@{assemblies_to_test}" />
```

The following attributes are supported by `NUnitTeamCity` task:

Property name	description
Platform	Execution mode on a x64 machine. Supported values are: x86 , x64 and ANY .
RuntimeVersion	.NET Framework to use: v1.1 , v2.0 , v4.0 , ANY . By default, the MSBuild runtime is used. Default is v2.0 for MSBuild 2.0 and 3.5. For MSBuild 4.0 default value is v4.0
IncludeCategory	As used in the <code>NUnit</code> task from 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.  Supported NUnit versions: 2.2.10, 2.4.1, 2.4.6, 2.4.7, 2.4.8, 2.5.0, 2.5.2, 2.5.3, 2.5.4, 2.5.5, 2.5.6, 2.5.7, 2.5.8, 2.5.9, 2.5.10, 2.6.0, 2.6.1, 2.6.2, 2.6.3. Since Teamcity 9.1, NUnit 3.0 is also supported. 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-2.4.6" />
</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="" Text="{build.status.text} and some
aftertext" />
```

`{build.status.text}` is substituted with older status text.

Status can have `SUCCESS` value.

NUnit Addins Support

NUnit addin is an extension that plug into NUnit core and changes the way it operates. Refer to the [NUnit addins](#) page for more information. This section covers description of NUnit addins support for:

- [NAnt build runner](#)
- [TeamCity NUnit console launcher](#)
- [MSBuild build runner](#)

NAnt Build Runner

To support NUnit addins for NAnt build runner you need to provide in your build script the `teamcity.dotnet.nant.nunit2.addins` property in the following format:

```
<property name="teamcity.dotnet.nant.nunit2.addins" value="<list of paths>" />
```

where `<list>` is the list of paths to NUnit addins separated by `:`.

For example:

```
<property name="teamcity.dotnet.nant.nunit2.addins"
value="..../tools/addins/MyFirst.AddIn.dll;MySecond.AddIn.dll" />
```

TeamCity NUnit Console Launcher

To support NUnit addins for the console launcher you need to provide the `'/addins:<list of addins separated with ;>'` commandline option.

For example:

```
#{teamcity.dotnet.nunitlauncher}
/addin:..../tools/addins/MyFirst.AddIn.dll;nunit-addins/MySecond.AddIn.dll
```

MSBuild

To support NUnit addins for the MSBuild runner, specify the `Addins` property for the `NUnitTeamCity` task with the following format:

```
Addins="<list>"
```

where `<list>` is the list of addins separated by `;` or `,`.

For example:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
DefaultTargets="build">
  <ItemGroup>
    <TestAssembly Include="$(MSBuildProjectDirectory)/MyTests.dll" />
  </ItemGroup>
  <Target Name="build">
    <NUnitTeamCity Assemblies="@{TestAssembly}"
Addins="..../tools/addins/MyFirst.AddIn.dll;nunit-addins/MySecond.AddIn.dll" />
  </Target>
</Project>
```

TeamCity Addin for NUnit

If you run NUnit tests via the [NUnit console](#) and want TeamCity to track the test results without having to launch the TeamCity test runner, the best solution is to use TeamCity Addin for NUnit. You can plug this addin into NUnit, and the tests will be automatically reported to the TeamCity server.

Alternatively, you can opt to use the [XML Report Processing](#) build feature, or manually configure reporting tests by means of [service messages](#).



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, 2.6.3.**

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 2005/2008/2010/2012/2013 testing framework via parsing of the MSTest results file (.trx file). **Since TeamCity 9.1**, MSTest 2015 is also supported.

 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. **Since TeamCity 9.1 EAP3**, the MSTest runner is merged into the [Visual Studio Tests runner](#).

- Configure [XML Report Processing](#) via build feature or via [service message](#) to parse the .trx reports that are produced by your build procedure.

The easiest way to set up MSTest tests reporting in TeamCity is to add MSTest build runner as one of the steps to your build configuration and specify there all the required parameters.

Please, refer to [MSTest build runner](#) page for details.

If the tests are already run within your build script and MSTest generates .trx reports, you can configure [service messages](#) to parse the reports. Autodetection of MSTest

Prior to TeamCity 9.1 the MSTest location was reported as system properties: %system.MSTest.8.0%, %system.MSTest.9.0%, %system.MSTest.10.0%, %system.MSTest.11.0%, %system.MSTest.12.0%, %system.MSTest.14.0% that referred to MSTest 2008, 2010, 2012, 2013, 2015 correspondingly.

Since TeamCity 9.1 system parameters of the %system.MSTest.xx.yy% format were changed to configuration parameters of the %teamcity.dotnet.mstest.xx.yy% format.

If system properties are required for the build, the [mstest-legacy-provider](#) plugin can be used.

 TeamCity auto-detects MSTest based on the registry values that describe the Visual Studio installation path. If Visual Studio is installed in a non-standard location, or the registry key is corrupted, or the TeamCity agent has no access to the VisualStudio directory, TeamCity may not be able to detect MSTest. In this case, the corresponding configuration parameter of the %teamcity.dotnet.mstest.xx.yy% format must be added to the build manually. It should contain the full path including the MSTest.exe executable, e.g. the default path for MSTest 2013 is C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\MSTest.exe

See also:

[Concepts: Testing Frameworks](#)

[Administrator's Guide: NUnit Support](#)

Configuring .NET Code Coverage

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:

- [JetBrains dotCover](#)
- [NCover](#)
- [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](#)

JetBrains dotCover

TeamCity comes bundled with the console runner of JetBrains dotCover. 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.

On this page:

- [dotCover Settings](#)
- [Compile and Test in Different Builds](#)
 - [Bundled dotCover Versions](#)

dotCover Settings

Path to dotCover Home	Leave this field blank to use the 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: <code>+ :assembly=* ; type=* ; method=***</code> . Use <code>- :assembly</code> 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.
Additional dotCover.exe arguments since TeamCity 9.1 EAP2	Provide a new-line separated list of additional commandline parameters to pass to dotCover.exe

Note that dotCover coverage engine reports statement coverage instead of line coverage.

Compile and Test in Different Builds

To build a consistent coverage report, dotCover has to be able to find source files under the build checkout directory which should be easy if you build binaries and collect coverage in the same build, or if you use different builds, but they use a [snapshot dependency](#) and the same agent as well as the same [VCS settings](#).

If you need to build binaries in one build and collect code coverage in another one using different [checkout settings](#), some additional properties are required. It is assumed that:

- Build configuration **A** compiles code with debugging information and creates an artifact with assemblies and .pdb files
- Build configuration **B** runs tests with dotCover enabled and has a [snapshot dependency](#) on A.

To display the source code in the [Code Coverage tab](#) of build results of B, you need to point B to the same [VCS root](#) as A to get your source code in an appropriate location ([the checkout root](#)) and add an [artifact dependency](#) on **build from the same chain** of A (for dotCover to get the paths to the sources from the .pdb files).

You also need to tell TeamCity where to find the source code.

To do this, perform the following:

1. Add the `teamcity.dotCover.sourceBase` configuration parameter with the value `%teamcity.build.checkoutDir%` to the compiling build configuration A.
2. Add the configuration parameter `dotNetCoverage.dotCover.source.mapping` to your test configuration B with the value `%dep.btA.teamcity.dotCover.sourceBase%=>%teamcity.build.checkoutDir%`, where `btA` is the actual [id](#) of your configuration A.

Bundled dotCover Versions

This section provides information on the versions of dotCover bundled with different TeamCity versions.

TeamCity Version	dotCover Version
TeamCity 9.1.3	dotCover 3.2
TeamCity 9.1.1	dotCover 3.1.1
TeamCity 9.1	dotCover 3.1.1
TeamCity 9.0.4	dotCover 3.0
TeamCity 9.0.3	dotCover 3.0
TeamCity 9.0.2	dotCover 3.0
TeamCity 9.0	dotCover 2.7

TeamCity 8.1.4	dotCover 2.7
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](#)
[Troubleshooting: dotCover issues](#)

NCover

TeamCity supports code coverage with NCover (1.x and 3.x) for NUnit tests run via TeamCity NUnit test runner, which can be configured in one of the following ways: web UI, [command line](#), [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 <code>/ea</code> 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:

1. While creating/editing Build Configuration, go to the Build Steps page.
2. Add one of the build steps that support NCover ([.NET Process Runner](#), [MSBuild](#), [MSpec](#), [MSTest](#), [NAnt](#), [NUnit](#)), configure unit tests.
3. Select **NCover (3.x)** in **.NET coverage tool**.
4. Set up the NCover options - refer to the description of the available options below.

Option	Description
Path to NCover 3	Specify the path to NCover. Alternatively, use <code>%system.ncover.v3.x86.path%</code> or <code>%system.ncover.v3.x64.path%</code> 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 <code>//ias.*</code> to get coverage of all assemblies.
NCover Reporting Arguments	Specify additional NCover reporting arguments, except for the output path. Use <code>//or</code> <code>FullCoverageReport:Html:{teamcity.report.path}</code> to get the report.

Reporting NCover Results Manually

If .NET code coverage is collected by the build script and needs to be reported inside TeamCity (for example, [Rake runner](#), or if you run tests via a test launcher other than TeamCity NUnit Test Launcher), there is a way to let TeamCity know about the coverage data. Please read more at [Manually Configuring Reporting Coverage](#).

PartCover

TeamCity supports code coverage with PartCover (2.2 and 2.3) for NUnit tests run via the TeamCity NUnit test runner, which can be configured in one of the following ways: via the web UI, [command line](#), [NUnitTeamCity task](#), [NUnit task](#), [nunit2 task](#).

Important Notes

- In order to launch coverage, PartCover should be installed on an agent where coverage builds will run.
- You don't need to make any modifications to your build script to enable coverage.
- You don't need to explicitly pass any of the PartCover arguments to the TeamCity NUnit test runner.

To configure PartCover:

1. While creating/editing Build Configuration, go to the Build Runner page.
2. Select **PartCover (2.2 or 2.3)** as a .NET coverage tool.
3. Select the .Net runtime platform and version.

Info 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 [*]* to include all assemblies.
Exclude Assemblies	Explicitly specify the assemblies to be excluded from coverage statistics. If you have specified [*]* to profile all assemblies, type [JetBrains*]* here to exclude TeamCity NUnit test runner sources.
Report XSLT	<p>Write new-line delimited xslt transformation rules in the following format: file.xslt=>generatedFileName.html. You can use the default PartCover xslt as file.xslt, or your own. The Xslt files path is relative to the build checkout directory.</p> <div style="border: 1px solid #ccc; padding: 5px; margin-top: 10px;"> i 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. </div>

Reporting PartCover Results Manually

If .NET code coverage is collected by the build script and needs to be reported inside TeamCity (for example, [Rake runner](#), or if you run tests via a test launcher other than TeamCity NUnit Test Launcher), there is a way to let TeamCity know about the coverage data. Please read more at [Manually Configuring Reporting Coverage](#).

Manually Configuring Reporting Coverage

If you run .Net tests using [NUnit](#), [MSTest](#), [MSpec](#) or [.NET Process Runner](#) runners or run NUnit tests via supported tasks of [MSBuild](#) or [NAnt](#) runners, you can turn on coverage collection in the TeamCity web UI for the specific runner.

For other cases, when the .Net code coverage is collected by the build script and needs to be reported inside TeamCity (for example, [Rake runner](#), or if you run NUnit tests via a test launcher other than TeamCity NUnit Test Launcher), there is a way to let TeamCity know about the coverage data.

First, make sure the build script actually collects the code coverage according to the coverage engine documentation.

Then, report the collected data to TeamCity:

Communication is done via [service messages](#).

First, the build script needs to let TeamCity know details on the coverage engine with the "dotNetCoverage" message.

Then, the build script can issue one or several "importData" messages to import the actual code coverage data files collected.

As a result, TeamCity will display coverage statistics and an HTML report for the coverage.

Configuring Code Coverage Engine

Use the following service message template:

```
##teamcity[dotNetCoverage <key>='<value>' <key1>='<value1>' ...]
```

where **key** is one of the following:

For dotCover (optional):

key	description
dotcover_home	The full path to the dotCover home folder to override the bundled dotCover.

For NCover 3.x:

key	description	sample value
ncover3_home	Full path to NCover installation folder.	The path to the NCover3 installation directory
ncover3_reporter_args	Arguments for the NCover report generator.	Set " //or FullCoverageReport:Html:{teamcity.report.path} " or another NCover commandline argument

For NCover 1.x:

key	description	sample value
-----	-------------	--------------

ncover_explorer_tool	Path to NCoverExplorer.	Path to NCoverExplorer
ncover_explorer_tool_args	Additional arguments for NCover 1.x.	
ncover_explorer_report_type	Value for /report: argument.	1
ncover_explorer_report_order	Value for /sort: argument.	1

For PartCover:

key	description	value
partcover_report_xslts	Write xslt transformation rules one per line (use n as separator) in the following format: file.xslt=>generatedFileName.html.	file.xslt=>generatedFileName.html

Importing Coverage Data Files

To pass xml report generated by a coverage tool to TeamCity, in your build script use the following service message:

```
##teamcity[importData type='dotNetCoverage' tool='<tool name>' path='<path to the results file>']
```

where tool name can be **dotcover**, **partcover**, **ncover** or **ncover3**.

 For dotCover you should send paths to the **snapshot** file that is generated by the `dotCover.exe cover` command.

Java Testing Frameworks Support

TeamCity supports JUnit and TestNG by means of following build runners:

- Ant (when tests are run by junit and testng tasks directly within the script)
- Maven2 (when tests are run by Surefire/Failsafe Maven plugins, on-the-fly reporting is not available.)
- IntelliJ IDEA project: IntelliJ IDEA's JUnit and TestNG run configurations are supported. Note, that such run configurations should be shared and checked in to the version control.

Configuring Java Code Coverage

TeamCity supports Java code coverage based on the IntelliJ IDEA coverage engine, [EMMA](#) open-source toolkit, and JaCoCo. See details in the dedicated sections:

- [IntelliJ IDEA](#)
- [EMMA](#)
- [JaCoCo](#)

See also:

Concepts: [Code Coverage](#)

Administrator's Guide: [IntelliJ IDEA](#) | [EMMA](#)

IntelliJ IDEA

The IntelliJ IDEA coverage engine in TeamCity is the same engine that is used within IntelliJ IDEA to measure code coverage. This coverage attaches to the JVM as a java agent and instruments classes on the fly when they are loaded by the JVM. In particular that means that classes are not changed on the disk and can be safely used for distribution packages.

The IntelliJ IDEA coverage engine currently supports Class, Method and Line coverage. There is no Branch/Block coverage yet.



Make sure your tests run in the `fork=true` mode. Otherwise the coverage data may not be properly collected.

To configure code coverage using IntelliJ IDEA engine, follow these steps:

1. While creating/editing Build Configuration, go to the [Build Step](#) page.
2. Select the [Ant](#), [IntelliJ IDEA Project](#), [Gradle](#) or [Maven](#) build runner.
3. In the **Code Coverage** section, select **IntelliJ IDEA** as a coverage tool in the **Choose coverage** runner drop-down.
4. Set up the coverage options - refer to the description of the available options below.

Option	Description
Classes to instrument	Specify Java packages for which code coverage will be gathered. Use new-line delimited patterns that start with a valid package name and contain *. For example: org.apache.*.
Classes to exclude from instrumentation	Use newline-separated patterns for fully qualified class names to be excluded from the coverage, for example: *Test. Exclude patterns have priority over include patterns.

See also:

Concepts: [Build Runner](#) | [Code Coverage](#)

Administrator's Guide: [Configuring Java Code Coverage | EMMA](#)

EMMA

EMMA Integration Notes

The following steps are performed when collecting coverage with EMMA:

1. After each compilation step (with javac/javac2), the build agent invokes EMMA to instrument the compiled classes and to store the location of the source files. As a result, the `coverage.em` file containing the classes metadata is created in the build checkout directory. The collected source paths of the java files are used to generate the final HTML report.

 All `coverage.*` files are removed in the beginning of the build, so you have to ensure that full recompilation of sources is performed in the build to have the actual `coverage.em` file.
2. Test run. At this stage, the actual runtime coverage information is collected. This process results in creation of the `coverage.ec` file. If there are several test tasks, data is appended to `coverage.ec`.
3. Report generation. When the build ends, TeamCity generates an HTML coverage report, creates the `coverage.zip` file with the report and uploads it to the server. It also generates and uploads the summary report in the `coverage.txt` file, and the original `coverage.ec` 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="border: 2px solid red; padding: 5px; margin-top: 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

TeamCity supports [JaCoCo](#), a Java Code Coverage tool allowing you to measure a wide set of coverage metrics and code complexity.

JaCoCo is available for the following build runners: [Ant](#), [IntelliJ IDEA Project](#), [Gradle](#) and [Maven](#).



To ensure the coverage data is collected properly, make sure your tests run in (one or more) separate JVMs.

- Ant and IntelliJ Idea Project runners: this is the default setting for [TestNG](#), for [Junit test task](#), set `fork=true`.
- Maven runner: set `forkCount` to a value [higher than 0](#).
- Gradle runner: this is the default setting for [Gradle tests](#).

On this page:

- [Enabling JaCoco coverage](#)
- [Importing JaCoco coverage data to TeamCity](#)

Enabling JaCoco coverage

TeamCity supports the java agent coverage mode allowing you to collect coverage without modifying build scripts or binaries. No additional build steps needed - just choose JaCoCo coverage in a build step which runs tests:

1. In the **Code Coverage** section, select **JaCoCo** as a coverage tool in the **Choose coverage runner** drop-down.
2. Set up the coverage options - refer to the description of the available options below.

Option	Description	Example
Classfile directories or jars	Newline-delimited set of path patterns in the form of <code>+ -:[path]</code> to scan for classfiles to be analyzed. Libraries and test classfiles don't have to be listed unless their coverage is wanted.	<code>+:target/main/java/**</code>
Classes to instrument	Newline-delimited set of classname patterns in the form of <code>+ -:[path]</code> . Allows filtering out unwanted classes listed in " Classfile directories or jars " field. Useful in case test classes are compiled.	<code>+:com.package.core.* -:com.package.*Test*</code>



By default, in TeamCity the `jacoco.sources` property is set to `".`, which means that TeamCity will scan whole checkout directory including all subdirectories for your sources.

Check that your classfiles are compiled with debug information (including the source file info) to see with highlighted source code in the report.

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

Importing JaCoco coverage data to TeamCity

Since **TeamCity 9.0**, TeamCity is able to parse JaCoco coverage data and generate a report using a [service message](#) of the following format:

```
##teamcity[jacocoReport dataPath='<path to jacoco.exec file>']
```

Attribute name	Description	Default value	Example
dataPath	Space-delimited set of paths relative to the checkout directory to read the jacoco data file		jacocoResults/jacoco.exec jacocoResults/anotherJacocoRun.exec
includes	Space-delimited set of classname include patterns	*	com.package.core.* com.package.api.*
excludes	Space-delimited set of classname exclude patterns		com.package.test.* .*Test
sources	Space-delimited set of paths relative to the checkout directory to read sources from. Does not need to be listed by default.	.	src
classpath	Space-delimited set of path patterns in the form of + -:[path] to scan for classfiles to be analyzed. Libraries and test classfiles do not need to be listed unless their coverage is wanted.	+:**/*	+:target/main/java/**
reportDir	Path to the directory to store temporary files. The report will be generated as coverage.zip under this directory. Check that there is no existing directory with the same name.	A random directory under Agent's temp directory	jacocoReport

An example of a complete service message:

```
##teamcity[jacocoReport dataPath='jacoco.exec' includes='com.package.core.*'
classpath='classes/lib/some.jar' reportDir='temp/jacocoReport']
```

Running Risk Group Tests First

This section covers:

- Reordering Risk Tests for JUnit and TestNG
 - JUnit
 - TestNG
 - TestNG versions less than 5.14:
 - TestNG versions 5.14 or newer:
- Reordering Risk Tests for NUnit

Reordering Risk Tests for JUnit and TestNG

Supported environments:

- Ant and IntelliJ IDEA Project runners
- JUnit and TestNG frameworks when tests are started with usual JUnit or TestNG tasks

 TeamCity also allows to [implement tests reordering feature for a custom build runner](#).

You can instruct TeamCity to run some tests before others. You can do this on the build runner settings page. Currently there are two groups of tests that TeamCity can run first:

- recently failed tests, i.e. the tests failed in previous finished or running builds as well as tests having high failure rate (a so called blinking tests)
- new and modified tests, i.e. tests added or modified in changelists included in the running build



The recently added or modified tests in your [personal build](#) have the highest priority, and these tests run even before any other reordered test.

TeamCity allows you to enable both of these groups or each one separately.

TeamCity operates on test case basis, that is not the individual tests are reordered, but the full test cases. The tests cases are reordered only within a single test execution Ant task, so to maximize the feature effect, use a single test execution task per build.

Tests reordering works the following way:

JUnit

1. TeamCity provides tests that should be run first (test classes).
2. When a JUnit task starts, TeamCity checks whether it includes these tests.
3. If at least one test is included, TeamCity generates a new fileset containing included tests only and processes it before all other filesets. It also patches other filesets to exclude tests added to the automatically generated fileset.
4. After that JUnit starts and runs as usual.



Some cases when automatic tests reordering will not work:

- if JUnit suites are created manually in test cases with help of suite() method
- if @RunWith annotation is used in JUnit4 tests

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 in the **Failure Conditions** section of the of the Build

Configuration Settings page.

On this page:

- Common build failure conditions
- Additional Failure Conditions
 - Fail build on metric change
 - Adding custom build metric
 - Fail build on specific text in build log

Common build failure conditions

In the Common Failure Conditions, specify how TeamCity will fail builds by selecting appropriate options from in the **Fail build if** area:

- **it runs longer than ... minutes:** Check this option and enter a value in minutes to enable execution timeout for 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.
- **the build process exit code is not zero:** Check this option to mark the build as failed if the build process doesn't exit successfully.
- **at least one test failed:** Check this option to mark the build as failed if the build fails at least one test. If this option is not checked, the build can be marked successful even if it fails to pass a number of tests. Regardless of this option, TeamCity will run all build steps.
- **an error message is logged by build runner:** Check this option to mark the build as failed if the build runner reports an error while building.
- **an out of memory or crash is detected (Java only):** Check this option to mark the build as failed if a crash of the JVM is detected, or Java out of memory problems. If possible, TeamCity will upload crash logs and memory dumps as artifacts for such builds.

Additional Failure Conditions

You can instruct TeamCity to mark a build as failed if some of its metrics has deteriorated in comparison with another build, e.g. code coverage, artifacts size, etc. For instance, you can mark build as failed if the code duplicates number is higher than in the previous build.

Another build failure condition causes TeamCity to mark build as failed when a certain text is present in the build log.

To add such build failure condition to your build configuration, click **Add build failure condition** and select from the list:

- Fail build on metric change
- Fail build on specific text in build log

 You can disable a build failure condition temporarily or permanently at any time, even if it is inherited from a build configuration template.

Fail build on metric change

When using code examining tools in your build, like code coverage, duplicates finders, inspections and so on, your build generates various numeric metrics. For these metrics you can specify a threshold which, when exceeded, will fail a build.

In general there are two ways to configure this build fail condition:

- A *build metric* exceeds or is less than the specified threshold. For example: **Fail build if** build duration (secs) compared to constant value is more than* 300. In this case a build will fail if it runs more than 300 seconds.
- A *build metric* has changed comparing to a specific build by a specified value. For example: **Fail build if** its build duration (secs) compared to a value from another build is more by at least 300 default units for this metric than the value in the Last successful build. In this case a build will fail if it runs 300 seconds longer than the last successful build. If Branch specification is configured, then the following logic is applied.

The following builds can be used as the basis for comparing build metrics:

- last successful build
- last pinned build
- last finished build
- build with specified build number
- last finished build with specified tag.

By default, TeamCity provides the wide range of *build metrics*:

- artifacts size(bytes)
- build duration (secs)
- number of classes
- number of code duplicated
- number of covered classes
- number of covered lines
- number of covered methods
- number of failed tests

- number of ignored tests
- number of inspection errors
- number of inspection warnings
- number of lines of code
- number of methods
- number of passed tests
- number of tests
- percentage of block coverage
- percentage of class coverage
- percentage of line coverage
- percentage of method coverage
- percentage of statement coverage
- test duration (secs)
- total artifacts size (bytes)

 Note that **since TeamCity 9.0**, the way TeamCity counts tests [has changed](#).
Adding custom build metric

You can add your own build metric. To do so, you need to modify the TeamCity configuration file `<TeamCity Data Directory>/config/main-config.xml` and add the following section there:

```
<build-metrics>
<statisticValue key="myMetric" description="build metric for number of files"/>
</build-metrics>
```

So, if your build publishes the `myMetric` value, you can use it as a criterion for a build failure.

Fail build on specific text in build log

TeamCity can inspect all lines in build log for some particular text occurrence that indicates a build failure.
Lines are matched without the time and block name prefixes which precede each message in the build log representation.

To configure this build failure condition, specify:

- a string or a Java Regular Expression whose presence/absence in the build log is an indicator of a build failure,
- a failure message to be displayed on the build results page when a build fails due to this condition.

Configuring Build Triggers

Once a build configuration is created, builds can be triggered manually by clicking the [Run button](#) or initiated automatically with the help of Triggers.

A *build trigger* is a rule which initiates a new build on certain events. The build is put into the [build queue](#) and is started when there are agents available to run it.

While creating/editing a build configuration, you can configure triggers using the **Triggers** sections of the Build Configuration Settings page by clicking [Add new trigger](#) and specifying the trigger settings. For configuration details on each trigger, refer to the corresponding sections.

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 package update detected in the NuGet repository.

 Note that if you create a build configuration from a template, it inherits build triggers defined in the template, and they cannot be edited or deleted. However, you can specify additional triggers or disable a trigger permanently or temporarily.

In addition to the triggers defined for the build configuration, you can also trigger a build by an [HTTP GET request](#), or manually by running a custom build.

Configuring VCS Triggers

VCS triggers automatically start a new build each time TeamCity detects new changes in the configured [VCS roots](#).

A new VCS trigger with the default settings triggers a build each time new changes are detected: the version control is polled for changes according to the [Checking for changes interval](#) of a VCS root. Newly detected changes appear as *Pending Changes* of a build configuration. If several check-ins are made during this time, only **one build will be triggered**. If you have several VCS roots attached to a build configuration, TeamCity will add the build to the queue only **after the longest of the specified intervals**.

After the last change is detected, there is a [quiet period](#) before the build is started.

The global default value for both options is 60 seconds and can be configured for the server on the [Administration | Global Settings](#) page.

You can adjust a VCS trigger to your needs using the options described below:

- [Trigger a build on changes in snapshot dependencies](#)
- [Per-check-in Triggering](#)
- [Quiet Period Settings](#)
- [VCS Trigger Rules](#)
 - [Trigger Rules Example](#)
- [Branch Filter](#)

Trigger a build on changes in snapshot dependencies

If you have a [build chain](#) (i.e. a number of build configurations interconnected by [snapshot dependencies](#)) you should configure the triggers in the build you want to get in the result, the top-most build. When the build is triggered, all its snapshot dependencies are triggered too.

If you use different VCS settings in the builds of a chain you might benefit from enabling the **Trigger a build on changes in snapshot dependencies** VCS trigger 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

When this option is **not** enabled, several check-ins by different committers can be made; and once they are detected, TeamCity will add only one build to the queue with all of these changes.

If you have fast builds and enough build agents, you can make TeamCity launch a new build **for each check-in** ensuring that no other changes get into the same build. To do that, select the **Trigger a build on each check-in** option. If you select the **Include several check-ins in build if they are from the same committer** option, and TeamCity will detect a number of pending changes, it will group them by user and start builds having single user changes only.

This helps to figure out whose change broke a build or caused a new test failure, should such issue arise.

Quiet Period Settings

By specifying the quiet period you can ensure the build is not triggered in the middle of non-atomic check-ins consisting of several VCS check-ins.

A **quiet period** is a period (in seconds) that TeamCity maintains between the moment the last VCS change is detected and a build is added into the queue. If new VCS change is detected in the Build Configuration within the period, the period starts over from the new change detection time. The build is added into the queue only if there were no new VCS changes detected within the quiet period. Note that the actual quiet period will not be less than the maximum [checking for changes interval](#) among the VCS roots of a build configuration, as TeamCity must ensure that changes were collected at least once during the quiet period.

The quiet period can be set to the default value (60 seconds, can be changed globally at the [Administration | Global Settings](#) page) or to a custom value for a build configuration.

 Note, that when a build is triggered by a trigger with the VCS quiet period set, the build is put into the queue with fixed VCS revisions. This ensures the build will be started with only the specific changes included. Under certain circumstances this build can later become a [History Build](#).

VCS Trigger Rules

If no trigger rules specified, a build is triggered upon any detected change displayed for the build configuration. You can affect the changes detected by changing the VCS root settings and specifying [Checkout Rules](#).

To limit the changes that trigger the build, use the VCS trigger rules. You can add these rules manually in the text area (one per line), or use the **Add new rule** option to generate them.

Each rule is either an "include" (starts with "+") or an "exclude" (starts with "-").

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, the "?" pattern is **not** supported. The file paths in the rule can be relative (not started with '/' or '\') to match resulting paths on the agent or absolute (started with '/') to match VCS paths relative to a VCS root. For each file in a change the most specific rule is found (the rule matching the longest file path). The build is triggered if there is at least one file with a matching "include" rule or a file with no matching rules.
- **VCS_username** - if specified, limits the rule only to the changes made by a user with the corresponding [VCS username](#).
- **VCS_root_id** - if specified, limits the rule only to the changes from the corresponding VCS root.
- **VCS_comment_regexp** - if specified, limits the rule only to the changes that contain specified text in the VCS comment. Use the [Java Regular Expression](#) pattern for matching the text in a comment (see examples below). The rule matches if the comment text contains a matched text portion; to match the entire text, include the ^ and \$ special characters.

 When entering rules, please note that as soon as you enter any "+" rule, TeamCity will remove the default "include all" setting. To include all the files, use "+:." rule.

Also, rules are sorted according to path specificity. I.e. if you have an explicit inclusion rule for /some/path, and exclusion rule -:user=r=some_user: . for all paths, commits to the /some/path from some_user will be **included** unless you add a specific exclusion rule for this user and this path at once, like -:user=some_user:/some/path/**

Trigger Rules Example

```
+:  
-: **.html  
-:user=techwriter;root=InternalSVN:/misc/doc/*.xml  
-:lib/**  
-:comment=minor:**  
-:comment=^oops$:**
```

Here,

- "-: **.html" excludes all .html files from triggering a build.
- "-:user=techwriter;root=InternalSVN:/misc/doc/*.xml" excludes builds being triggered by .xml files checked in by the **VC S 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 principle](#) s ^ and \$ in pattern stand for string beginning and ending)

Branch Filter

The *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.

By default, the branch filter in the VCS Trigger is set to accept all branches (+ : *), which is also the equivalent of the **empty** branch filter.

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. The [Builds Schedule](#) page of the current project settings displays the configured build times.

This section describes the triggering conditions used by the Schedule Trigger.

On this page:

- [Date and Time](#)
 - [Examples](#)
 - [Brief description of the cron format used](#)
- [VCS Settings](#)
 - [VCS Trigger Rules](#)
 - [Trigger Rules Example](#)
 - [Branch Filter](#)
 - [Build Changes](#)

Date and Time

Besides triggering builds **daily** or **weekly** at a specified time for a particular time zone, you can specify advanced time settings using **cron-like** expressions. This format provides more flexible scheduling options.

TeamCity uses [Quartz](#) for working with cron expressions.

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 the VCS root settings and specifying [Checkout Rules](#).

To limit the changes that trigger the build, use the VCS trigger rules. You can add these rules manually in the text area (one per line), or use the **Add new rule** option to generate them.

Each rule is either an "include" (starts with "+") or an "exclude" (starts with "-").

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, the "?" pattern is **not** supported. The file paths in the rule can be relative (not started with '/') or absolute (started with '/') to match resulting paths on the agent or absolute (started with '/') to match VCS paths relative to a VCS root. For each file in a change the most specific rule is found (the rule matching the longest file path). The build is triggered if there is at least one file with a matching "include" rule or a file with no matching rules.
- **VCS_username** - if specified, limits the rule only to the changes made by a user with the corresponding [VCS username](#).
- **VCS_root_id** - if specified, limits the rule only to the changes from the corresponding VCS root.
- **VCS_comment_regexp** - if specified, limits the rule only to the changes that contain specified text in the VCS comment. Use the [Java Regular Expression](#) pattern for matching the text in a comment (see examples below). The rule matches if the comment text contains a matched text portion; to match the entire text, include the ^ and \$ special characters.

i When entering rules, please note that as soon as you enter any "+" rule, TeamCity will remove the default "include all" setting. To include all the files, use "+:." rule.

Also, rules are sorted according to path specificity. I.e. if you have an explicit inclusion rule for `/some/path`, and exclusion rule `-:user=some_user: .` for all paths, commits to the `/some/path` from `some_user` will be **included** unless you add a specific exclusion rule for this user and this path at once, like `-:user=some_user:/some/path/**`

Trigger Rules Example

```
+ : .
- : **.html
- :user=techwriter;root=InternalSVN:/misc/doc/*.xml
- :lib/**
- :comment=minor:**
- :comment=^oops$:**
```

Here,

- "-:**.html" excludes all .html files from triggering a build.
- "-:user=techwriter;root=InternalSVN:/misc/doc/*.xml" excludes builds being triggered by .xml files checked in by the VC S 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 principle](#) s ^ and \$ in pattern stand for string beginning and ending)

Branch Filter

The *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.

By default, the branch filter in the VCS Trigger is set to accept all branches (+ : *), which is also the equivalent of the **empty** branch filter.

Examples:

Only default branch is accepted:

```
+ :<default>
```

All branches except default are accepted:

```
+ : *  
- :<default>
```

Only branches with with `feature-` prefix are accepted:

```
+ :feature-*
```

The branch filter in the Schedule Trigger works as follows:

- if the option **trigger build only if there are pending changes** is turned **ON**, then the trigger will add a build to the queue for all branches matched by the trigger branch filter where pending changes exist
- if **trigger build only if there are pending changes** is turned **OFF**, then the trigger will add a build to the queue for all branches matched by the trigger branch filter regardless of presence of pending changes there

By default, the branch filter in the Schedule Trigger is set to accept only the default branch (+ : <default>), which is also the equivalent of the **empty** branch filter.

Build Changes

Since TeamCity 9.1, the Schedule Trigger has the ability to watch builds in other build configurations and trigger a build if these builds change.

In the **Build Changes** section, select the corresponding box and specify the build configuration and the type of build to watch: last successful build, last pinned build, last finished build, or the last finished build with a specified tag.

TeamCity can promote a build if there is a dependency (snapshot or artifact) on its build configuration.

Builds Schedule

The **Builds Schedule** page in the administration area of a specific project displays [schedule triggers](#) configured for [build configurations](#) belonging to this project.

[Builds Schedule](#) for the [Root Project](#) displays the list of triggers for the entire TeamCity server.

You can conveniently view the available schedule and plan your builds optimizing allocation of dependent hardware/software resources.

From this page it is also possible to [edit](#), disable or delete the triggers.

Since TeamCity 9.1, the Build Schedule page displays the information for [paused build configurations](#).

Configuring Maven Triggers

The **Triggers** page of the Build Configuration Settings allows you to add the following Maven dependency triggers:

- [Maven Snapshot Dependency Trigger](#)
- [Maven Artifact Dependency Trigger](#)
 - [Advanced Options](#)
 - [Version Ranges](#)

Checksum Based Triggering

The trigger checks if the content of the dependency has actually changed by verifying its checksum from the repository against the locally stored version. Before triggering a build, TeamCity tries to determine the checksum of the required dependency by downloading the file digest (MD5/SHA-1) associated with that artifact.

If the checksum can be retrieved, and it matches a locally stored one, no build is triggered. If the checksum is different, a build is triggered.

If the checksum cannot be retrieved from the remote server, the dependency will be downloaded, TeamCity will calculate its checksum and follow the build triggering mechanism described above.

Maven Snapshot Dependency Trigger

Maven snapshot dependency trigger adds a new build to the queue when there is a real modification of the snapshot dependency content in the remote repository which is detected by the checksum change.

Dependency artifacts are resolved according to the POM and the server-side [Maven Settings](#).



Note that since Maven deploys artifacts to remote repositories sequentially during a build, not all artifacts may be up-to-date at the moment the snapshot dependency trigger detects the first updated artifact.

To avoid inconsistency, select the **Do not trigger a build if currently running builds can produce snapshot dependencies** check box when adding this trigger, which will ensure the build won't start while builds producing snapshot dependencies are still running.



Simultaneous usage of snapshot dependency and dependency trigger for a build

Assume build A depends on build B by both snapshot and trigger dependency. Then, after the build B finishes, build A will be added into the queue, only if build B is not a part of the build chain containing A.

Maven Artifact Dependency Trigger

Maven artifact dependency trigger adds build to the queue when there is a real modification of the dependency content which is detected by the checksum change.

To add a trigger, specify the following parameters in the **Add New Trigger** dialog:

Parameter	Description
Group Id	Specify an identifier of a group the desired Maven artifact belongs to.
Artifact Id	Specify the artifact's identifier.
Version or Version range	Specify a version or version range of the artifact. The version range syntax is described in the section below. SNAPSHOT versions can also be used.
Type	Define explicitly the type of the specified artifact. By default, the type is <code>.jar</code> .
Classifier	(Optional) Specify the classifier of an artifact.
Maven repository URL	Specify a URL to the Maven repository. Note that this parameter is optional. If the URL is not specified, then: <ul style="list-style-type: none">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.

Advanced Options

Since **TeamCity 9.0 EAP2**, the following advanced options have been added to the trigger:

Parameter	Description
Repository ID	Allows using authorization from the effective Maven settings
User settings selection	Allows selecting effective settings. The same as User Settings of the Maven runner.

TeamCity determines the **effective repository** to be checked for the artifact updates and to trigger builds if changes are detected as follows:

- if a URL and Repository ID are set, authentication will be chosen from the effective settings (see below)
- if only a URL is set, the old behavior is preserved: a temporary repository ID is used ("`_tc_temp_remote_repo`")
- if URL is not set (regardless of the Repository ID), the artifact will be looked up in a repository available according to the effective settings.

TeamCity determines **effective settings** as follows:

- in the trigger settings a user can choose among the default, custom or uploaded Maven settings. See [Maven Server-Side Settings](#) for details.
- if no specific settings are configured for the trigger, [Maven](#) build step settings are used
- if no settings for the trigger are configured and there are no Maven build steps, the default [server Maven settings](#) will be used.

Version Ranges

For specifying version ranges use the following syntax, as proposed in the [Maven documentation](#).

Note that Maven Artifact Dependency Trigger can be used not only for fixed-version artifacts but also for snapshots as a fine-grained alternative to the Maven Snapshots Dependency Trigger.

Range	Meaning
(, 1.0]	$x \leq 1.0$
1.0	"Soft" requirement on 1.0 (just a recommendation - helps select the correct version if it matches all ranges)
[1.0]	Hard requirement on 1.0
[1.2 , 1.3]	$1.2 \leq x \leq 1.3$

[1.0 , 2.0)	$1.0 \leq x < 2.0$
[1.5 ,)	$x \geq 1.5$
(,1.0] , [1.2 ,)	$x \leq 1.0$ or $x \geq 1.2$. Multiple sets are comma-separated
(,1.1) , (1.1 ,)	This excludes 1.1, if it is known not to work in combination with this library
1.0-SNAPSHOT	The trigger will check the latest snapshot version for updates

NuGet Dependency Trigger

The NuGet Dependency Trigger allows starting a new build if a NuGet packages update is detected in the NuGet repository.

Requirements and limitations

For a TeamCity server running on **Windows, .NET 4.0** is required.

For a TeamCity server running on **Linux**, the NuGet dependency trigger will reportedly work with the following **limitations**:

- filtering by Package Version Spec is not supported
- only HTTP package sources are supported
- NuGet feed version 1.0 is used, so case-sensitivity issues might occur
- the current trigger implementation on Linux might increase the server load
- authentication issues might occur

Configuring NuGet Dependency Trigger

- Select the NuGet version to use from the **NuGet.exe** drop-down list (if you have installed NuGet beforehand), or specify a custom path to `NuGet.exe`;
- Specify the NuGet package source, if it is different from `nuget.org`;
- Specify the credentials to access NuGet feed if required
- Enter the 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.

You can also opt to trigger build if pre-release package version is detected by selecting corresponding check box. Note that this is only supported for NuGet version 1.8 or newer.

See also:

[Administrator's Guide: NuGet](#)

Configuring Finish Build Trigger

The *Finish build trigger* triggers a build of the current build configuration when a build of the selected build configuration is finished. If the **Trigger after successful build only** checkbox is enabled, a build is triggered only after a successful build of the selected configuration.

To monitor builds in other build configurations and trigger a build if these builds change, please see [this option](#) of the Schedule build trigger.

i In most of the cases the Finish Build Trigger should be used with snapshot dependencies, i.e. the current build configuration where the trigger is defined should have a direct or an indirect snapshot dependency on the build configuration selected in the trigger. If there is no snapshot dependency, the following limitations exist:

- it is likely that a build of the build configuration being triggered will not have the same revisions as the finished build even if both configurations have the same VCS settings
- if a build configuration with the Finish Build Trigger has an artifact dependency on the last finished build of the build configuration specified in the trigger settings, there is no guarantee that artifacts of a build which caused build triggering will be used, because, while the triggered build sits in the build queue, another build may finish
- the build triggered by the Finish Build Trigger will always be triggered in the default branch even if the finished build has some other branch

All these limitations **do not apply** if a build configuration with "Finish build trigger" has a snapshot dependency on the selected build configuration. In this case the trigger will run build on the same revisions and will attach the build to the chain. It will also use consistent artifacts if they are produced by dependencies.

Branch Filter

If the build configuration selected in the Finish Build Trigger uses **Feature branches**, you can enable the branch filter to limit the branches where

finished builds will trigger new builds of the current configuration.

The *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-*
```

By **default**, the branch filter in the Finish Build Trigger is set to accept only the default branch (+:<default>), which is also the equivalent of the **empty** branch filter.

Configuring Dependencies

A build configuration can be made dependent on the artifacts or sources of builds of some other build configurations.

For [snapshot dependencies](#), TeamCity will run all dependent builds on the sources taken at the moment the build they depend on starts.

For [artifact dependencies](#), before a build is started, all artifacts this build depends on will be downloaded and placed in their configured target locations and then will be used by the build.



The dependencies of the build can later be viewed on the build results page - the Dependencies tab. This tab also displays indirect dependencies, e.g. if a build A depends on a build B which depends on builds C and D, then these builds C and D are indirect dependencies for build A.

See also:

Concepts: [Dependent Build](#)

Administrator's Guide: [Accessing artifacts via HTTP](#) | [Snapshot Dependencies](#) | [Artifact Dependencies](#)

External Resources: <http://ant.apache.org/ivy/> (additional information on Ivy)

Artifact Dependencies

This page details configuration of the TeamCity Artifact Dependencies.

- [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:

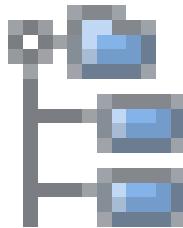
1. When [creating/editing a build configuration](#), open the **Dependencies** page.
2. Click the **Add new artifact dependency** link and specify the following settings:

Option	Description
Depend on	Specify the build configuration for the current build configuration to depend on. A dependency can be configured on a previous build of the same build configuration
Get artifacts from	Specify the type of build whose artifacts are to be taken: last successful build, last pinned build, last finished build, build from the same chain (this option is useful when you have a snapshot dependency and want to obtain artifacts from a build with the same sources), build with a specific build number or the last finished build with a specified tag.
	<p> • 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.</p> <p>• 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.</p>
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	<i>This field appears if the dependency has a branch specified in the VCS Root settings.</i> 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	<p>A newline-delimited set of rules. Each rule must have the following syntax:</p> <pre>[+ : - :]SourcePath[!ArchivePath] [=>DestinationPath]</pre> <p>Each rule specifies the files to be downloaded form the "source" build. The <i>SourcePath</i> 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.</p> <p>Downloaded artifacts will keep the "source" directory structure starting with the first * or ?.</p> <p><i>DestinationPath</i> specifies the destination directory on the agent where downloaded artifacts are to be placed. If the path is relative (which is recommended), it will be resolved against the build checkout directory. If needed, the destination directories can be cleaned before downloading artifacts. If the destination path is empty, artifacts will be downloaded directly to the checkout root.</p> <p>Basic examples:</p> <ul style="list-style-type: none"> • Use <i>a/b/**=>lib</i> to download all files from <i>a/b</i> directory of the source build to the <i>lib</i> directory. If there is a <i>a/b/c/file.txt</i> file in the source build artifacts, it will be downloaded into the file <i>lib/c/file.txt</i>. • At the same time, artifact dependency <i>**/*.*txt=>lib</i> will preserve the directories structure: the <i>a/b/c/file.txt</i> file from source build artifacts will be downloaded to <i>lib/a/b/c/file.txt</i>. <p><i>ArchivePath</i> is used to extract downloaded compressed artifacts. Zip, 7-zip, jar, tar and tar.gz are supported. <i>ArchivePath</i> follows general rules for <i>SourcePath</i>: 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)</p> <p>For example: <i>release.zip!*.dll</i> command will extract all .dll files residing in the root of the <i>release.zip</i> artifact.</p> <p>Archive processing examples:</p> <ul style="list-style-type: none"> • <i>release-* .zip!*.dll=>d1ls</i> will extract *.dll from all archives matching the <i>release-* .zip</i> pattern to the <i>d1ls</i> directory. • <i>a.zip!**=>destination</i> will unpack the entire archive saving the path information. • <i>a.zip!a/b/c/**/*.*dll=>d1ls</i> will extract all .dll files from <i>a/b/c</i> and its subdirectories into the <i>d1ls</i> directory, without the <i>a/b/c</i> 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"
  passwd="PASSWORD" />
```

where `TEAMCITY_HOST` is hostname or IP address of your TeamCity server (without port and servlet context).

As `USER_ID/PASSWORD` you can use either username/password of a regular TeamCity user (the user should have corresponding permissions to access artifacts of the source build configuration) or system properties `teamcity.auth.userId/teamcity.auth.password`.

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.
	<p> 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.</p>
Only use successful builds from suitable ones	A new triggered build will only "use" successfully finished suitable builds as dependencies. If the latest finished "suitable" build is failed, it will be re-run.
Run build on the same agent	When enabled, and B snapshot-dependes 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.
On failed dependency/ On failed to start/canceled dependency	<p>Since TeamCity 9.0 if the dependency fails, you can manage the status of the dependent build by selecting one of the following options:</p> <ul style="list-style-type: none"> • Run build, but add problem: the dependent build will be run and the problem will be added to it, changing its status to failed (if problem was not muted earlier) • Run build, but do not add problem: the dependent build will be run and no problems will be added • Make build failed to start: the dependent build will not run and will be marked as "Failed to start" • Cancel build: the dependent build will not run and will be marked as "Canceled".

Suitable Builds

A "suitable" build in terms of snapshot dependencies is a build which can be used instead a queued dependency build within a [build chain](#). That is, a queued build which is a part of a build chain can be dropped and the builds depending on it can be made dependent on another queued, running or already finished "suitable" build. This behavior only works when the **Do not run new build if there is a suitable one** option of a corresponding snapshot dependency is selected.

For a build to be considered "suitable", it should comply with all of the conditions below:

- use the same sources snapshot as the entire queued build chain being processed. If the build configurations have the same VCS settings, this basically means the one with the same sources revision. If the VCS settings are different (VCS roots or checkout rules), then "same sources snapshot" revisions means revisions taken simultaneously at some moment in time.
- be successful (if "Only use successful builds from suitable ones" snapshot dependency option is set)
- be a usual, not a [personal build](#)
- have no customized parameters (see also [TW-23700](#))
- have no VCS settings preventing effective revision calculation, see [below](#)
- there is no 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). This also includes no changes to the parameters of all the parent projects of the build configuration. You can check if the settings were changed between several builds by comparing `.teamcity/settings/digest.txt` file in the [hidden build's artifacts](#)
- if there is also an artifact dependency in addition to snapshot one, the suitable build should have artifacts
- all the dependency builds (the builds the current one depends on) are "suitable" and are appropriately merged

Some settings in VCS roots can effectively disable builds reusing. These settings are:

- Subversion: **Checkout, but ignore changes** mode
- CVS: **Checkout by tag** mode
- Perforce: **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

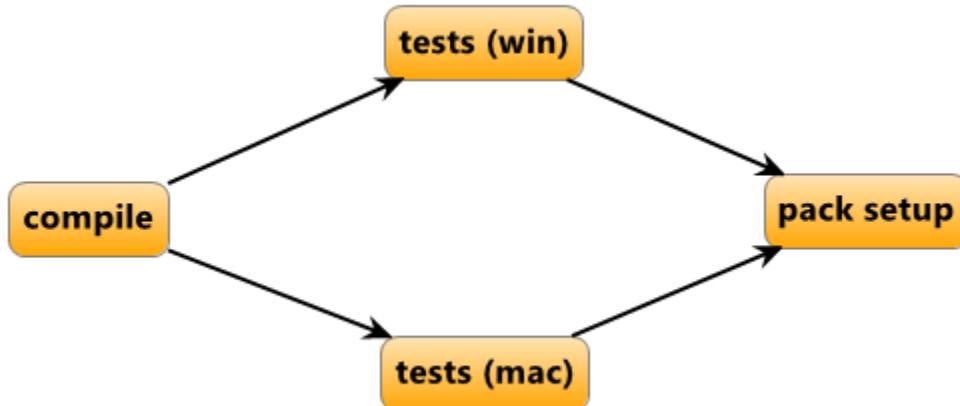
This page is intended to give you the general idea on how dependencies work in TeamCity based on an example. For the dependencies description, please see [Dependent Build](#).

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:

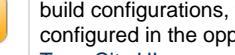
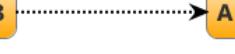
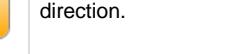
- [Introduction](#)
- [Basics](#)
- [Artifact Dependencies](#)
- [Snapshot Dependencies](#)
 - [When to Create Build Chain](#)
 - [Build Chains in TeamCity UI](#)
 - [How Snapshot Dependencies Work](#)
 - [Example 1](#)
 - [What Happens When Build A is Triggered](#)
 - [What Happens When Build B is Triggered](#)
 - [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.

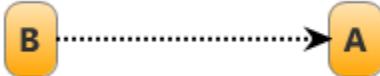
As you noticed, there are 2 types of dependencies in TeamCity: **artifact** dependencies and **snapshot** dependencies. In two words, the first one allows to use the output of one build in another, while the second one can trigger builds from several build configurations in a specific order, but on the 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.

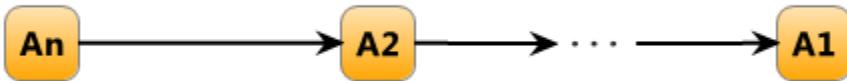


[Learn more](#)

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](#):



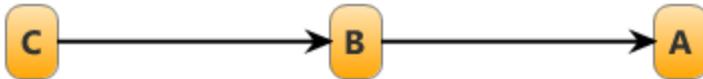
Here are the main rules:

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.

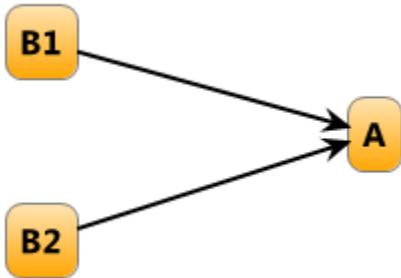
i Note, that if the build configurations connected with a snapshot dependency share the same set of VCS roots, all builds will run on the same sources. Otherwise, if the VCS roots are different, changes in the VCS will correspond to the same moment in time.

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
(mercurial) TW-17252 handle mercurial caches correctly
Pavel Sher (pavel.sher)
(perforce) fix changelog styles
(perforce) disable title + newline
(perforce) fix bug which reproduces after server restart + test

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):

- a specific build (via [Run Custom Build](#) dialog)
- Build Configuration settings (the **Parameters** page of Build Configuration settings) or [Build Configuration Template](#)
- Project settings (the **Parameters** page of the Project settings). These affect all the Build Configurations and Templates of the project and its subprojects.
- an agent (the `<Agent home>/conf/buildAgent.properties` file on the agent)

Any textual setting can reference a parameter which makes the string in the format of `%parameter.name%` be substituted with the actual value at the time of build.

If there is a reference to a parameter which is not defined, it is considered an [implicit agent requirement](#) so the build will only run on the agents with the parameter defined.

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 **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	<code>system.build.number</code> 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.

Password fields can also contain references to parameters, though in this case you cannot see the reference as it is masked as any password value.

Using Build Parameters in VCS Labeling Pattern and Build Number

In Build number pattern and VCS labeling pattern, you can use `%[env|system].property_name%` syntax to reference the properties that are known on the server-side. These are [server](#) and [reference](#) predefined properties and properties defined in the settings of the build configuration on the [Parameters](#) page.

For example, VCS revision number: `%build.vcs.number%`.

Using Build Parameters in the Build Scripts

All build parameters starting with "env." prefix (environment variables) are passed into the build's process environment (omitting the prefix).

All build parameters starting with "system." prefix (system properties) are passed to the supported build script engines and can be referenced there just by the property name (without "system." prefix):

- For Ant, Maven and NAnt use `$(<property name>)`
- For Gradle use `teamcity["<property name>"]`, see a [related example](#). Since **TeamCity 9.1.2** you can also reference them as `System.properties["<property_name>"]`.
- For MSBuild (Visual Studio 2005/2008 Project Files) use `$(<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.files` system property or `TEAMCITY_BUILD_PROPERTIES_FILE` environment variable description in [Predefined Build Parameters#Agent Build Properties](#) for details.

See also:

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

Predefined Build Parameters

TeamCity provides a number of [build parameters](#) which are ready to be used in the settings of a build configuration or in build scripts.

On this page:

- [Server Build Properties](#)
- [Configuration Parameters](#)
 - [Dependencies Properties](#)
 - [Overriding Dependencies Properties](#)
 - [VCS Properties](#)
 - [Branch-Related Parameters](#)
 - [Other Parameters](#)
- [Agent Properties](#)
- [Agent Environment Variables](#)
 - [Java Home Directories](#)
 - [Detecting Java on Agent](#)
 - [Defining Custom directory to Search for Java](#)
 - [Defining Java-related Environment Variables](#)
- [Agent Build Properties](#)

The predefined build parameters can originate from several scopes:

- [#Server Build Properties](#) - the parameters generated by TeamCity on the server-side in the scope of a particular build. An example of such property is a build number.
- [#Agent Properties](#) - the parameters provided by an agent on connection to the server. The parameters are not specific to any build and characterize the agent environment (for example, the path to .Net framework). These are mainly used in [agent requirements](#).
- [#Agent Build Properties](#) - the parameters provided on the agent side in the scope of a particular build right before the build start. For example, a path to a file with a list of changed files.

All these parameters are finally passed to the build.

There is also a special kind of server-side build parameters that can be used in references while defining other parameters, but which are not passed into the build. See [Configuration Parameters](#) below for the list of such properties.



The most up-to-date list of parameters can be obtained in the TeamCity web UI while defining a text value supporting parameters: either click on icon to the right of the text field, or enter "%" in the text field.

Server Build Properties

System properties can be referenced using `%system.propertyName%`.

System Property Name	Environment Variable Name	Description
<code>teamcity.version</code>	TEAMCITY_VERSION	The version of TeamCity server. This property can be used to determine the build is run within TeamCity.
<code>teamcity.projectName</code>	TEAMCITY_PROJECT_NAME	The name of the project the current build belongs to.
<code>teamcity.buildConfName</code>	TEAMCITY_BUILDCONF_NAME	The name of the Build Configuration the current build belongs to.
<code>build.is.personal</code>	BUILD_IS_PERSONAL	Is set to <code>true</code> if the build is a personal one . Is not defined otherwise.
<code>build.number</code>	BUILD_NUMBER	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>	none	The internal unique id used by TeamCity to reference builds.
<code>teamcity.auth.userId</code>	none	A generated username that can be used to download artifacts of other build configurations. Valid only during the build.
<code>teamcity.auth.password</code>	none	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> <pre>teamcity.buildVcsNumberCompatibilityMode=true</pre>
---	---	--

Configuration Parameters

These are the parameters that other properties can reference (only if defined on the [Parameters](#) page), but that are not passed to the build themselves.

You can get the full set of such server properties by adding the `system.teamcity.debug.dump.parameters` property to a build configuration and examining the "Available server properties" section in the build log.

Among these properties are the following:

- Dependencies Properties
 - Overriding Dependencies Properties
- VCS Properties
- Branch-Related Parameters
- Other Parameters
 - Detecting Java on Agent
 - Defining Custom directory to Search for Java
 - Defining Java-related Environment Variables

Dependencies Properties

These are properties provided by the builds the current build depends on (via a snapshot or an artifact [dependency](#)).

In the [dependent build](#), dependencies properties have the following format:

```
dep.<btID>.<property name>
```

- `<btID>` — is the [ID](#) of the build configuration to get the property from. Only the configurations the current one has snapshot or artifact dependencies on are supported. Indirect dependencies configurations are also available (e.g. A depends on B and B depends on C - A will have C's properties available).
- `<property name>` — the name of the [build parameter](#) of the build configuration with the given ID.

Overriding Dependencies Properties

Since TeamCity 9.0, there is a possibility to redefine the [build parameters](#) in the dependency (snapshot-dependency) builds when the current build starts. For example, build configuration A depends on B and B depends on C; A has the ability to change parameters in any of its dependencies using the following format:

```
reverse.dep.<btID>.<property name>
```

It is also possible to change parameter in all dependencies at once:

```
reverse.dep.*.<property name>
```

The `reverse.dep.` parameters are processed on queuing of the build where the parameters are defined. As the parameter's values should be

known at that stage, they can only be defined either as [build configuration parameters](#) or in the [custom build dialog](#). Setting the parameter during the build has no effect.

Pushing a new parameter into the build will supersede the "Do not run new build if there is a suitable one" snapshot dependency option and may trigger a new build if the parameter is set to a non-default value.

Note that the values of the `reverse.dep.` parameters are pushed to the dependency builds "as is", without reference resolution. %-references, if any, will be resolved in the context of the build where the parameters are pushed to.

`<property name>` is the name of the property to set in the noted build configuration. To set system property, `<property name>` should contain "system." prefix.

VCS Properties

These are the settings of VCS roots attached to the build configuration.

VCS properties have the following format:

```
vcsroot.<VCS root ID>.<VCS root property name>
```

- `<VCS root ID>` — is the VCS root ID as described on the [Configuring VCS Roots page](#).
- `<VCS root property name>` — the name of the VCS root property. This is VCS-specific and depends on the VCS support. You can get the available list of properties as described [above](#).

If there is only one VCS root in a build configuration, the `<VCS root ID>.` part can be omitted.

Properties marked by the VCS support as `secure` (for example, passwords) are not available as reference properties.

Branch-Related Parameters

When TeamCity starts a build in a build configuration where [Branch specification](#) is configured, it adds a branch label to each build. This logical branch name is also available as a configuration parameter:

```
teamcity.build.branch
```

To distinguish builds started on a default and a non-default branch, there is an additional boolean configuration parameter available since 7.1.5 which allows differentiating these cases:

```
teamcity.build.branch.is_default=true|false
```

For Git & Mercurial, TeamCity provides additional parameters with the names of VCS branches known at the moment of the build start. Note that these may differ from the logical branch name as per branch specification configured.

This VCS branch is available form a configuration parameter with the following name:

```
teamcity.build.vcs.branch.<VCS root ID>
```

Where `<VCS root ID>` is the VCS root ID as described on the [Configuring VCS Roots page](#).

Other Parameters

Parameter Name	Description
<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, `teamcity.agent.jvm.os.name` or `teamcity.agent.jvm.os.arch`, etc. — these are provided by the JVM running on agent) agents also have properties based on installed applications. TeamCity automatically detects a number of applications including the presence of .NET Framework, Visual Studio and adds the corresponding system properties and environment variables. A complete list of predefined agent-specific properties is provided in the table below.

If additional applications/libraries are available in the environment, the administrator can manually define the property in the `<agent home>/conf/buildAgent.properties` file. These properties can be used for setting various build configuration options, for defining build configuration requirements (for example, existence or absence of some property) and inside build scripts. For more information on how to reference these properties, see the [Defining and Using Build Parameters in Build Configuration](#) page.

In the TeamCity Web UI, the actual properties defined on the agent can be reviewed by going to the **Agents** tab at the top navigation bar|<Agent>|<Agent> page|the **Agent Parameters** tab:

Predefined Property	Description
agent.name	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.
agent.work.dir	The path of Agent Work Directory .
agent.home.dir	The path of Agent Home Directory .
teamcity.agent.jvm.os.name	The corresponding JVM property (see JDK help for properties description)
teamcity.agent.jvm.os.arch	The corresponding JVM property
teamcity.agent.jvm.os.version	The corresponding JVM property
teamcity.agent.jvm.user.country	The corresponding JVM property
teamcity.agent.jvm.user.home	The corresponding JVM property
teamcity.agent.jvm.user.timezone	The corresponding JVM property
teamcity.agent.jvm.user.name	The corresponding JVM property
teamcity.agent.jvm.user.language	The corresponding JVM property
teamcity.agent.jvm.user.variant	The corresponding JVM property
teamcity.agent.jvm.file.encoding	The corresponding JVM property
teamcity.agent.jvm.file.separator	The corresponding JVM property
teamcity.agent.jvm.path.separator	The corresponding JVM property
DotNetFramework<version>[_x86 x64]	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)
DotNetFramework<version>[_x86 x64]_Path	This property value is set to the corresponding framework version(s) path(s)
DotNetFrameworkSDK<version>[_x86 x64]	This property is defined if the corresponding version(s) of .NET Framework SDK is installed. (Supported versions are 1.1, 2.0)
DotNetFrameworkSDK<version>[_x86 x64]_Path	This property value is the path of the corresponding framework SDK version.
WindowsSDK<version>	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)
VS[2003 2005 2008 2010 2012 2013]	This property is defined if the corresponding version(s) of Visual Studio is installed
VS[2003 2005 2008 2010 2012 2013]_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, <code>Visual Studio (sln)</code> .
teamcity.dotnet.msbuild.extensions2.0	The property value is the path to the directory that contains MSBuild 2.0 listener and tasks assemblies.
teamcity.dotnet.msbuild.extensions4.0	The property value is the path to the directory that contains MSBuild 4.0 listener and tasks assemblies.



- Make sure to replace "." with "_" when using properties in MSBuild scripts; e.g. use `teamcity_dotnet_nunitlauncher_msbuild_task` instead of `teamcity.dotnet.nunitlauncher.msbuild.task`
- `_x86` and `_x64` property suffixes are used to designate the specific version of the framework.
- `teamcity.dotnet.nunitlauncher` properties cannot be hidden or disabled.

Agent Environment Variables

An agent can define some environment variables. These variables can be used in build scripts as usual environment variables.

Java Home Directories

When a build agent starts, first the installed JDK and JRE are detected; when they are found, the Java-related environment variables are defined as described in [the section below](#).



The environment variables are defined only if they are not already present in the environment: if a started agent already has the Java-related environment variables set, they are not redefined.

Detecting Java on Agent

The installed Java is searched for in the ALL locations listed below. Then, every discovered Java is launched to verify that it is a valid Java installation, and the Java version and bitness are determined based on the output.

The following locations are searched (a number of locations is common for all operating systems; some of them are OS-specific):

For All OS

- It is checked whether the `JAVA_HOME`, `JDK_HOME`, `JRE_HOME` variables are defined
- The `PATH` environment variables are searched and the discovered directories are checked for containing Java
- If defined, a custom directory on an agent is searched for Java installations. Defining a custom directory to search for Java is described [below](#).

OS-specific locations

Windows

- The Windows Registry is searched for the Java installed with the Java installer
- `C:\Program Files` and `C:\Program Files (x86)` directories are searched for `Java` and `JavaSoft` subdirectories
- the `C:\Java` directory is searched

Unix

The following directories are searched for Java subdirectories:

- `/usr/local/java`
- `"/usr/local`
- `/usr/java`
- `/usr/lib/jvm`
- `/usr`

Mac OS

The following directories are searched:

- `/System/Library/Frameworks/JavaVM.framework/Versions/<Java Version>/Home`
- `/Library/Java/JavaVirtualMachine/Versions/<Java Version>/Home`
- `/Library/Java/Jcd cdavaVirtualMachine/<Java Version>/Contents/Home`

Defining Custom directory to Search for Java

You can define a custom directory on an agent to search for Java installations in by adding the `teamcity.agent.java.search.path` property to the [buildAgent.properties](#) file.

You can define a list of directories separated by an OS-dependent character.

Defining Java-related Environment Variables

For each major version `V` of java, the following variables can be defined:

- `JDK_1V`
- `JDK_1V_x64`
- `JRE_1V`
- `JRE_1V_x64`

The **JDK** variables are defined when the JDK 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 point to both 32-bit or 64-bit installations but 32-bit ones are preferred.

If several installations with the same major version and the same bitness but different minor version/update are found, the latest one is selected.

In addition, the following variables are defined:

- `JAVA_HOME` - for the latest JDK installation (but 32-bit one is preferred)
- `JDK_HOME` - the same as `JAVA_HOME`
- `JRE_HOME` - for the latest JRE or JDK installation (but 32-bit one is preferred), defined even if JDK is found.

The `JRE_HOME` and `JDK_HOME` variables may point to different installations; for example, if JRE 1.7 and JDK 1.6 but no JDK 1.7 installed - `JRE_HOME` will point to JRE 1.7 and `JDK_HOME` will point to JDK 1.6.

All variables point to the java home directories, not to binary files. For example, if you want to execute `javac` version 1.6, you can use the following path:

In a TeamCity build configuration:

```
%env.JDK_16%/bin/javac
```

In a Windows bat/cmd file:

```
%JDK_16%\bin\javac
```

In a unix shell script:

```
$JDK_16/bin/javac
```

Agent Build Properties

These properties are unique for each build: they are calculated on the agent right before build start and are then passed to the build.

System Property Name	Environment Variable Name	Description
<code>teamcity.build.checkoutDir</code>	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 <code>system.*</code> 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](#)
- [Agent Level Build Parameters](#)

Project Level Build Parameters

TeamCity allows you to define build parameters for a project, **all** its subprojects and build configurations in one place: [Project Settings -> Parameters](#) tab.

Note that if a build parameter P is defined in a build configuration and a build parameter with the same name exists on the project level, the

following heuristics applies:

Case 1: Project A, Build Configuration from project A.

Parameters defined in the build configuration have priority over the parameters with the same names defined on project level.

Case 2: Project A, Template T from project A, build configuration from project A inherited from template T.

Parameters of the build configuration have priority over the parameters with the same name defined in project A, and project-level parameters have priority over parameters with the same name defined in the template.

Case 3: Project A1, Project A2, Template T from project A1, build configuration from project A2 inherited from template T.

Parameters of project A2 (the one build configuration belongs to) have priority over the parameters with the same names defined in the template.

You can also define parameters for only those build configurations of the project that use **the same VCS root**. To do that, create a text file named `teamcity.default.properties`, and check it into the VCS root. Ensure that the file appears directly in the build working directory by specifying the appropriate [checkout rules](#). The name and path to the file can be customized via the `teamcity.default.properties` property of a build configuration.

Properties defined this way are not visible in the TeamCity web UI, but are passed directly to the build process.

Agent Level Build Parameters

To define agent-specific properties edit the Build Agent's `buildAgent.properties` file (`<agent home>/conf/buildAgent.properties`). Refer to the [Agent-Specific Properties](#) page for more information.

When defining system properties and environment variables in `teamcity.default.properties` or `buildAgent.properties` file, use the following format:

```
[env|system].<property_name>=<property_value>
```

For example: `env.CATALINA_HOME=C:\tomcat_6.0.13`

See also:

[Administrator's Guide: Configuring Build Parameters](#) | [Defining and Using Build Parameters in Build Configuration](#) | [Predefined Build Parameters](#)

Typed Parameters

When adding a [build parameter](#) (system property, environment variable or configuration parameter), you can extend its definition with a specification that will regulate parameter's control presentation and validation.

This specification is the parameter's "meta" information that is used to display the parameter in the [Run Custom Build](#) dialog. It allows making a custom build run more user-friendly and usable by non-developers.

Consider a simple example. You have a build configuration in which you have a monstrous-looking build parameter that regulates if a build has to include a license or not; can be either true or false; and by default is false. It may be clear for a build engineer, which build parameter regulates license generation and which value it is to have, but it may not be obvious to a regular user.

Using the build parameter's specification you can make your parameters more readable in the [Run Custom Build](#) dialog.

On this page:

- [Adding Parameter Specification](#)
- [Manually Configuring Parameter Specification](#)
- [Copying Parameter Specification](#)
- [Modifying Parameter Specification via REST API](#)

Adding Parameter Specification

To add specification to a build parameter, click the **Edit** button in the **Spec** area when editing/adding a build parameter.

All parameters specifications support a number of common properties, such as:

- **Label:** some text that is shown near the control in the Run Custom Build dialog.
- **Description:** some text that is shown below the control containing an explanatory note of the control use.
- **Display:** If *hidden* is specified, the parameter will not be shown in the **Run Custom Build** dialog, but will be sent to a build; if *prompt* is specified, TeamCity will always require a review of the parameter value when clicking the **Run** button (won't require the parameter if build is triggered automatically); if *normal* is selected, the parameter will be shown as usual.
- **Type :** Currently you can present parameters in following forms:
 - a simple text field with the ability to validate its value using regular expression;
 - a checkbox;
 - a select control;
 - a password field.

The table below provides more details on each control type.

Type	Description
Text	The default. Represents a usual text string without any extra handling
Checkbox	True/false option represented by a check box
Select	"Select one" or "select many" control to set the value to one of predefined settings
Password	This is designed to store passwords or other secure data in TeamCity settings. TeamCity makes the value of the password parameter never appear in the TeamCity Web UI: it affects the settings screens and the Run Custom Build dialog where password fields appear. Also, the value is replaced in the build's Parameters tab and build log. The value is stored scrambled in the configuration files under TeamCity Data Directory. Please note that build log value hiding is implemented with simple search-and-replace, so if you have a trivial password of "123", all occurrences of "123" will be replaced, potentially exposing the password. Setting the parameter to type password does not guarantee that the raw value cannot be retrieved. Any project administrator can retrieve it and also any developer who can change the build script can in theory write malicious code to get the password.

Depending on the specification's type, there are additional settings.

Text	Allowed value - choose the allowed value. For the Regex option, specify Pattern , a Java-style regular expression to validate the field value, as well as a validation message .
Checkbox	Checked value/Unchecked value: Specify values for the parameter to have depending on the checkbox's state.
Select	Check the Allow multiple box to enable multiple selection. In the Items field specify a newline-separated list of items. Use following syntax <code>label => value or value</code> .

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

You can also 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="border: 1px solid #ccc; padding: 10px; margin-top: 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: <code>equals</code>

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](#)

Copy, Move, Delete Build Configuration

To copy, move or delete a build configuration, use the **Actions** menu on the right of the build configuration settings pages.

- [Copy and Move Build Configuration](#)
- [Delete Build Configuration](#)

Copy and Move Build Configuration

Build configurations can be copied and moved to another project by project administrators:

- A copy duplicates all the settings of the original build configuration, but no data related to builds is preserved. The copy is created with empty build history and no statistics. You can copy a build configuration into the same or another project.
- When moving a build configuration between projects, TeamCity preserves its settings and associated data, as well as its build history and dependencies.

On copying, TeamCity automatically assigns a new ID to the copy. It is also possible to change the ID manually.

Selecting the **Copy associated user, agent and other settings** option makes sure that all the settings like notification rules or agent's compatibility are exactly the same for the copied and original build configurations for all the users and agents affected.

If the build configuration uses VCS Roots or is associated with a template which is not accessible in the target project (does not belong to the target project or one of its parent projects), the copies of these VCS roots and the template will be created in the target project. (see also related issue [TW-28550](#))

⚠ When running TeamCity in the [Professional mode](#) with the maximum allowed number of build configurations (20 unless you purchased additional Build Agent licenses), the **Copy** option will not be displayed for build configurations.

Delete Build Configuration

When you delete a build configuration, TeamCity will remove its .xml configuration file. After the deletion, there is a [configurable timeout](#) (24 hours by default) before the builds of the deleted configuration are removed during the build history clean-up.

i If you attempt to delete a build configuration which has [dependent build configurations](#), TeamCity will warn you about it. If you proceed with deletion, the dependencies will no longer function.

Ordering Projects and Build Configurations

By default, TeamCity displays projects, their subprojects, build configurations and templates in the alphabetical order.

Starting from [TeamCity 9.1](#), project administrators can apply custom ordering to subprojects, and build configurations a project on the [Project Settings](#) page for the parent project and use it as the default order.

To enable reordering, use the corresponding button above the list.

Individual users can still manually tweak the display using the up-down button in the [Configure Visible Projects](#) pop-up on the [Projects Overview](#) page.

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 main development and commit all the changes for the feature onto the branch, merging the changes into the main branch when your feature is complete. This approach brings a number of advantages to software development teams; however, in continuous integration servers that do not have dedicated support for it, it also causes a number of problems, like constant build configurations duplication, poor visibility, and, in the end, loss of control over the process.

TeamCity support for feature branches is continuously increasing, 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](#). TeamCity 8.1 introduced [Automatic Merge](#) functionality to merge a branch into another after a successful build.

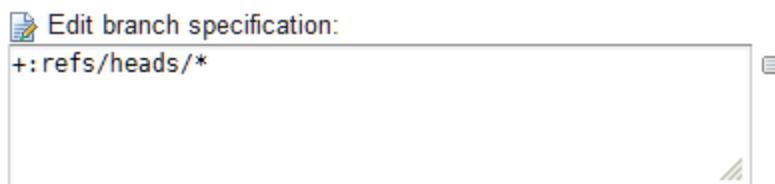
- Configuring branches
- Default branch
- Logical branch name
- Builds
- Changes
- Active branches
- Tests
- Failure Conditions
- Triggers
- Dependencies
- Notifications
- Build configuration status
- Multiple VCS roots
- Build parameters
- Clean-up
- Manual branch merging

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](#):

Branch Specification:



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.

The use of * wildcard matches branches and extracts the branch name to be displayed in the TeamCity interface (also known as [logical branch name](#)):

+:`refs/heads/*` will match `refs/heads/feature1` branch, but in the TeamCity interface you'll see `feature1` only as a branch name.

To see an extended logical name in the TeamCity interface, use the part of the name to be displayed in round brackets:

+:`refs/heads/(v8.1/*)` and in the interface you'll see `v8.1/feature1`

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 round 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. A logical branch name can be a part of the full VCS branch name. It is calculated by applying a [branch specification](#) to the branch name from the version control. For example, if the branch specification is defined like this:

```
+:refs/heads/*
```

then the part matched by * (e.g. `master`) is a logical branch name. For the default branch <default> can be used.

If the branch specification pattern uses parentheses, the logical name then is made up of the part of the name within the parentheses; e.g. the branch specification +:`refs/heads/(v8.1/*)` will give you the `v8.1/feature1` logical name for the VCS branch `refs/heads/v8.1/feature1`.

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 Bankev... (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 number of parameters define whether a branch is active. The parameters can be changed either in a build configuration (this will affect one build configuration only), project, or in the [internal properties](#) (this defines defaults for the entire server). A parameter in the configuration overrides a parameter in the [internal properties](#).

A branch is considered active if:

- it is present in the VCS repository and has recent commits (i.e. commits with the age less than the value of `teamcity.activeVcsBranch.age.days` parameter, 7 days by default).
- or it has recent builds (i.e. builds with the age less than the value of `teamcity.activeBuildBranch.age.hours` parameter, 24 hours by default).

A closed VCS branch with builds will still be displayed as active during 24 hours after last build. To remove closed branches from display, set `teamcity.activeBuildBranch.age.hours=0`.

Tests

TeamCity tries to detect new failing tests in a build, and for those tests which are not new, you can see in which build the test started to fail. This functionality is aware of branches too, i.e. when the first build is calculated, TeamCity traverses builds from the same branch.

Additionally, a [branch filter](#) is available on the test details page and you can see a history of test passes or failures in a single branch.

Failure Conditions

If build fail condition is configured as follows: build metric has changed comparing to a last successful/finished/pinned build, then the build from the same branch will be used. If there is no suitable build on the same branch, then build from default branch is used and the corresponding message is added to the build log.

Triggers

The VCS trigger is fully aware of branches and will trigger a build once a check-in is detected in a branch. All VCS trigger options like per-checkin triggering, quiet period, and triggering rules are directly available for builds from branches. By default, the Schedule and Finish build trigger will watch for builds in the default branch only.

Additionally, a [branch filter](#) can be specified for the VCS, Schedule and Finish build triggers.

Dependencies

If a build configuration with branches has snapshot dependencies on other build configurations, when a build in a branch is triggered, the other builds in the chain also get the branch associated.

In particular:

- all the builds down the chain (which the build depends on) which have the branch in their VCS root (not excluded by the branch specification) will be marked with the branch;
- all the builds up the chain (depending on the build) will be marked with the same branch. If their VCS roots have the branch and it is not excluded by their branch specification, the branch will be checked out. Otherwise the default branch will be checked out.

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](#).

Manual branch merging

You can merge branches in TeamCity manually, e.g. if you want to merge branches only after a code review / approval, or if you want to perform the merge despite the tests failure in a branch.

To merge sources manually:

Open the [build results page](#), click the **Actions** menu in the top-right corner and select "**Merge this build sources...**". The dialog that appears enables you to select the destination branch and add a commit message (required).

It is also possible to merge branches [automatically](#).

See also:

[Administrator's Guide: Git | Mercurial](#)

Triggering a Custom Build

A build configuration usually has [build triggers](#) configured which automatically start a new build each time the conditions are met, like scheduled time, or detection of VCS changes, etc.

Besides triggering a build automatically, TeamCity allows you to run a build manually whenever you need, and customize this build by adding properties, using specific changes, running the build on a specific agent, etc.

On this Page:

- Run Custom Build dialog
 - General Options
 - Dependencies
 - Changes
 - Include changes
 - Build branch
 - Use settings
 - Parameters
 - Comment and Tags
- Promoting Build

There are several ways of launching a custom build in TeamCity:

- Click the ellipsis on the **Run** button, and specify the options in the **Run Custom Build** dialog described [below](#).
- To run a custom build with specific changes, open the build results page, go to the **Changes** tab, expand the required change, click the **Run build with this change** and proceed with the [options](#) in the **Run Custom Build** dialog.
- Use [HTTP request](#) or [REST API request](#) to TeamCity to trigger a build.
- Promote a build - see the section [below](#).

Run Custom Build dialog

General Options

Select an agent you want to run the build on from the drop-down list. Note that for each agent in the list, TeamCity displays its current state and estimates when the agent will become idle if it is running a build at the moment. Besides the possibility to run a build on a particular agent from the list, you can also use one of the following options:

- **fastest idle agent** — *default option*; if selected, TeamCity will automatically choose an agent to run a build on based on calculated estimates.
- **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 will be taken. By default, the last 20 builds are displayed. To increase the number of builds displayed in the drop-down to 50, use the `teamcity.runCustomBuild.buildsLimit=50` [internal property](#).

Changes

This tab is available only if you have permissions to access VCS roots for the build configuration.

The tab allows you to specify a particular change to be included to the build.

The build will use the change's revision to checkout the sources. That is, all the changes up to the selected one will be included into the build. Note that TeamCity displays only the changes detected earlier for the current build configuration VCS roots. If the VCS root was detached from the build configuration after the change occurred, there is no ability to run the build on such a change. A limited number of changes is displayed. If there is an earlier change in TeamCity that you need to run a build on, you can locate the change in the Change Log and use the **Run build with this change** action.

Include changes

The **Include changes** drop-down allows selecting the changes in the VCS roots attached to the configuration to run the build on.

- **Latest changes at the moment the build is started:** TeamCity will automatically include all changes available at the moment.
- <Last change to include>: When you select a change in the drop-down list, TeamCity runs the build with the selected change and all changes that were made before it. The build run with the changes earlier than the latest available is marked as a [History Build](#).

Build branch

The **Build branch** drop-down, available if you have branches in your build configuration (or in snapshot dependencies of this build configuration), allows choosing a branch to be used for the custom build.

Use settings

Since TeamCity 9.1, if your project has [versioned settings](#) enabled, you can tell TeamCity to run a build:

- with the settings defined for the project, either the current settings on the server or the settings from VCS (since TeamCity 9.1.2)
- with the project settings currently defined on the server
- with the settings loaded from the VCS revision calculated for the build.

If changes are selected in the [step above](#), the revision of the project settings corresponding to the selected changes will be loaded.

To define which settings to take, use one of the corresponding options from the **Use settings** drop-down (the option here will override the [project-level setting](#)).

Parameters

These settings are available only if you have permissions to change system properties and environment variables for the build configuration. This tab allows adding new parameters/properties/variables, and editing or deleting them, as well as redefining values of the [predefined ones](#).

When adding/editing/deleting properties and variables, note the following:

- For a predefined property/variable, only the value is editable.
- Only newly added properties/variables can be deleted. You cannot delete predefined properties.

Comment and Tags

Add an optional comment as well as one or more [tags](#) to the build.

Since TeamCity 9.1, you can add a custom build to [favorites](#) by checking the corresponding box in this section.



A greater build number does not mean more recent changes and the last build in the builds history does not reflect the state of the latest project sources: builds in the builds history are sorted by their start time, not by changes they include.

Promoting Build

Build promotion refers to triggering a custom build with an overridden [artifact](#) or [snapshot dependency](#), i.e. manual launching of a build with dependencies configured, but using a build different from the build specified in the dependency.

To promote a build, open the build results page of the dependency build and click **Actions | Promote**.

For example, your build configuration A is configured to take artifacts from the last successful build of configuration B, but you want to run a build of configuration A using artifacts of a different build of configuration B (not the last successful build), so you promote an earlier build of B.

Build promotion affects only a single run of the dependent build. Once you click **Promote**, a build of the dependent build configuration which uses the artifacts of the specified build is queued. Any further runs of the dependent build configuration will use artifacts as configured (last successful, last pinned etc.), unless you use another promotion.

More details are available in the [related blog-post](#).

See also:

Concepts: [Build Queue](#) | [Dependent Build](#) | [Personal Build](#)
Administrator's Guide: [Configuring Build Triggers](#)

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.

TeamCity optimizes the [queue](#) when adding builds; however, it is possible to manage the queue builds manually.

On this page:

- [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.

This feature is useful when some tests fail for some known reason, but it is currently not possible to fix them. For example, the responsible developer is on vacation, or you are waiting for the system administrators to fix the environment, or the test is failing intentionally, for example, if the required functionality is not yet written (TDD). In these cases you can mute such failures and avoid unnecessary disturbance of other developers.

When a test is muted, it is **still run** in the future builds, but its failure does not fail the build (by "at least one test failed" **build failure condition**). The test can be unmuted manually on a specific date or after a successful run. Also, tests can be muted only in a single build configuration or in all the build configurations of a specific TeamCity project.

Your build script might need adjustment to make the build green when there are failing but muted tests. Make sure that the build does not fail because of other build failure conditions (e.g. "Fail if build process exit code is not zero") in case the only errors encountered were tests failures. See also the related issue [TW-16784](#).



#6.0.1717.0 ● [Tests passed: 2075, ignored: 63, muted: 3](#)

Win32 Notify | Last built: 9 minutes ago

[View build results](#)

How to mute tests

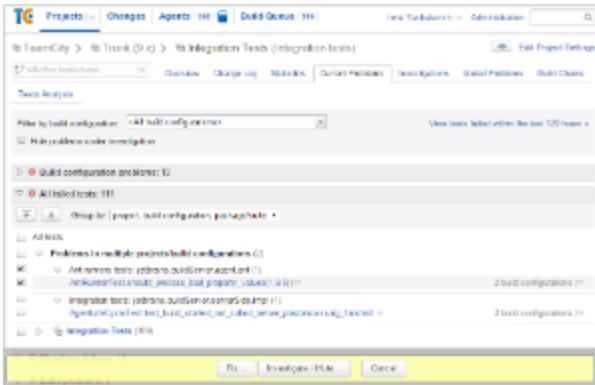
The **Mute/unmute problems in project** permission required to mute tests is granted to **Project administrator** and **System administrator** by default.

You can mute a test failure from:

- The **Projects** page
- The Project overview page
- The Build Configuration home page
- The Current Problems tab



On the build results page you can select several test failures (or all) to be muted or unmuted:



Note that you can start investigation of the problem simultaneously with muting the failure. When muting a test failure, you can specify conditions when it should be unmuted: on a specified date or when it is fixed. Alternatively, you can unmute it manually.

- On the build results page you can view the list of muted test failures, their stacktraces and details about the mute status:



- 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.

Changing Build Status Manually

Overview

A user with appropriate permissions can change the status of a build manually, i.e. make it either failed or successful (issue [TW-2529](#)).

The corresponding action is available in the Actions menu on the [build results](#) page.

Marking build as successful

You may want to make build successful to:

- Change the **last successful build** anchor when using [Build failure conditions](#), i.e. if your last build failed because of an incorrect value of a metric, and this new value is valid, you may mark this build with a successful anchor.
- Allow using an incorrectly failed build with good artifacts in the "[last successful](#)" dependencies.
- For a running [personal build](#), you can mark the current failures as non-relevant to allow pre-tested commit to pass (if the user has permission to do this).

The "Mark as successful" action is not available for [Failed to Start Builds](#).

Marking build as failed

You may want to mark a build as failed when:

- The build has some problem which didn't affect the final [build](#) status.
- There is a known problem with the build, and it should be ignored by your QA team.
- You've mistakenly marked the build as successful manually.

Permissions

By default, the permission to change the build status is granted to **Project Administrator**.

Customizing Statistics Charts

To help you track the condition of your projects and individual build configurations over time, TeamCity gathers statistical data across all their history and displays it as visual charts. 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/projects/<project_name>/pluginData/plugin-settings.xml` file. In this file a similar format is used for all types of pre-defined graphs:

Chart Type	XML Tag Name
Code Coverage	coverage-graph
Code Duplicates (Java and .NET)	duplicates-graph
Code Inspections	inspections-graph

Disabling Charts of Particular Type on Project Level

To disable charts of particular type for a project, use the following syntax:

```
<coverage-graph enabled="false"/>
```

In this example, all code coverage charts will be removed from the Statistics page.

Showing Charts Only for Specific Build Configurations on Project Level

To show the code coverage chart related only to a particular build configuration, use the following syntax:

```
<coverage-graph enabled="true">
    <build-type id="myConf1"/>
    <build-type id="myConf2"/>
</coverage-graph>
```

where **myConf1** and **myConf2** values are [build configuration IDs](#).

However, note that build configurations specified should contain code coverage data for the charts to be shown. If the data is available, two charts will be shown (one for each specified build configuration).

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](#)

Storing Project Settings in Version Control

Since TeamCity 9.0, TeamCity allows storing the project configuration settings in a **Git** or **Mercurial** version control repository.
Since TeamCity 9.1, **Perforce** and **Subversion** version controls are also supported.

On this page:

- Synchronizing Settings with VCS
 - Defining Settings to Apply to Builds
 - Forcing Synchronization
 - Displaying Changes
- Security Implications

- Known Limitations

Synchronizing Settings with VCS

By default, the synchronization of the project settings with the version control is disabled; a user with "Enable/disable versioned settings" permission (having role "System administrator" by default) can enable the two-way synchronization on the **Versioned Settings** page under the project settings in **Administration**.

The **Configuration** tab is used to define whether:

- the synchronization settings are the same as in the parent project
- the synchronization is enabled.
 - when synchronization is enabled, you can define which settings to use when build starts. See details below.

When synchronization is enabled:

- each administrative change made to the project settings in the TeamCity Web UI is committed to the version control; the changes are made noting the TeamCity user as the committer;
- if the settings are altered in the version control, the TeamCity server will detect the modifications and apply them to the project on the fly.

The settings in the VCS are stored in the same format as in the [TeamCity Data Directory](#). The format of the settings differs from one TeamCity version to another.

Enabling synchronization for a project also enables it for all its subprojects. TeamCity synchronizes all changes to the project settings (including modifications of [build configurations](#), [templates](#), [VCS roots](#), etc.) with the exception of [SSH keys](#).

 You can override the synchronization settings inherited from a project on the **Versioned Settings| Configuration** page of a subproject.

As soon as synchronization is enabled in a project, TeamCity will make an initial commit in the selected repository for the whole project tree (the project with all its subprojects) to store the current settings from the server.

If the settings for the given project are found in the specified VCS root (the VCS root for the parent project settings or the user-selected VCS root), a warning will be displayed asking if TeamCity should:

- **overwrite the settings in VCS** with the current project settings on the TeamCity server
- **import the settings from VCS** replacing the the current project settings on the TeamCity server with those from version control

The settings are stored in the `.teamcity` folder in the root of the repository; the default branch is used with [Git](#) and [Mercurial](#).

 For Perforce and Subversion, if you wish to change the path used by TeamCity, you can create a special VCS Root dedicated to the VCS settings storage, and specify the path as you want there. TeamCity will use the `.teamcity` directory relative to the configured client.

When the settings are changed via the UI, TeamCity will wait for the changes to be completed with a commit to the VCS before running a build with the latest changes.

If the settings are changed via the user interface, a commit in the VCS will be performed on behalf of the user specified in the VCS root. For Perforce and Subversion the commit message will also contain the username of the TeamCity user who actually made the change via the UI.

Defining Settings to Apply to Builds

Since **TeamCity 9.1**, you can start builds with settings different from those currently defined in the build configuration. For projects where versioned settings are enabled, you can tell TeamCity which settings to take **when build starts**.

This gives you several possibilities:

- if you're starting a [history build](#), TeamCity will use the settings corresponding to the moment of the selected change
- if you are using [TeamCity feature branches](#), you can define a branch specification in the VCS root used for versioned settings, and TeamCity will run a build in a branch using the settings from this branch
- you can now start a [personal build](#) with changes made in the `.teamcity` directory, and these changes will affect the build behavior.

Before starting a build, TeamCity stores configuration for this build in build internal artifacts under `.teamcity/settings` directory. These configuration files can be examined later to understand what settings were actually used by the build.

To define which settings to take **when build starts**, select one of the following options:

- **always use current settings**: when this option is set, builds use current project settings from the TeamCity server. Settings changes in branches, history and personal builds are ignored. Users cannot run a build with custom project settings.
- **use current settings by default**: when this option is set, a build uses the latest project settings from the TeamCity server. Users can run a build with older project settings via the [custom build dialog](#).
- **use project settings from VCS**: when this option is set, a build loads settings from the versioned settings revision calculated for the build. This includes builds in branches and history builds, which use settings from VCS. Users can change configuration settings in [personal builds from IDE](#) or can run a build with project settings current on the TeamCity server via the [custom build dialog](#).

There are a few **limitations** when using settings from the VCS:

- changes in snapshot dependencies will be ignored, TeamCity will continue reading snapshot dependencies settings from the build configuration
- changes in build failure conditions and build features working on the server (like automatic merge and labeling) are ignored too
- changing some of the settings does not make much sense for build, for instance, build triggers, general settings like limitation on a number of concurrently running builds, and some others.

Forcing Synchronization

To force synchronization of the current project settings, use the **Commit current project settings...** option on the **Versioned Settings| Configuration** page.

Displaying Changes

TeamCity will not only synchronize the settings, but will also automatically display changes to the project settings the same way it is done for regular changes in the version control. You can configure the changes to be displayed for the affected build configurations. Such changes are ignored by build triggers.

All changes in the VCS root where project settings are stored are listed on the **Versioned Settings| Change log** tab of the **Versioned Settings** page.

Security Implications

Enabling storing settings in VCS has some security implications and it is recommended carefully consider those before deciding on the scope of the feature use.

In particular:

- if the **Show settings changes in builds** option is enabled, any user who can see the content of the build's file changes in the TeamCity web UI (having the "View VCS file content" permission) can see all the project settings, including the values of the password fields (in scrambled form)
- if the projects or build configurations with settings in VCS have password fields defined, the values appear in the XML settings committed into the VCS (though, in scrambled form)
- being able to change the settings in arbitrary manner via VCS, if is possible to trigger builds of any build configurations and obtain settings of any build configurations irrespective of the build configurations permissions configured in TeamCity
- by committing wrong or malicious XML settings, user can affect the entire server performance or due server presentation to other users

Known Limitations

- The supported version controls are [Git](#) and [Mercurial](#), and **since TeamCity 9.1**, [Perforce](#) and [Subversion](#).
- When running a [history build](#) in TeamCity, the current project settings will be used. **Since TeamCity 9.1**, TeamCity will attempt to use the settings corresponding to the moment of the selected change. For details, see the [section above](#).
-  Project settings may contain scrambled passwords. Once you enable settings synchronization, everyone who has a developer role in the project will be able to see these passwords using the [changes difference viewer](#).

Managing Licenses

In this section:

- [Licensing Policy](#)
- [Third-Party License Agreements](#)

Licensing Policy

This page covers:

- [Licensing Overview](#)
- [Editions](#)
 - Number of Build Configurations
 - Number of Agents
- [Managing Licenses](#)
- [Valid TeamCity Versions](#)
- [License Expiration](#)
- [Ways to Obtain a License](#)
- [Upgrading From Previous Versions](#)
 - Upgrading from TeamCity 5.x and later
 - Upgrading from TeamCity 4.x to TeamCity 5.0 and later
 - Upgrading from TeamCity 3.x to TeamCity 4.0
 - Upgrading from TeamCity 1.x-2.x to TeamCity 4.0
 - Upgrading with IntelliJ IDEA 6.0 License Key

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](#).

Licensing Overview

JetBrains offers several licensing options that allow you to scale TeamCity to your needs.

This section illustrates the main differences between the TeamCity server [editions](#) and provides general information on the TeamCity [Build Agent](#) license.

For detailed information, refer to the sections below.

Professional Server	Enterprise Server
no license key is required, free	a license key is required, price options
20 build configurations	unlimited number of build configurations
full access to all product features	free 1-year subscription to upgrades
support via community forum	priority email support
3 build agents included, buy more as necessary	from 3 to 100 build agents included, buy more as necessary

If you need more build agents that are included with your TeamCity server edition, you can purchase additional build agent licenses.

Build Agent License
connects 1 additional build agent
if using Professional edition, adds 10 additional build configurations
a license key is required, price options

Editions

There are two editions of TeamCity: **Professional** and **Enterprise**.

The editions are equal in all the features except for the maximum number of build configurations allowed.

The same TeamCity distribution and installation 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.

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 or more [build agents](#). Each additional **build agent** above the bundled ones requires a new build agent license key in both editions.

Besides the Professional and Enterprise licenses, there are two more license types:

- **Evaluation** — has an expiration date and provides an unlimited number of agents and build configurations. To obtain the evaluation license, 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.

Number of Build Configurations

The Enterprise edition has no limit on the number of build configurations.

The Professional edition allows 20 [build configurations](#) per server. Since TeamCity 8.0, each build agent license key gives you 10 more build configurations in Professional edition in addition to one more agent. All build configurations are counted (i.e. including those in archived projects).

Number of Agents

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. The licenses are not bound to specific server instance, machine, etc. The only limitation is that a license cannot be used on several servers at the same time.

When you already own license(s) and buy more licenses, you can [request](#) JetBrains sales to make the new licenses co-termed with those already purchased, so that all the licenses have equal maintenance expiration date. The cost of the licenses is then lowered proportionally.

When buying many licenses you are welcome to [contact](#) our sales for available volume discounts.

Valid TeamCity Versions

TeamCity licenses are perpetual for the TeamCity versions they cover. This means that you can run a covered TeamCity version with existing licenses for unlimited time and the licenses will stay valid for this TeamCity version.

Each TeamCity license (including Enterprise Server and Agent) has a **maintenance period** (generally 1 year). The license key is valid with any TeamCity version released within the maintenance period. Licenses valid for the major/minor release (changes in the first two release numbers) is also considered valid for the corresponding bugfix updates (changes in the third release number).

Before you [upgrade](#) to a newer TeamCity version, please check the validity of the existing licenses with the new version.

If the new TeamCity server effective [release date](#) is not covered by the maintenance period of some of the licenses, the corresponding licenses will not be valid with the TeamCity version and would need an [upgrade](#).

When a new version is available, TeamCity displays a notification in the web UI and warns you if any of your license keys are incompatible with this new version. A notification on the new TeamCity version is also displayed in the Global Configuration Items of the [Server Health](#) report, visible to system administrators. System administrators can use the link in the "Some Licenses are incompatible" message to quickly navigate to the [Licenses](#) page, where all incompatible licenses will have a warning icon. The information about the license keys installed on your server is secure as it is not sent over the Internet.

Regular upgrades are recommended as new releases contain lots of fixes (and of course new features).

Please note that TeamCity [email support](#) covers only the recent TeamCity versions and can be provided only to customers with not expired maintenance period of the enterprise server license.

License Expiration

If an Enterprise license key is removed from the server, or an evaluation license expires, or a TeamCity server is upgraded to a version released out of the maintenance window of the available Enterprise license, TeamCity automatically switches to the Professional mode.

If the number of build configurations or the number of authorized agents 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.

Build Agent Licenses work the same way as the Server Licenses. If you upgrade the server to the version which is not covered by the agent license maintenance window, then this agent license will expire.

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
ApacheDS	Apache
Apache HttpComponents	Apache
Apache Ivy	Apache
Apache Jakarta Project	Apache
Apache Log4j 2	Apache
Apache log4net	Apache
Apache Lucene	Apache
Apache Tomcat	Apache
Apache Xerces2 Java Parser	Apache
Atmosphere	Apache, CDDL
AWS SDK for Java	Apache 2
Behaviour	BSD
Byteman	GNU LGPL
CassiniDev	Ms-PL

CodeMirror	MIT-style
Core4j	Apache 2.0
cgleb	Apache
CVS client library	CDDL
CyberNekoHTML Parser	Apache-style
DHTML Tip Message	
EhCache	Apache
Expat XML Parser Toolkit	MIT
Flot	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
Jackson	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 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	GPL
Logstash Log4J Layout	Apache 2.0
Managed Stack Explorer	Ms-PL

Maragogype	Apache
Maven	Apache
Metrics	CPL-1
Microsoft.Web.Infrastructure	MVC-3-EULA
Missing Link Ant Tasks	Apache
Mono.Cecil	MIT/X11
NanoContainer	NanoContainer
NUnit	NUnit
OData4j	Apache 2.0
opencsv	Apache
pack:tag	GPL
Paul Johnson's MD5	BSD
Perf4J	Apache 2
PicoContainer	PicoContainer
PocketHTTP	Mozilla
PocketXML-RPC	Mozilla
PostgreSQL Data Base Management System	PostgreSQL License
Prototype	MIT
pty4j	EPL
Raphael	MIT
Rome	Apache
RouteMagic	Ms-PL
Script.aculo.us	MIT License
SilverStripe Unobtrusive Javascript Tree Control	BSD
Shaj	Apache
Slf4j	MIT
Smack	Apache
Spring Framework	Apache
Code snippets from Subclipse	EPL
SVNKit	TMate Open Source
typica	Apache
Tom Wu's jsbn	BSD
Trove High Performance Collections for Java	GNU LGPL
Underscore.js	MIT
UserAgentUtils, version 1.12	user-agent-utils
Waffle	EPL
WebActivator	MS-PL
Winp, version 1.7 (patched)	MIT
XML Pull Parser	Extreme! Lab, Indiana University License

XML-RPC.NET	MIT X11
XStream	XStream
XZ	XZ license
YUI Compressor	BSD

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.

UserAgentUtils Copyright © 2013, Harald Walker (bitwalker.eu) All Rights Reserved.

XML Pull Parser: Copyright © 2002 Extreme! Lab, Indiana University. All rights reserved. This product includes software developed by the Indiana University Extreme! Lab (<http://www.extreme.indiana.edu/>).

XML-RPC.NET Copyright © 2006 Charles Cook.

XStream Copyright © 2003-2006, Joe Walnes, Copyright © 2006-2007, 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):

A screenshot of the TeamCity project home page for '10 Goya 8.0.x'. A tooltip is displayed over a link labeled 'View changes in external viewer' (highlighted by a red box). The tooltip contains the text: 'Open in browser' and 'View changes in external viewer'.

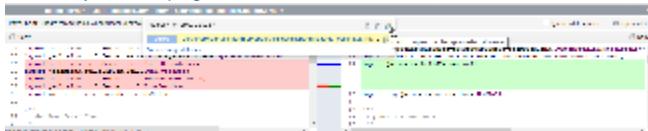
- the **Changes** tab of the build results page:

A screenshot of the TeamCity build results page for build #27557. The 'Changes' tab is selected. It shows a table of changes with columns: 'VCS Root', 'Revision', and 'Label'. Several changes are listed, each with a link labeled 'View changes in external viewer' (highlighted by a red box). Below the table, there are search and filter options.

- 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

TeamCity can be integrated with your issue tracker to provide you a comprehensive view of your development project. TeamCity detects issues mentioned in the comments to version control changes, turning them into links to your issue tracker in the TeamCity Web UI.

Since TeamCity 9.0, the integration has moved from the server to the project level: the Project Administrator permissions are required. You can configure integration if you have multiple projects on both the TeamCity and the issue tracker server or if you are using different issue-trackers for different projects.

Enabling integration for the project also enables it for all its subprojects; if the configuration settings are different in a subproject, its settings have priority over the project's settings.

In this section:

- [Dedicated Support for Issue Trackers](#)
- [Recommendations on Using Issue Tracker Integration](#)
- [Enabling Issue Tracker Integration](#)
- [Integrating TeamCity with Other Issue Trackers](#)

Dedicated Support for Issue Trackers

TeamCity supports [Jira](#), [Bugzilla](#) and [YouTrack](#) issue trackers out of the box. The [Supported Platforms and Environments](#) page lists supported versions.

When integration is configured, TeamCity automatically transforms an issue ID (=issue key in JIRA) mentioned in the VCS commit comment into a link to the corresponding issue and the basic issue details are displayed in the TeamCity Web UI when hovering over the icon next to the issue ID (e.g. on the [Changes](#) tab of the build results).



Issues fixed in the build can also be viewed on the [Issues](#) tab of the build results.

On the build configuration home page, you can review all the issues mapped to the comments at the [Issue Log](#) tab. You can filter the list to a particular range of builds and view issues mentioned in comments with their states.



Recommendations on Using Issue Tracker Integration

To get maximum benefit from the issue tracker integration, do the following:

- When committing changes to your version control, **always mention the issue id (issue key)** related to the fix in the comment to the commit.
- Resolve issues when they are fixed (the time of resolve does not really matter).
- Use **Issue Log** of a build configuration to get issues related to builds; turn on the "Show only resolved issues" option to only display the issues fixed in the builds.

Enabling Issue Tracker Integration

To enable integration, you need to create a connection to your issue tracker on the **Project Settings | Issue Trackers** page (**since TeamCity 9.0**). For earlier versions, use the **Administration | Issue Tracker** page.

The settings described below are common for all issue trackers:

Connection type	Select the type of your issue tracker from the list.
Display name	A symbolic name that will be displayed in the TeamCity UI for the issue tracker.
Server URL	The Issue tracker URL
Username/Password	Credentials to log in to the issue tracker, if it requires authorization.

In addition to these general settings, you also need to specify which strings are to be recognized by TeamCity and converted to the links to issues in your issue tracker. For details, 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:

- The TeamCity server needs direct access to the issue tracker. (Also, TeamCity does not yet support proxy for connections to issue trackers).
- The user configured in the connection to the issue tracker has to have sufficient permissions to view the issues that can be mentioned in TeamCity. Also, TeamCity users will be able to view the issue details in TeamCity for all the issues that the configured user has access to.

Integrating TeamCity with Other Issue Trackers

To integrate TeamCity with an issue tracker other than Jira/YouTrack/Bugzilla, 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 are to be recognized as references to issues in your tracker.

For Bugzilla, you need to specify the **Issue Id Pattern**: a Java Regular Expression pattern to find the issue ID in the text. The matched text (or the first group if there are groups defined) is used as the issue number. The most common case seems to be `#(\d+)` - this will extract 1234 as issue ID from text "Fix for #1234".

Requirements

If the username and password are specified, you need to have Bugzilla XML-RPC interface switched on. This is not required if you use

anonymous access to Bugzilla without the username and password.

Known Issues

There are several known issues in Bugzilla regarding XMLs generated for the issues, which makes it hard to communicate with it. However this can usually be fixed by tweaking the Bugzilla configuration.

- If you see a *path/to/bugzilla.dtd not found* error, this means that the issue XML contains the relative path to the `bugzilla.dtd` file, and not the URL. To fix that, set the server URL in Bugzilla.
- Sometimes you may see a `SAXParseException` saying that *Open quote is expected for attribute "type_id" associated with an element type "flag"*. This happens because the generated XML does not correspond to the bundled `bugzilla.dtd`. To fix it, make the `type_id` attribute `#IMPLIED` (optional) in the `bugzilla.dtd` file. The issue and the workaround are described in detail [here](#).

See also:

[Concepts: Supported Issue Trackers](#)

[Administrator's Guide: Integrating TeamCity with Issue Tracker](#)

JIRA

Converting Strings into Links to Issues

When [enabling issue tracker integration](#) in addition to general settings, you need to specify which strings should be recognized as references to issues in your tracker.

For JIRA, you need to provide a space-separated list of [Project keys](#).

For example, if a project key is **WEB**, an issue key like **WEB-101** mentioned in a VCS comment will be resolved to a link to the corresponding issue.

See also:

[Concepts: Supported Issue Trackers](#)

[Administrator's Guide: Integrating TeamCity with Issue Tracker](#)

YouTrack

Converting Strings into Links to Issues

When [enabling issue tracker integration](#) in addition to general settings, you need to specify which are to be recognized as references to issues in your tracker.

For YouTrack, you need to provide a space-separated list of [Project Ids](#). For example, if a project id is **TW**, an issue id like **TW-18802** mentioned in a VCS comment will be resolved to a link to the corresponding issue.

Enhancing Integration with YouTrack

YouTrack provides native TeamCity integration which enhances the set of available features. For example:

- YouTrack is able to fill "Fixed in build" field with a specific build number.
- YouTrack allows you to apply commands to issues by specifying them in a comment to a VCS change commit.

To use these features, [configure YouTrack](#).

See also:

[Concepts: Supported Issue Trackers](#)

[Administrator's Guide: Integrating TeamCity with Issue Tracker](#)

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.

Since TeamCity 9.1, multiple usernames are supported for a VCS root type and for a separate VCS root: several newline-separated values can be used for each VCS username.

The names set here will be used to:

- show builds with changes committed by a user with such VCS username on the [Changes](#) page
- highlight such builds on the [Projects](#) page if the appropriate [option is selected](#),
- notify the user on such builds when the **Builds affected by my changes** option is selected in [notifications settings](#).

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|Authentication](#) 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.
- 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 be notified about](#). The notification messages can be customized globally on a per-server basis.

Notifications can also be received via Atom/RSS syndication feeds, but since the feeds use the "pull" model for receiving notifications instead of "push", some of the approaches are different for the feeds.

On this page:

- [Notifications Lifecycle](#)
- [Customizing Notifications Templates](#)
 - [Notification Templates Location](#)
 - [Supported Output Values](#)
 - [Customization Examples](#)
 - Including `ERRORS` from the log
 - Listing build artifacts
 - Listing build parameters
 - [Default Data Model](#)
 - [TeamCity Notification Properties](#)
 - [Syndication Feed Template](#)

Notifications Lifecycle

TeamCity supports a set of events that can generate user notifications (such as build failures, investigation state changes, etc). On an event occurrence, for each notifier type, TeamCity processes notification settings for all the users to determine which users to notify.

When the set of users is determined, TeamCity fills the notification model (the objects relevant to the notification, such as as "build", investigation data, etc.) and evaluates a notification template that corresponds to the notification event.

The template uses the data model objects to generate the output values (e.g. notification message text). The output values are then used by the notifier to send the message. Each notifier supports a specific set of the output values.

Please note that the template is evaluated once for an event which means that notification properties cannot be adjusted on a per-user basis.

The output values defined by the template are then used by the notifier to send alerts to the selected users.

Customizing Notifications Templates

Notification Templates Location

Each of the bundled notifier has a directory under `<TeamCity data directory>/config/_notifications/` which stores [FreeMarker \(.ftl\)](#) templates. There are also `.dist` files that store the default templates. Each notification type evaluates a template file with a corresponding name. The template files can be modified while the server is running.

By default, the server checks for changes in the files each 60 seconds, but this can be changed by setting the `teamcity.notification.temp`

`late.update.interval` internal property to the desired number of seconds.

If there occurs an error during the template evaluation, TeamCity logs the error details into `teamcity-notifications.log`. There can be non-critical errors that result in ignoring part of the template or critical errors that result in inability to send notification at all. Whenever you make changes to the notification templates please ensure the notification can still be sent.

This document doesn't describe the FreeMarker template language, so if you need a guidance on the FreeMarker syntax, please refer to the corresponding template manual at <http://freemarker.org/docs/dgui.html>.

Supported Output Values

TeamCity notifiers use templates to evaluate output values (global template variables) which are then retrieved by name. The following output values are supported:

Email Notifier

- **subject** - subject of the email message to send
- **body** - plain text of the email message to send
- **bodyHtml** - (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 an email. One header per line. For example:

```
<#global headers>
X-Priority: 1 (Highest)
Importance: High
</#global>
```

Jabber

- **message** - plain text of the message to send

IDE Notifications and Windows Tray Notifications

- **message** - plain text of the message to send
- **link** - URL of the TeamCity page that contains detailed information about the event

(i) The Atom/RSS feeds template differs from the others. For the details, please refer to the [dedicated section](#).

Customization Examples

This section provides Freemarker code snippets that can be used for customization of notifications:

Including **ERRORS** from the log

```
<#list build.buildLog.messages[1..] as message><!-- skipping the first message (it is
a root node)-->
<if message.status == "ERROR" || message.status == "FAILURE" >
    ${message.text}
</if>
</list>
```

You can include it into the `build_failed.ftl` template as follows:

```

<!-- Uses FreeMarker template syntax, template guide can be found at
http://freemarker.org/docs/dgui.html -->

<#import "common.ftl" as common>

<#global subject>[<@common.subjMarker/> FAILED] ${project.name}:${buildType.name} -
Build: ${build.buildNumber}</#global>

<#global body>TeamCity build: ${project.name}:${buildType.name} - Build:
${build.buildNumber} failed ${var.buildShortStatusDescription}.
Agent: ${agentName}
Build results: ${link.buildResultsLink}

${var.buildCompilationErrors}${var.buildFailedTestsErrors}${var.buildChanges}

<#list build.buildLog.messages[1..] as message><!-- skipping the first message (it is
a root node)-->
<if message.status == "ERROR" || message.status == "FAILURE" >
    ${message.text}
</if>
</list>

<@common.footer/></#global>

<#global bodyHtml>
<div>
    <div>
        TeamCity build: <i>${project.name}:${buildType.name} - <a
        href='${link.buildResultsLink}'>Build: ${build.buildNumber}</a></i> failed
        ${var.buildShortStatusDescription}
    </div>
    <@common.build_agent build/>
    <@common.build_comment build/>
    <br>
    <@common.build_changes var.changesBean/>
    <@common.compilation_errors var.compilationBean/>
    <@common.test_errors var.failedTestsBean/>
    <@common.footerHtml/>
</div>
</#global>

```

The errors will be listed in the plain text part of the e-mail. To include it into the HTML part as well, you will need to add the same snippet into `<#global bodyHtml>` (e.g. right before "`<@common.footerHtml/>`")

Listing build artifacts

```

<p>Build artifacts:</p>
<#list build.artifactsDirectory.listFiles() as file>
    <a href="${webLinks.getDownloadArtifactUrl(build.buildTypeExternalId,
build.buildId, file.name)}">${file.name}</a> (${file.length()}B)<br/>
</list>

```

This will list only the root artifacts and include the `.teamcity` directory, which can be changed by modifications to the code.

Listing build parameters

```
<#list build.parametersProvider.all?keys as param>
    ${param} - ${build.parametersProvider.all[param]}
    <br>
</#list>
```

This will list the parameters that are passed to the build from the server.

Default Data Model

For the template evaluation TeamCity provides the default data model that can be used inside the template. The objects exposed in the model are instances of the corresponding classes from [TeamCity server-side open API](#).

The set of available objects model differs for different events.

You can also add your own objects into the model via plugin. See [Extending Notification Templates Model](#) for details.

Here is an example description of the model (the code can be used in IntelliJ IDEA to edit the template with completion):

```
<#--- @ftlvariable name="project" type="jetbrains.buildServer.serverSide.SProject" --->
<#--- @ftlvariable name="buildType" type="jetbrains.buildServer.serverSide.SBuildType" --->
<#--- @ftlvariable name="build" type="jetbrains.buildServer.serverSide.SBuild" --->
<#--- @ftlvariable name="agentName" type="java.lang.String" --->
<#--- @ftlvariable name="buildServer"
type="jetbrains.buildServer.serverSide.SBuildServer" --->
<#--- @ftlvariable name="webLinks" type="jetbrains.buildServer.serverSide.WebLinks" --->

<#--- @ftlvariable name="var.buildFailedTestsErrors" type="java.lang.String" --->
<#--- @ftlvariable name="var.buildShortStatusDescription" type="java.lang.String" --->
<#--- @ftlvariable name="var.buildChanges" type="java.lang.String" --->
<#--- @ftlvariable name="var.buildCompilationErrors" type="java.lang.String" --->

<#--- @ftlvariable name="link.editNotificationsLink" type="java.lang.String" --->
<#--- @ftlvariable name="link.buildResultsLink" type="java.lang.String" --->
<#--- @ftlvariable name="link.buildChangesLink" type="java.lang.String" --->
<#--- @ftlvariable name="responsibility"
type="jetbrains.buildServer.responsibility.ResponsibilityEntry" --->
<#--- @ftlvariable name="oldResponsibility"
type="jetbrains.buildServer.responsibility.ResponsibilityEntry" --->
```

TeamCity Notification Properties

The following properties can be useful to customize the notifications behaviour:

teamcity.notification.template.update.interval - how often the templates are reread by system (integer, in seconds, default 60)
teamcity.notification.includeDebugInfo - include debug information into the message in case of template processing errors (boolean, default false)
teamcity.notification.maxChangesNum - max number of changes to list in e-mail message (integer, default 10)
teamcity.notification.maxCompilationDataSize - max size (in bytes) of compilation error data to include in e-mail message (integer, default 20480)
teamcity.notification.maxFailedTestNum - max number of failed tests to list in e-mail message (integer, default 50)
teamcity.notification.maxFailedTestStacktraces - max number of test stacktraces in e-mail message (integer, default 5)
teamcity.notification.maxFailedTestDataSize - max size (in bytes) of failed test output data to include in a single e-mail message (integer, default 10240)

Syndication Feed Template

The template uses different approach to configuration from other notification engines.

The default template is stored in the file: <TeamCity data directory>/config/default-feed-item-template.ftl. This file should never be edited: it is overwritten on every server startup with the default copy. To specify a new template to use, copy the file under the name feed-item-template.ftl into the same directory. This file can be edited and will not be overwritten. It will be used by the engine if present.

The template is a [FreeMarker](#) template and can be freely edited.

You can use several templates on the single sever. The template name can be passed as a [URL parameter](#) of the feed URL.

During feed rendering, the template is evaluated to get the feed content. The resultant content is defined by the global variables defined in the template.

See the default template for an example of available input variables and output variables.

See also:

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

Assigning Build Configurations to Specific Build Agents

It is sometimes necessary to manage the [Build Agents](#)' workload more effectively. For example, if the time-consuming performance tests are run, the Build Agents with low hardware resources may slow down. As a result, more builds will enter the [build queue](#), and the feedback loop can become longer than desired. To avoid such situation, you can:

1. Establish a [run configuration policy](#) for an agent, which defines the build configurations to run on this agent.
2. Define [special agent requirements](#), to restrict the pool of agents, on which a build configuration can run the builds. These requirements are:
 - [Build Agent name](#). If the name of a build agent is made a requirement, the build configuration will run builds on this agent only.
 - [Build Agent property](#). If a certain property, for example, a capability to run builds of a certain configuration, an operating system etc., is made a requirement, the build configuration will run builds on the agents that meet this requirement.



- You can modify these parameters when setting up the project or build configuration, or at any moment you need. The changes you make to the build configurations are applied on the fly.
- You can specify a particular build agent to run a build on when [Triggering a Custom Build](#).

Agent pools

You could split agents into pools. Each project could be associated to a number of pools. See [Agent Pools](#).

Establishing a Run Configuration Policy

To establish a Build Agent's run configuration policy:

1. Click the **Agents** and select the desired build agent.
2. Click the **Compatible Configurations** tab.
3. Select **Run selected configurations only** and tick the desired build configurations names to run on the build agent.

Making Build Agent Name and Property a Build Configuration Requirement

To make a build configuration run the builds on a build agent with the specified name and properties:

1. Click **Administration** and select the desired build configuration.
2. Click Agent Requirements (see [Configuring Agent Requirements](#)).
3. Click the **Add requirement for a property** link, type the **agent.name** property, set its condition to **equals** and specify the build agent's name.



Note
You can also use the condition **contains**, however, it may include more than one specific build agent (e.g. a build configuration with a requirement **agent.name contains** Agent10, will run on agents named **Agent10**, **Agent10a**, and **Agent10b**).

4. Click the **Add requirement for a property** link and add the required property, condition, and value. For example, if you have several Linux-only builds, you can add the **teamcity.agent.jvm.os.name** property and set the **starts with** condition and the **linux** value.

See also:

[Concepts: Build Agent](#) | [Agent Requirements](#) | [Run Configuration Policy](#)
[Administrator's Guide: Triggering a Custom Build](#)

Patterns For Accessing Build Artifacts

 It is recommended to use the TeamCity REST API for accessing artifacts from scripts, as the REST API provides build selection facilities and allows for artifacts listing.

This section is preserved for **backward-compatibility** with the previous TeamCity versions and for some specific functionality.

Check the following information as well:

- If you need to access the artifacts in your builds, consider using the TeamCity's built-in [Artifact Dependency](#) feature.
- You can also download artifacts from TeamCity using the [Ivy](#) dependency manager.
- For artifact downloads from outside TeamCity builds, consider using [REST API](#).
- See also [Accessing Server by HTTP](#) on basic rules covering HTTP access from scripts.

This page covers:

- [Obtaining Artifacts](#)
- [Obtaining Artifacts from an Archive](#)
- [Obtaining Artifacts from a Build Script](#)
- [Links to the Artifacts Containing the TeamCity Build Number](#)

Obtaining Artifacts

To download artifacts of the latest builds (last finished, successful or pinned), use the following paths:

```
/repository/download/BUILD_TYPE_EXT_ID/.lastFinished/ARTIFACT_PATH  
/repository/download/BUILD_TYPE_EXT_ID/.lastSuccessful/ARTIFACT_PATH  
/repository/download/BUILD_TYPE_EXT_ID/.lastPinned/ARTIFACT_PATH
```

To download artifacts by the **build id**, use:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_ID:id/ARTIFACT_PATH
```

To download artifacts by the **build number**, use:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_NUMBER/ARTIFACT_PATH
```

To download artifacts from the latest build with a specific tag, use:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_TAG.tcbuildtag/ARTIFACT_PATH
```

To download all artifacts in a **.zip archive**, use:

```
/repository/downloadAll/BUILD_TYPE_EXT_ID/BUILD_SPECIFICATION
```

where

- **BUILD_TYPE_EXT_ID** is a build configuration ID.

- **BUILD_SPECIFICATION** can be `.lastFinished`, `.lastSuccessful` or `.lastPinned`, specific buildNumber or **build id** in format `BUILD_ID:id`.
- **ARTIFACT_PATH** is a path to the artifact on the TeamCity server. This path can contain a `{build.number}` pattern which will be replaced with the build number of the build whose artifact is retrieved.
By default, the archive with all artifacts does not include **hidden artifact**. To include them, add "`?showAll=true`" at the end of the corresponding URL.

To download artifact from the last finished, last successful, last pinned or tagged build in a specific branch, add the "`?branch=<branch_name>`" parameter at the end of the corresponding URL.

Obtaining Artifacts from an Archive

TeamCity allows obtaining a file from an archive from the build artifacts directory by means of the following URL patterns:

```
/repository/download/BUILD_TYPE_EXT_ID/BUILD_SPECIFICATION/<archive>!/PATH_WITHIN_ARCHIVE
```

- **BUILD_TYPE_EXT_ID** is a **build configuration ID**.
- **BUILD_SPECIFICATION** can be `.lastFinished`, `.lastPinned`, `.lastSuccessful`, specific buildNumber or **build id** in format `BUILD_ID:id`.
- **PATH_WITHIN_ARCHIVE** is a path to a file within a zip/7-zip/jar/tar.gz archive on TeamCity server.

Following archive types are supported (case-insensitive):

- `.zip`
- `.7z`
- `.jar`
- `.war`
- `.ear`
- `.nupkg`
- `.sit`
- `.apk`
- `.tar.gz`
- `.tgz`
- `.tar.gzip`
- `.tar`

Obtaining Artifacts from a Build Script

It is often required to download artifacts of some build configuration by tools like `wget` or another downloader which does not support HTML login page. TeamCity asks for authentication if you accessing artifacts repository.

To authenticate correctly from a build script, you have to change URLs (add `/httpAuth/` prefix to the URL):

```
/httpAuth/repository/download/BUILD_TYPE_EXT_ID/.lastFinished/ARTIFACT_PATH
```

Basic authentication is required for accessing artifacts by this URLs with `/httpAuth/` prefix.

You can use existing TeamCity username and password in basic authentication settings, but consider using `teamcity.auth.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 `HKEY_LOCAL_MACHINE\Software\Novell\Mono\DefaultCLR`
2. Frameworks paths are extracted from `HKEY_LOCAL_MACHINE\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).

- Global Settings
- User-Level Settings

Global Settings

Maven *global-level* settings are used from the `.xml` file in the default Maven location for the TeamCity server process: `$(env.M2_HOME)/conf/settings.xml` (or `$(system.maven.home)/conf/settings.xml`) (global values of `M2_HOME` environment variable and `maven.home` JVM option are used - those set for the TeamCity server process),

User-Level Settings

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).

Since **TeamCity 9.0 EAP2**, user-level settings can be configured in the [Maven Artifact Dependency Trigger](#).

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. Maven settings are defined on the project level: the **Project Settings | Maven Settings** tab. The settings are stored in the `<TeamCity Data Directory>/config/projects/%projectId%/pluginData/mavenSettings` directory.



The settings are available in the current project and its subprojects. To override the inherited settings, in a subproject create a new `settings.xml` file with the same name as the inherited one.)

For the logic of Maven settings, please refer to the related Maven [documentation](#).

See also:

[Administrator's Guide: Maven | Maven Artifact Dependency Trigger | Creating Maven Build Configuration](#)

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... (Apple > Settings... on Mac OS X)** 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 the **Update** command in the **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 Tools** area on the **My Settings & Tools** page of TeamCity web UI.

To install the TeamCity plugin:

1. In the top right corner of the TeamCity web UI, click the arrow next to your username, and select **My Settings & Tools**.
2. 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"

See also:

Troubleshooting: Logging in IntelliJ IDEA/Platform-based IDEs

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 2014.1 and Eclipse EGIt 1.0+

Installing the Plugin

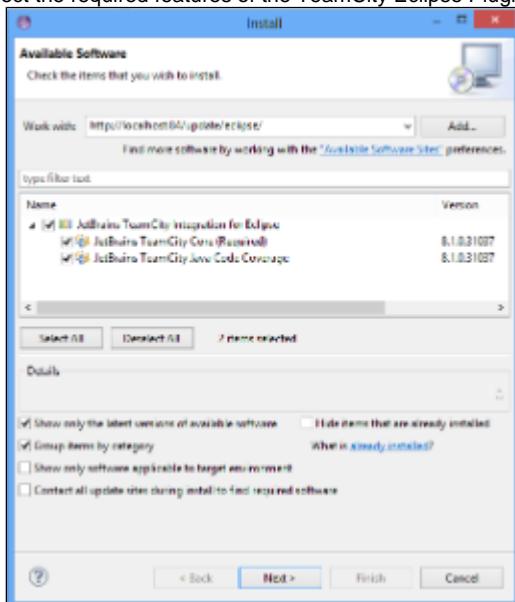
The TeamCity Eclipse plugin version must correspond to the version of the TeamCity server it connects to. Connections to TeamCity servers with different versions are generally not supported.

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.

See also:

Visual Studio Addin

The TeamCity add-in for Microsoft Visual Studio provides the following features:

- Remote Run for TFS, Subversion and Perforce (for remote run for Mercurial and Git see [Branch Remote Run Trigger](#)),
- [Pre-Tested \(Delayed\) Commit](#) for TFS, Subversion and Perforce,
- fetching JetBrains dotCover coverage analysis data from the TeamCity server (see [more](#)) to MS Visual Studio (requires dotCover of the supported version installed in Visual Studio),
- viewing recently committed changes and personal builds with their build status in the My Changes tool window,
- opening build failure details in MS Visual Studio from the TeamCity web UI,
- viewing failed tests' details for a build,
- re-running tests failed in the TeamCity build locally via the [ReSharper test runner](#) (requires ReSharper 5.0 installed),
- navigation from the IDE to the build results web page,
- re-applying changes sent in Remote Run or Pre-tested commit to the working directory.

For detailed instructions, refer to the TeamCity Add-in Help section in Visual Studio.

 To enable navigation to the failed tests in MS Visual Studio by using "open in IDE" actions in the web UI, make sure that .pdb file generation for the assemblies involved in NUnit/MSTest unit tests is switched on in the current Visual Studio project.

On this page:

- [Installing the Add-in](#)
- [Requirements](#)

Installing the Add-in

The TeamCity VS Add-in version must correspond to the version of the TeamCity server it connects to. Connections to TeamCity servers with different versions are generally not supported.

1. Close all running instances of Visual Studio before starting the Add-in installation (initial or upgrade).
2. Navigate to the download page of the Visual Studio Add-in:
 - a) Click the arrow next to your username in the top right corner of the TeamCity web UI and select **My Settings&Tools**
 - b) In the **TeamCity Tools** section on the right, click the Visual Studio Add-in download link.
3. Select the appropriate version of the TeamCity Visual Studio Add-in and click the link to download the installer.

There are two versions of the Visual Studio Add-in:

- [The TeamCity Visual Studio Add-in is shipped as a part of ReSharper Ultimate products bundle since TeamCity 9.0](#). This is the recommended installer for the TeamCity Visual Studio Add-in. After installation, the TeamCity Add-in will be available under the **RESHARPER** menu in Visual studio.
 Note that the installer will remove the pre-bundle products versions: TeamCity and ReSharper versions prior to 9.0, dotCover prior to 3.0, dotTrace prior to 6.0.
- [The Legacy version of TeamCity VS Add-in](#) (available for Visual Studio versions from 2005 to 2013 and compatible with JetBrains .NET tools prior to ReSharper 9.0, dotCover 3.0 and dotTrace 6.0.) should be chosen if
 - you are using the versions of JetBrains .Net tools integrated with Visual Studio **prior to ReSharper Ultimate** and **not planning to upgrade** (the new TeamCity VS Add-in web installer will remove the pre-bundle products versions)
 - you are using the Visual Studio versions **2005 and 2008 not supported by the ReSharper Ultimate**.

 The TeamCity Add-in installed as a part of **ReSharper Ultimate** will not use the settings provided by the 8.1 version. The legacy add-in downloaded from the TeamCity server will use the settings from the previous version.

Requirements

See the [Supported Platforms and Environment](#) page for the system requirements to configure integration with different version control systems or coverage tools.

See also:

[Related blog posts: TeamCity plugin for Visual Studio@TeamCity blog](#)
[Troubleshooting TeamCity Visual Studio Add-in issues](#)

Windows Tray Notifier

The Windows Tray Notifier is a utility which allows monitoring the status of 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](#)

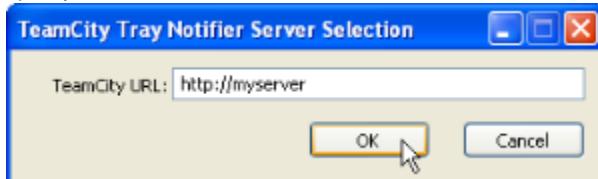
[Installing Tools: Working with Windows Tray Notifier](#)

Working with Windows Tray Notifier

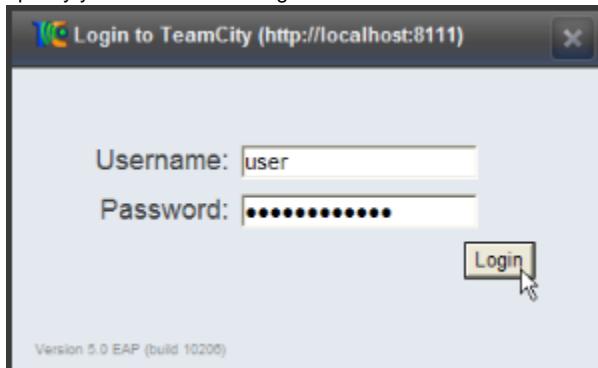
To launch Windows Tray Notifier, run the **Start > Programs > Windows Tray Notifier** menu.

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

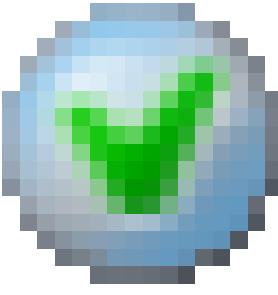
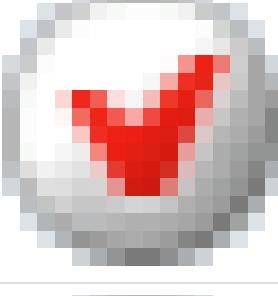
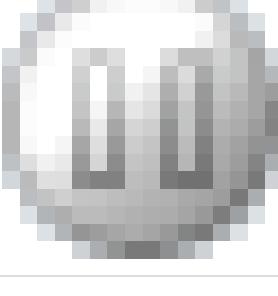
- #Tray Status Icons
- #Quick View Window
- #Pop-up Notification

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
------	---------

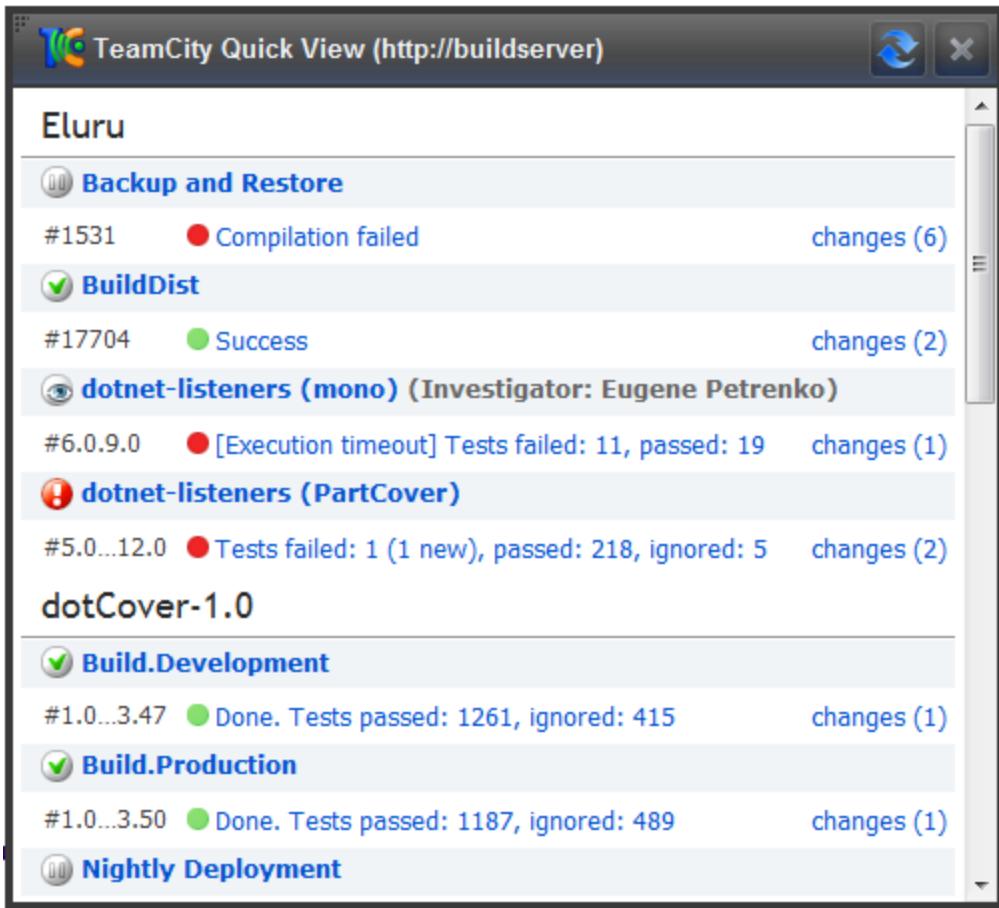
	Build is successful
	Build failed and nobody is investigating the failure
	Some team member has started investigation of the build failure
	The person who investigated the build failure has submitted a fix, but the build has not executed successfully yet
	Build configuration is paused, and builds are 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:



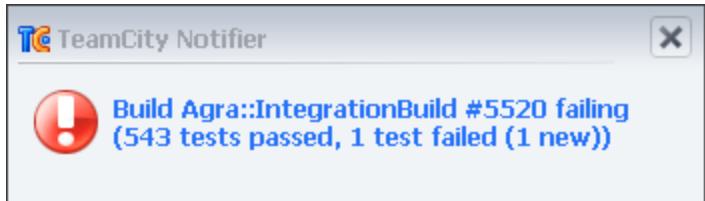
Click the *results* or *changes* links to navigate to the **Build Results** and **Changes** pages in TeamCity web interface, respectively, and investigate the desired issues deeper.

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, in the Syndication Feed section click the **Customize** link.
3. In the [Feed URL Generator page](#) specify the build configurations and events (builds and/or changes) you want to be notified about, and define authentication settings.
4. Copy the Feed URL, generated by TeamCity, to your feed reader, or click **Subscribe**.
5. Preview summary of the subscription and click **Subscribe now**.

See also:

[User's Guide: Feed URL Generator](#)

Extending TeamCity

TeamCity behavior can be extended in several ways. You can communicate with TeamCity from the build script and report tests, change build number or provide statistics data. Or you can write full-fledged plugin which will provide custom UI, customized notifications and much more.

If you cannot find relevant information here, have questions or want to share your TeamCity plugins experience with other users, welcome to [TeamCity Plugins Forum](#).

Customizing TeamCity without Plugins

- Build Script Interaction with TeamCity
- Accessing Server by HTTP
- Including Third-Party Reports in the Build Results

Plugin Development

- Typical Plugins
 - Build Runner Plugin
 - Risk Tests Reordering in Custom Test Runner
 - Custom Build Trigger
 - Extending Notification Templates Model
 - Issue Tracker Integration Plugin
 - Version Control System Plugin
 - Version Control System Plugin (old style - prior to 4.5)
 - Custom Authentication Module
 - Custom Notifier
 - Custom Statistics
 - Custom Server Health Report
 - Extending Highlighting for Web diff view
- Bundled Development Package
- Open API Changes
- Plugin Types in TeamCity
- Plugins Packaging
- Server-side Object Model

- Agent-side Object Model
- Extensions
- Web UI Extensions
- Plugin Settings
- Development Environment
- Developing Plugins Using Maven
- Plugin Development FAQ
- Getting Started with Plugin Development

[Open API Javadoc](#)

[Open API Javadoc \(ver. 9.0.x\)](#)

Publicly Available Plugins

- TeamCity Plugins
- Open-source Bundled Plugins

Build Script Interaction with TeamCity

If TeamCity doesn't support your testing framework or build runner out of the box, you can still avail yourself of many TeamCity benefits by customizing your build scripts to interact with the TeamCity server. This makes a wide range of features available to any team regardless of their testing frameworks and runners. Some of these features include displaying real-time test results and customized statistics, changing the build status, and publishing artifacts before the build is finished.

The build script interaction can be implemented by means of:

- service messages in the build script
- `teamcity-info.xml` file



If you use MSBuild build runner, you can use MSBuild Service Tasks.

In this section:

- Service Messages
 - Common Properties
 - Message Creation Timestamp
 - Message FlowId
 - Reporting Messages For Build Log
 - Blocks of Service Messages
 - Reporting Compilation Messages
 - Reporting Tests
 - Supported test service messages
 - Nested test reporting
 - Interpreting test names
 - Reporting .NET Code Coverage Results
 - Publishing Artifacts while the Build is Still in Progress
 - Reporting Build Progress
 - Reporting Build Problems
 - Reporting Build Status
 - Reporting Build Number
 - Adding or Changing a Build Parameter
 - Reporting Build Statistics
 - Disabling Service Messages Processing
 - Importing XML Reports
 - Libraries reporting results via TeamCity Service Messages
- `teamcity-info.xml`
 - Modifying the Build Status
 - Reporting Custom Statistics
 - Providing data using the `teamcity-info.xml` file
 - Describing custom charts

Service Messages

Service messages are used to pass commands/build information to a TeamCity server from the build script. In order to be processed by TeamCity, they should be printed into a 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 cannot contain a newline character inside it, it cannot 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=OWSP'value'WSPpropertyName_IDOWSP=OWSP'value'...OWSP]
```

where:

- **messageName** is a name of the message. See below for supported messages. The message name should be a valid Java id (only alpha-numeric characters and "-", starting with an alpha character)
- **propertyName** is a name of the message attribute. Should be a valid Java id.
- **value** is a value of the attribute. Should be an escaped value (see below).
- **WSP** is a required whitespace(s): space or tab character (\t)
- **OWSP** is an optional whitespace(s)
- ... is any number of **WSP****propertyName**OWSP=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' ... ]
```

The 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' ...]
```

Reporting Messages For Build Log

You can report messages for a build log in the following way:

```
##teamcity[message text='<message text>' errorDetails='<error details>'  
status='<status value>']
```

where:

- The `status` attribute may take following values: NORMAL, WARNING, FAILURE, ERROR. The default value is NORMAL.
- The `errorDetails` attribute is used only if `status` is ERROR, in other cases it is ignored.

This message fails the build in case its status is ERROR and "Fail build if an error message is logged by build runner" box is checked on the [Build Failure Conditions](#) page of a build configuration. For example:

```
##teamcity[message text='Exception text' errorDetails='stack trace' status='ERROR']
```

Blocks of Service Messages

Blocks are used to group several messages in the build log.

Block opening:

```
##teamcity[blockOpened name='<blockName>']
```

Block closing:

```
##teamcity[blockClosed name='<blockName>']
```



Please note that when you close the block, all inner blocks are closed automatically.

Reporting Compilation Messages

```

##teamcity[compilationStarted compiler='<compiler name>']
...
##teamcity[message text='compiler output']
##teamcity[message text='compiler output']
##teamcity[message text='compiler error' status='ERROR']
...
##teamcity[compilationFinished compiler='<compiler name>']

```

where:

- `compiler name` is an arbitrary name of the compiler performing compilation, eg. `javac`, `groovyc` and so on. Currently it is used as a block name in the build log.
- any message with status `ERROR` reported between `compilationStarted` and `compilationFinished` will be treated as a compilation error.

Reporting Tests

To use the TeamCity on-the-fly test reporting, a testing framework needs dedicated support for this feature to work (alternatively, [XML Report Processing](#) can be used).

If TeamCity doesn't support your testing framework natively, it is possible to modify your build script to report test runs to the TeamCity server using service messages. This makes it possible to display test results in real-time, make test information available on the [Tests](#) tab of the [Build Results](#) page.

Supported test service messages

Test suite messages:

Test suites are used to group tests. TeamCity displays tests grouped by suites on [Tests](#) tab of the [Build Results](#) page and in other places.

```

##teamcity[testSuiteStarted name='suiteName']
<individual test messages go here>
##teamcity[testSuiteFinished name='suiteName']

```

All the individual test messages are to appear between `testSuiteStarted` and `testSuiteFinished` (in that order) with the same `name` attributes.

Nested test reporting

[Prior to TeamCity 9.1](#), one test could have been reported from within another test.

[Since TeamCity 9.1](#), starting another test finishes the currently started test in the same "flow". To still report tests from within other tests, you will need to specify another `flowId` in the nested test service messages.

Test start/stop messages:

```

##teamcity[testStarted name='testName' captureStandardOutput='<true/false>']
<here go all the test service messages with the same name>
##teamcity[testFinished name='testName' duration='<test_duration_in_milliseconds>']

```

Indicates that the test "`testName`" was run. If the `testFailed` message is not present, the test is regarded successful.

duration (optional numeric attribute) - sets the test duration in milliseconds (should be an integer) to be reported in TeamCity UI. If omitted, the test duration will be calculated from the messages timestamps. If the timestamps are missing, from the actual time the messages were received on the server.

captureStandardOutput (optional boolean attribute) - if `true`, all the standard output (and standard error) messages received between `testStarted` and `testFinished` messages will be considered test output. The default value is `false` and assumes usage of `testStdOut` and `testsStdErr` 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 must be reported under the same name in every build). Include absolute paths in the reported test names is **strongly discouraged**.

Ignored tests:

```
##teamcity[testIgnored name='testName' message='ignore comment']
```

Indicates that the test "testName" is present but was not run (was ignored) by the testing framework.

As an exception, the `testIgnored` message can be reported without the matching `testStarted` and `testFinished` messages.

Test output:

```
##teamcity[testStarted name='className.testName']
##teamcity[testStdOut name='className.testName' out='text']
##teamcity[testStdErr name='className.testName' out='error text']
##teamcity[testFinished name='className.testName' duration='50']
```

The `testStdOut` and `testStdErr` service messages report the test's standard and error output to be displayed in the TeamCity UI. There must be only one `testStdOut` and one `testStdErr` message per test.

An alternative but a less reliable approach is to use the `captureStandardOutput` attribute of the `testStarted` message.

Test result:

```
##teamcity[testStarted name='MyTest.test1']
##teamcity[testFailed name='MyTest.test1' message='failure message' details='message and stack trace']
##teamcity[testFinished name='MyTest.test1']

##teamcity[testStarted name='MyTest.test2']
##teamcity[testFailed type='comparisonFailure' name='MyTest.test2' message='failure message' details='message and stack trace' expected='expected value' actual='actual value']
##teamcity[testFinished name='MyTest.test2']
```

Indicates that the "`testname`" test failed. Only one `testFailed` message can appear for a given test name.

`message` contains the textual representation of the error.

`details` contains detailed information on the test failure, typically a message and an exception stacktrace.

`actual` and `expected` attributes can only be used together with `type='comparisonFailure'` to report comparison failure. The values will be used when opening the test in the IDE.

Here is a longer example of test reporting with service messages:

```
##teamcity[testSuiteStarted name='suiteName']
##teamcity[testSuiteStarted name='nestedSuiteName']
##teamcity[testStarted name='package_or_namespace.ClassName.TestName']
##teamcity[testFailed name='package_or_namespace.ClassName.TestName' message='The number should be 20000' details='junit.framework.AssertionFailedError: expected:<20000> but was:<10000>|n|r      at junit.framework.Assert.fail(Assert.java:47)|n|r      at junit.framework.Assert.failNotEquals(Assert.java:280)|n|r...']
##teamcity[testFinished name='package_or_namespace.ClassName.TestName']
##teamcity[testSuiteFinished name='nestedSuiteName']
##teamcity[testSuiteFinished name='suiteName']
```

Interpreting test names

The **Tests** tab of the **Build Results** page allows grouping by suites, packages/namespaces, classes, and tests. Usually the attribute values are provided as they are reported by your test framework and TeamCity is able to interpret which part of the reported names is the test name, class, package as follows:

- TeamCity takes the suite name from the corresponding suite message
- if the reported test name starts with the suite name, it is truncated
- the part of the test name *after* the last dot is treated as a test name
- the part of the test name *before* the last dot is treated as a class name
- the rest of the test name is treated as a package/namespace name

Reporting .NET Code Coverage Results

You can configure .NET coverage processing by means of service messages. To learn more, refer to [Manually Configuring Reporting Coverage page](#).

Publishing Artifacts while the Build is Still in Progress

You can publish the build artifacts while the build is still running, immediately after the artifacts are built.

To do this, you need to output the following line:

```
##teamcity[publishArtifacts '<path>']
```

The <path> has to adhere to the same rules as the [Build Artifact specification](#) of the Build Configuration settings.



To be processed, an artifact file or directory must be located in the checkout directory, and the path must be relative to [this directory](#).

The files matching the <path> will be uploaded and visible as the artifacts of the running build.

The message should be printed after all the files are ready and no file is locked for reading. Artifacts are uploaded in the background, which can take time. Please make sure the matching files are not deleted till the end of the build. (e.g. you can put them in a directory that is cleaned on the next build start, in a [temp directory](#) or use [Swabra](#) to clean them after the build.)



The process of publishing artifacts process can affect the build because it consumes network traffic and some disk/CPU resources (should be pretty negligible for not large files/directories).

Artifacts that are specified in the build configuration setting will be published as usual.

Reporting Build Progress

You can use special progress messages to mark long-running parts in a build script. These messages will be shown on the projects dashboard for the corresponding build and on the [Build Results page](#).

To log a single progress message, use:

```
##teamcity[progressMessage '<message>']
```

This progress message will be shown until another progress message occurs or until the next target starts (in case of Ant builds).

If you wish to show a progress message for a part of a build only, use:

```
##teamcity[progressStart '<message>']
...some build activity...
##teamcity[progressFinish '<message>']
```



The same message should be used for both `progressStart` and `progressFinish`. This allows nesting progress blocks. Also note that in case of Ant builds, progress messages will be replaced if an Ant target starts.

Reporting Build Problems

To fail a build directly from the build script, a build problem has been reported. Build problems appear on the [Build Results](#) page and also affect the build status text.

To add a build problem to a build, use:

```
##teamcity[buildProblem description='<description>' identity='<identity>']
```

where:

- **description** - (mandatory) a human-readable plain text describing the build problem. By default, the `description` appears in the build status text and in the list of build's problems. The text is limited to 4000 symbols, and will be truncated if the limit is exceeded.
- **identity** - (optional) a unique problem id. Different problems must have different identity, same problems - same identity, which should not change throughout builds if the same problem occurs, e.g. the same compilation error. It should be a valid Java id up to 60 characters. If omitted, the `identity` is calculated based on the `description` text.

Reporting Build Status

TeamCity allows changing the **build status text** from the build script. Unlike [progress messages](#), this change persists even after a build has finished.

You can also change the build status of a failing build to **success**.

Prior to TeamCity 7.1, this service message could have been used for changing build status to **failed**. Since TeamCity 7.1, the `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 the build is running and will also affect the final build results.

Reporting Build Number

To set a custom build number directly, specify a `buildNumber` message using the following format:

```
##teamcity[buildNumber '<new build number>']
```

In the `<new build number>` value, you can use the `{build.number}` substitution to use the current build number automatically generated by TeamCity. For example:

```
##teamcity[buildNumber '1.2.3_{build.number}-ent']
```

Adding or Changing a Build Parameter

By using a dedicated service message in your build script, you can dynamically update build parameters of the build right from a build step. The changed build parameters will be available in the build steps following the modifying one. They will also be available as build parameters and can be used in the dependent builds via `%dep.*%` parameter references.

```
##teamcity[setParameter name='ddd' value='fff']
```

When specifying a build parameter's name, mind the prefix:

- **system** for system properties.
- **env** for environment variables.
- no prefix for configuration parameter.
[Read more about build parameters and their prefixes.](#)

Reporting Build Statistics

In TeamCity, it is possible to configure a build script to report statistical data and then display the charts based on the data. Please refer to the [Customizing Statistics Charts#customCharts](#) page for a guide to displaying the charts on the web UI. This section describes how to report the statistical data from the build script via service messages. You can publish the build statics values in two ways:

- Using a service message in a build script directly
- Providing data using the `teamcity-info.xml` file

To report build statistics using service messages:

- Specify a 'buildStatisticValue' service message with the following format for each statistics value you want to report:

```
##teamcity[buildStatisticValue key='<valueTypeKey>' value='<value>']
```

The key should not be equal to any of [predefined keys](#).

The value should be a positive/negative integer of up to 13 digits. **Since TeamCity 9.0**, float values with up to 6 decimal places are also supported.

Disabling Service Messages Processing

If you need for some reason to disable searching for service messages in the output, you can disable the service messages search with the messages:

```
##teamcity[enableServiceMessages]
##teamcity[disableServiceMessages]
```

Any messages that appear between these two are not parsed as service messages and are effectively ignored.

For server-side processing of service messages, enable/disable service messages also supports the flowId attribute and will ignore only the messages with the same flowId.

Importing XML Reports

In addition to the [UI Build Feature](#), XML reporting can be configured from within the build script with the help of the `importData` service message.

Also, the message supports importing of the previously collected code coverage and code inspection/duplicates reports.

The service message format is:

```
##teamcity[importData type='typeID' path='<path to the xml file>']
```



To be processed, report XML files (or a directory) must be located in the checkout directory, and the path must be relative to [this directory](#).

where `typeID` can be one of the following (see also [XML Report Processing](#)):

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 the findBugsHome attribute specified pointing to the home directory of the installed FindBugs tool.

³⁾ also requires the tool='<tool name>' service message attribute, where the <tool name> is either dotcover, partcover, ncover or ncover3.

- If not specially noted, the report types support Ant-like wildcards in the path attribute.
- the verbose='true' attribute will enable detailed logging into the build log.
- the parseOutOfDate='true' attribute will process all the files matching the path. Otherwise, only those updated during the build (is determined by last modification timestamp) are processed.
- the whenNoDataPublished=<action> (where <action> is one of the following: info (default), nothing, warning, error) will change output level if no reports matching the path specified were found.

(deprecated, use Build Failure Conditions instead)

findBugs, pmd or checkstyle importData messages also take optional errorLimit and warningLimit attributes specifying errors and warnings limits, exceeding which will cause the build failure.



- After the importData message is received, TeamCity agent starts to monitor the specified paths on the disk and imports matching report files in the background as soon as the files appear on disk.
- The parsing only occurs within the build step in which the messages were received. On the step finish, the agent ensures all the present reports are processed before beginning the next step. This behavior is different from that of [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 of several directories or parse several types of the report, send the corresponding service messages one after another.



Only several reports of different types can be included in a build. Processing reports of several inspections or duplicates tools in a single build is not supported. See the [related feature request](#).

Libraries reporting results via TeamCity Service Messages

Several platform-specific libraries from JetBrains and external sources are able to report the results via TeamCity Service messages.

- [Service messages .NET library](#) - .NET library for generating (and parsing) TeamCity service messages from .NET applications. See a related blog post.
- [Jasmine 2.0 TeamCity reporter](#) - support for emitting TeamCity service messages from Jasmine 2.0 reporter
- [Perl TAP Formatter](#) - formatter for Perl to transform TAP messages to TeamCity service messages
- [PHPUnit Listener 1, PHPUnit Listener 2](#) - listeners which can be plugged via PHPUnit's suite.xml and will produce TeamCity service messages for tests
- [Python Unit Test Reporting to TeamCity](#) - the package that automatically reports unit tests to the TeamCity server via service messages (when run under TeamCity and provided the testing code is adapted to use it).
- [Mocha](#) - on-the-fly reporting via service messages for Mocha JavaScript testing framework. See the related [post](#) with instructions.
- [Karma](#) - support in the JavaScript testing tool to report tests progress into TeamCity using TeamCity service messages

teamcity-info.xml

It is also possible to have the build script collect information, generate an XML file called `teamcity-info.xml` in the root build directory. When the build finishes, this file will automatically be uploaded as a build artifact and processed by the TeamCity server.

Please note that this approach can be discontinued in the future TeamCity versions, so service messages approach is recommended instead. In case service messages does not work for you, please let us know the details and describe the case via [email](#).

Modifying the Build Status

TeamCity has the ability to change the build status directly from the build script. You can set the status (build failure or success) and change the text of the build status (for example, note the number of failed tests if the test framework is not supported by TeamCity).

XML schema for teamcity-info.xml

It is possible to set the following information for the build:

- **Build number** — Sets the new number for the finished build. You can reference the TeamCity-provided build number using `{build.number}`.
- **Build status** — Change the build status. Supported values are "FAILURE" and "SUCCESS".
- **Status text** — Modify the text of build status. You can replace the TeamCity-provided status text or add a custom part before or after the standard text. Supported action values are "append", "prepend" and "replace".

Example `teamcity-info.xml` file:

```
<build number="1.0.{build.number}">
    <statusInfo status="FAILURE"> <!-- or SUCCESS -->
        <text action="append"> fitness: 45</text>
        <text action="append"> coverage: 54%</text>
    </statusInfo>
</build>
```



It is up to you to figure out how to retrieve test results that are not supported by TeamCity and accurately add them to the `teamcity-info.xml` file.

Reporting Custom Statistics

It is possible to provide [custom charts](#) in TeamCity. Your build can provide data for such graphs using `teamcity-info.xml` file.

Providing data using the teamcity-info.xml file

This file should be created by the build in the root directory of the build. You can publish multiple statistics (see the details on the data format below) and create separate charts for each set of values.

The `teamcity-info.xml` file is to contain the code in the following format (you can combine various data in the `teamcity-info.xml` file):

```
<build>
    <statisticValue key="chart1Key" value="342"/>
    <statisticValue key="chart2Key" value="53"/>
</build>
```

The key should not be equal to any of [predefined keys](#).

The value should be a positive/negative integer of up to 13 digits. **Since TeamCity 9.0**, float values with up to 6 decimal places are also supported.

The key here relates to the key of the **valueType** tag used when describing the chart.

Describing custom charts

See [Customizing Statistics Charts](#) page for detailed description.

Accessing Server by HTTP

In addition to the commands described here, there is a [REST API](#) that you can use for certain operations. When available, using REST API is a preferred way over one described here.

The examples below assume that your server web UI is accessible via `http://teamcity.jetbrains.com:8111/` URL.

The TeamCity server supports basic HTTP authentication allowing 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&agentId=1&agentId=2
```

To trigger a build on all enabled and compatible agents, use "allEnabledCompatible" as agent ID:

```
http://testuser:testpassword@teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf&agentId=allEnabledCompatible
```

Triggering a Custom Build

TeamCity allows you to trigger a build with customized parameters. You can select particular build agent to run the build, define additional properties and environment variables, and select the particular sources revision (by specifying the last change to include in the build) to run the build with. These customizations will affect only the single triggered build and will not affect other builds of the build configuration.

To trigger a build on a specific change inclusively, use the following URL:

```
http://testuser:testpassword@teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf&modificationId=11112
```

`modificationId` — modification/change internal id which can be obtained from the web diff url.

To trigger a build with custom parameters (system properties and environment variables), use:

```
http://testuser:testpassword@teamcity.jetbrains.com:8111/httpAuth/action.html?add2Queue=MyBuildConf&name=<full property name1>&value=<value1>&name=<full property name2>&value=<value2>
```

Where <full property name> is a full property name with system./env. prefix or no prefix to define configuration parameter.
Please note that previous TeamCity versions used different syntax for this action. That syntax is still supported for compatibility reason, though.

To move build to the top of the queue, add the following to the query string

- &moveToTop=true

To run a personal build, add &personal=true to the query string.

To run a build on a feature branch:

```
http://testuser:testpassword@teamcity.jetbrains.com/httpAuth/action.html?add2Queue=bt10&branchName=master
```

REST API

On this page:

- General information
 - General Usage Principles
 - REST Authentication
 - Superuser access
 - REST API Versions
 - URL Structure
 - Locator
 - Examples
 - Supported HTTP Methods
 - Response Formats
 - Full and Partial Responses
 - 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
 - Build Artifacts
 - Authentication
 - Other Build Requests
 - Snapshot dependencies
 - Build Parameters
 - Build fields
 - Statistics
 - Build log
 - Tests
 - Investigations
 - Agents
 - Agent Pools
 - Assigning Projects to Agent Pools
 - Users
 - User Groups
- 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 an open-source [plugin](#) bundled **since TeamCity 5.0.**

To use the REST API, an application makes an HTTP request to the TeamCity server and parses the response.

The TeamCity REST API can be used for integrating applications with TeamCity and for those who want to script interactions with the TeamCity server. TeamCity's REST API allows accessing resources (entities) via URL paths.

Note: URL examples on this page assume that your server web UI is accessible via the <http://teamcity:8111/> URL.

General Usage Principles

This documentation is not meant to be comprehensive, but just provide some initial knowledge useful for using the API.

You can start by opening <http://teamcity:8111/httpAuth/app/rest> URL in your browser: this page will give you several pointers to explore the API.

Use <http://teamcity:8111/httpAuth/app/rest/application.wadl> to get the *full* list of supported requests and names of parameters. This is the primary source of discovery for the supported requests and their parameters.

You can start with <http://teamcity:8111/httpAuth/app/rest/server> request and then drill down following "href" attributes of the entities listed.

Please make sure you read through this "General information" section before using the API.

Experiment and read error messages returned: for the most part they should guide you to the right requests.

Example on how to explore the API

Suppose you want to know more on the agents and see (in "/app/rest/server" response) that there is a "/app/rest/agents" URL.

- try the "/app/rest/agents/" request - see the authorized agent list, get the "default" way of linking to an agent from the agent's element `href` attribute.
- get individual agent details via /app/rest/agents/id:10 URL (obtained from "href" for one of the elements of the previous request).
- if you send a request to "/app/rest/agents/aaa:bbb", you will get the list of the supported dimensions to find an agent via the agent's `locator`
- most of the attributes of the returned agent data (name, connected, authorized) can be used as "`<field name>`" in the "`app/rest/agents/<agentLocator>/<field name>`" request. Moreover, if you issue a request to the "`app/rest/agents/id:10/test`" URL, you will get a list of the supported fields in the error message

REST Authentication

You can authenticate yourself for the REST API in the following ways:

- Using basic HTTP authentication. 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) include "guestAuth" before the "/app/rest" part: e.g.: <http://teamcity:8111/guestAuth/app/rest/builds>
- if you are checking REST GET requests from within a browser and you are logged in to TeamCity in the browser, you can just use "/app/rest" URL: e.g.<http://teamcity:8111/app/rest/builds>

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 as `%system.teamcity.auth.userId%` and `%system.teamcity.auth.password%`). The server URI is available as `%teamcity.serverUrl%` within a build.

Superuser access

You can use the `super user account` with REST API: just provide no user name and the generated password logged into the server log.

REST API Versions

As REST API evolves from one TeamCity version to another, there can be incompatible changes in the protocol.

Under the <http://teamcity:8111/app/rest/> or <http://teamcity:8111/app/rest/latest> URL the latest version is available.

Under the <http://teamcity:8111/app/rest/<version>> URL, earlier versions CAN be available. Our general policy is to supply TeamCity

with at least ONE previous version.

In TeamCity 9.0.x you can use "8.1" or "7.0" instead of <version> to get earlier versions of the protocol.

Breaking changes in the API are described in [Upgrade Notes](#) section.

Please note that additions to the objects returned (such as new XML attributes or elements) are not considered major changes and do not cause the protocol version to increment.

Also, the endpoints marked with "Experimental" comment in `application.wadl` may change without a special notice in future versions.

Note: The examples on this page use the "/app/rest" relative URL, replace it with the one containing the version if necessary.

URL Structure

The general structure of the URL in the TeamCity API is `teamcityserver:port/<authType>/app/rest/<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`, etc.) 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 the following symbols: , :- ()
- dimension, allowing to filter entities using multiple criteria: `<dimension1>:<value1>,<dimension2>:<value2>`

Refer to each entity description for the supported locators.

If a request with invalid locators is sent, the error messages often hint on the error and list supported locator dimensions (only non-experimental ones) when an unknown dimension is detected.

Note: If the value contains 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/<projectsLocator>` - <http://teamcity:8111/httpAuth/app/rest/projects/id:RESTAPIPlugin> (the 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?locator=<buildLocator>` - <http://teamcity:8111/httpAuth/app/rest/buildTypes/id:bt284/builds?locator=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?locator=<changeLocator>` - [http://teamcity:8111/httpAuth/app/rest/changes?locator=buildType:\(id:bt133\),sinceChange:\(id:24234\)](http://teamcity:8111/httpAuth/app/rest/changes?locator=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

Full and Partial Responses

By default, when a list of entities is requested, only basic fields are included into the response. When a single entry is requested, all the fields are returned. The complex field values can be returned in full or basic form, depending on a specific entity.

It is possible to change the set of fields returned for XML and JSON responses for the majority of requests.

This is done by supplying the **fields** request parameter describing the fields of the top-level entity and sub-entities to return in the response. An example syntax of the parameter is: `field,field2(field2_subfield1,field2_subfield1)`. This basically means "include field and field2 of the top-level entity and for field2 include field2_subfield1 and field2_subfield1 fields". The order of the fields specification plays no role.

Examples:

```
http://teamcity.jetbrains.com/app/rest/buildTypes?locator=affectedProject:(id:TeamCityPluginsByJetBrains)&fields=buildType(id,name,project)
http://teamcity.jetbrains.com/app/rest/builds?locator=buildType:(id:bt345),count:10&fields=count,build(number,status,statusText,agent,lastChange,tags,pinned)
```

At this time, the response can sometimes include the fields/elements not specifically requested. This can change in the future versions, so it is recommended to specify all the fields/elements used by the client.

Logging

You can get details on errors and REST request processing in `logs\teamcity-rest.log` server log.

If you get an error in response to your request and want to investigate the reason, look into [rest-related server logs](#).

To get details about each processed request, turn on debug logging (e.g. set Logging Preset to "debug-rest" on the [Administration/Diagnostics](#) page or modify the Log4J "jetbrains.buildServer.server.rest" category).

CORS Support

TeamCity REST can be configured to allow [cross-origin requests](#).

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: <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/>
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> (produces XML, JSON and plain text depending on the "Accept" header, accepts plain text and (since 9.1) XML and JSON). Also supported are requests .../parameters/<parameter_name>/name and .../parameters/<parameter_name>/value.
Project name/description/archived status: GET/PUT http://teamcity:8111/httpAuth/app/rest/projects/<projectLocator>/<field_name> (accepts/produces text/plain) where <field_name> is one of "name", "description", "archived".

Project's parent project: GET/PUT XML <http://teamcity:8111/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> (produces XML, JSON and plain text depending on the "Accept" header, accepts plain text and (**since TeamCity 9.1**) XML and JSON). The requests .../parameters/<parameter_name>/name and .../parameters/<parameter_name>/value are also supported.

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-roots/<id>>

Attach VCS root to a build configuration: POST <http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/vcs-roots/<id>>. The XML posted is the same as retrieved by GET request to <http://teamcity:8111/httpAuth/app/rest/buildTypes/<buildTypeLocator>/vcs-roots/<id>>

Create a new build configuration with all settings: POST <http://teamcity:8111/httpAuth/app/rest/buildTypes>. The XML/JSON posted is the same as retrieved by GET request. (Note that /app/rest/project/XXX/buildTypes still uses the previous version notation

and accepts another entity.)

Create a new empty build configuration: POST plain text (name) to <http://teamcity:8111/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>)"`.

Note: If a build configuration utilizes feature branches, by default only builds from the default branch are returned. To retrieve all builds, add the following locator: `branch:default:any`. The whole path will look like this: `/httpAuth/app/rest/builds/?locator=buildType:One_Git,branch:default:any`

`agentName:<name>` - agent name to return only builds ran on the agent with name specified

`sinceBuild:(<buildLocator>)` - limit the list of builds only to those after the one specified

`sinceDate:<date>` - limit the list of builds only to those started after the date specified. The date should be in the same format as dates returned by REST API (e.g. "20130305T170030+0400").

`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.

Queued Builds

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/id:XXX>

For queued builds with snapshot dependencies, the revisions are available in the `revisions` element of the queued build node if a revision is fixed (for regular builds without snapshot dependencies it is not).

Get compatible agents for queued builds (useful for builds having "No agents" to run on)

GET <http://teamcity:8111/httpAuth/app/rest/buildQueue/id: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

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. This way, you can monitor the build completeness by querying build details using the "href" attribute of the build details returned on build triggering, until the build has the `state="finished"` attribute. (There is a related outstanding issue.)

Build node examples

Basic build for a build configuration:

```
<build>
    <buildType id="buildConfID" />
</build>
```

Build for a branch marked as personal with a fixed agent, comment and a custom parameter:

```
<build personal="true" branchName="logicBuildBranch">
    <buildType id="buildConfID" />
    <agent id="3" />
    <comment><text>build triggering comment</text></comment>
    <properties>
        <property name="env.myEnv" value="bbb" />
    </properties>
</build>
```

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' readdIntoQueue='true' /> item to the URL of a running or queued build.

▼ Example of cancelling a running build: click to expand

```

curl -v -u user:password --request POST "http://teamcity:8111/app/rest/<buildLocator>" --data "<buildCancelRequest
comment=" readdIntoQueue='true' />" --header "Content-Type: application/xml"

```

Note: Readding of builds into the queue is supported for running builds only.

▼ Example of cancelling a queued build: click to expand

```

curl -v -u user:password --request POST "http://teamcity:8111/app/rest/buildQueue/<buildLocator>" --data
"<buildCancelRequest comment=" readdIntoQueue='false' />" --header "Content-Type: application/xml"

```

Expose cancelled build details:

See the canceledInfo element of the build item (available via GET <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>>)

Build Artifacts

<path> below can be empty for the root of build's artifacts or be a path within the build's artifacts. The path can span into the archive content, e.g. dir/path/archive.zip!/path_within_archive

GET http://teamcity:8111/httpAuth/app/rest/builds/<build_locator>/artifacts/content/<path> (returns the content of a build artifact)

Media-Type: application/octet-stream or a more specific media type (determined from artifact file extension)

Possible error: 400 if the specified path references a directory

GET `http://teamcity:8111/httpAuth/app/rest/builds/<build_locator>/artifacts/metadata/<path>` (returns information about a build artifact)
Media-Type: application/xml or application/json

GET `http://teamcity:8111/httpAuth/app/rest/builds/<build_locator>/artifacts/children/<path>` (returns the list of artifact children for directories and archives)
Media-Type: application/xml or application/json
Possible error: 400 if the artifact is neither a directory nor an archive

GET `http://teamcity:8111/httpAuth/app/rest/builds/<build_locator>/artifacts/archived/<path>?locator=pattern:<wildcard>` (returns the archive containing the list of artifacts under the path specified. The optional `locator` parameter can have file `<wildcard>` to limit the files only to those matching the `wildcard`)
Media-Type: application/zip
Possible error: 400 if the artifact is neither a directory nor an archive<artifact relative name> supports referencing files under archives using "!" delimiter after the archive name.

Examples:

```
GET http://teamcity:8111/httpAuth/app/rest/builds/id:100/artifacts/children/my-great-tool-0.1.jar!\META-INF  
GET http://teamcity:8111/httpAuth/app/rest/builds/id:100/artifacts/metadata/my-great-tool-0.1.jar!\META-INF  
GET http://teamcity:8111/httpAuth/app/rest/builds/id:100/artifacts/metadata/my-great-tool-0.1.jar!/lib/comm  
ons-logging-1.1.1.jar!\META-INF/MANIFEST.MF  
GET http://teamcity:8111/httpAuth/app/rest/builds/id:100/artifacts/content/my-great-tool-0.1.jar!/lib/comm  
ons-logging-1.1.1.jar!\META-INF/MANIFEST.MF
```

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

Snapshot dependencies

Since TeamCity 9.1 there is an experimental ability to retrieve entire build chain (all snapshot-dependency-linked builds) for a particular build:

```
http://teamcity:8111/httpAuth/app/rest/builds?locator=snapshotDependency:(to:(id:XXXX),includeInitial:true  
,defaultFilter:false)
```

This gets all the snapshot dependency builds recursively for the build with id XXXX.

Build Parameters

Get the parameters of a build: `http://teamcity:8111/httpAuth/app/rest/builds/id:<build id>/resulting-properties`

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 statistics for a single build: 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>`

Get statistics for a list of builds: GET `http://teamcity:8111/httpAuth/app/rest/builds?locator=BUILDS_LOCATOR&fields=buil
d(id,number,status,buildType(id,name,projectName),statistics(property(name,value)))`

Build log

Downloading build logs via a REST request is not supported, but there is a way to download the log files described [here](#).

Tests

List tests:

GET <http://teamcity:8111/app/rest/testOccurrences?locator=<locator dimension>:<value>>

Supported locators:

- build
- test
- currentlyFailing:true,affectedProject:<project locator>
- currentlyMuted:true,affectedProject:<project locator>

Examples:

List all build's tests: GET <http://teamcity:8111/app/rest/testOccurrences?locator=build:<build locator>>

Get individual test history:

GET <http://teamcity:8111/app/rest/testOccurrences?locator=test:<test locator>>

Supported test locators:

- "id:<internal test id>" available as a part of the URL on the test history page
- "name:<full test name>"

Investigations

List investigations in the Root project and its subprojects: <http://teamcity:8111/app/rest/investigations>

Supported locators:

- test: (id:TEST_NAME_ID)
- test: (name:FULL_TEST_NAME)
- assignee: (<user locator>)
- buildType:(id:XXXX)

Examples:

Get investigations for a specific test:

[http://teamcity:8111/app/rest/investigations?locator=test:\(id:TEST_NAME_ID\)](http://teamcity:8111/app/rest/investigations?locator=test:(id:TEST_NAME_ID))

[http://teamcity:8111/app/rest/investigations?locator=test:\(name:FULL_TEST_NAME\)](http://teamcity:8111/app/rest/investigations?locator=test:(name:FULL_TEST_NAME))

Get investigations assigned to a user: [http://teamcity:8111/app/rest/investigations?locator=assignee:\(<user locator>\)](http://teamcity:8111/app/rest/investigations?locator=assignee:(<user locator>))

Get investigations for a build configuration: [http://teamcity:8111/app/rest/investigations?locator=buildType:\(id:XXXX\)](http://teamcity:8111/app/rest/investigations?locator=buildType:(id:XXXX))

Agents

List of agents: GET <http://teamcity:8111/httpAuth/app/rest/agents>

List of connected authorized agents: GET <http://teamcity:8111/httpAuth/app/rest/agents?locator=connected:true,authorized:true>

List of authorized agents: GET <http://teamcity:8111/httpAuth/app/rest/agents?locator=authorized:true>

Since Teamcity 9.1: List of enabled authorized agents: GET <http://teamcity:8111/httpAuth/app/rest/agents?locator=enabled:true,authorized:true>

Agent's single field: GET/PUT <http://teamcity:8111/httpAuth/app/rest/agents/<agentLocator>/<field name>>

See also an [example](#) for agent enabling/disabling

Delete a build agent:

DELETE <http://teamcity:8111/httpAuth/app/rest/agents/<agentLocator>>

Agent Pools

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

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/remove specific user: PUT/DELETE <http://teamcity:8111/httpAuth/app/rest/users/>

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>>

User Groups

List of groups: GET <http://teamcity:8111/httpAuth/app/rest/userGroups>

List of users within a group: http://teamcity:8111/httpAuth/app/rest/userGroups/key:Group_Key

Create a group: POST <http://teamcity:8111/httpAuth/app/rest/userGroups>

Delete a group: DELETE http://teamcity:8111/httpAuth/app/rest/userGroups/key:Group_Key

Other

Data Backup

Start backup: POST <http://teamcity:8111/httpAuth/app/rest/server/backup?includeConfigs=true&includeDatabase=true&includeBuildLogs=true&fileName=> where <fileName> is the prefix of the file to save backup to. The file will be created in the default backup directory (see more).

Get current backup status (idle/running): GET <http://teamcity:8111/httpAuth/app/rest/server/backup>

Typed Parameters Specification

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.

Since TeamCity 9.1, partial updates of a parameter are possible (currently in an experimental state):

- name: PUT the same XML or JSON as returned by GET to <http://teamcity:8111/httpAuth/app/rest/buildTypes/<locator>/parameters/NAME>
- type: GET/PUT accepting XML and JSON as returned by GET to the URL <http://teamcity:8111/httpAuth/app/rest/buildTypes/<locator>/parameters/NAME/type>
- type's rawValue: GET/PUT accepting plain text <http://teamcity:8111/httpAuth/app/rest/buildTypes/<locator>/parameters/NAME/type/rawValue>

Build Status Icon

Icon that represents build status:

Before TeamCity 9.1.2: GET <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/statusIcon>

Since TeamCity 9.1.2: GET <http://teamcity:8111/httpAuth/app/rest/builds/<buildLocator>/statusIcon.svg>

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>.

The CCTray-format XML does not include paused build configurations by default. The URL accepts "locator" parameter instead with standard build configuration locator.

Request Examples

Request Sending Tool

You can use `curl` command line tool to interact with the TeamCity REST API.

Example command:

```
curl -v --basic --user USERNAME:PASSWORD --request POST "http://teamcity:8111/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-YYY-...-ZZZ
```

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

 More examples (for TeamCity 8.0) are available in this external posting.

Including Third-Party Reports in the Build Results

If your reporting tool produces reports in HTML format, you can extend TeamCity with a custom tab to show the information provided by the third-party reporting tool.

The report provided by your tool can be then displayed either on the build results page, or on the project home page.

The general flow is as follows:

- configure the build script to produce the HTML report (preferably in a zip archive);
- configure 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 a subproject.
- **Project-level**: appears on the Project home page for a particular project only if a build within the project produces the specified reports artifact.

To configure a report tab, go to the **Project Settings|Report Tabs** and select what type of report tab you want to add.

For a **project report tab**, specify the following:

Option	Description
Tab Title	Specify a unique title of the report tab that will be displayed in the web UI.
Get artifacts from	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 path-to-archive!relative-path syntax , e.g. <code>javadoc.zip!index.html</code> . See the list of supported archives .

For a **build report tab**, specify the following:

Option	Description
Tab Title	Specify a unique title of the report tab that will be displayed in the web UI.
Start page	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 path-to-archive!relative-path 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 on the Statistics tab, it is possible to configure your own statistical charts based on the set of [statistic values provided by TeamCity](#) or values reported from a build script. In the latter case you will need to configure your build script to report custom statistical data to TeamCity.

You can view statistic values reported by the build on the [Build parameters](#) page.



The information in this section refers to **TeamCity 9.x**. For other versions, refer to the [listing](#) to choose the corresponding documentation.

On this page:

- Managing Custom Charts via the TeamCity Web UI
 - Adding Custom Charts
 - Modifying Custom Charts
 - Reordering Custom Charts
- Managing Custom Charts Manually
 - Displaying Custom Chart in TeamCity Web UI
 - Tags Reference
 - Chart Dimensions
 - Chart Axis Settings
 - Default Statistics Values Provided by TeamCity
 - Custom Build Metrics

Managing Custom Charts via the TeamCity Web UI

Since **TeamCity 9.0**, you can manage custom charts using the TeamCity Web UI:

Adding Custom Charts

- The **Statistics** tab for a project or build configuration provides an option to create a new chart. Note that only one build configuration can be currently added as the data source. More configurations can be added manually.
- On the **Parameters** tab of the [build results](#) page, the list of **Reported statistic values** provides checkboxes to select the statistics type for a new [project- or build-configuration-level](#) chart.
 - A project-level chart will be added to the selected target project. The [root project](#) cannot be selected as the target.
 - A build-configuration-level chart will be added to all build configurations of the selected target project and its subprojects. Specifying the [root project](#) as the target will add the chart to all build configurations available on the server.

Modifying Custom Charts

Use the pencil  icon to edit or delete a custom chart. Note that the **Add Statistic Values** drop-down displays all statistic values registered on the server. If you select a value non-existent in the current build configuration or project when editing a chart, the chart will not be saved.

Using the cog  icon, you can also configure the Y-axis settings and save them as defaults for all users.

 There is a number of [limitations](#) to editing charts from the TeamCity UI.

Reordering Custom Charts

Since TeamCity 9.0.2, you can reorder the charts for a project, since TeamCity 9.1 the same applies to build configuration charts.

To reorder custom charts for a project/build configuration, click the **Reorder** button and drag-and-drop the charts to arrange them as required and apply your changes.

Managing Custom Charts Manually

To manually create custom charts to be displayed in the TeamCity web UI, configure the `<TeamCity Data Directory>/config/projects/<ProjectID>/pluginData/plugin-settings.xml` following the procedure below.

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 `<TeamCity Data Directory>/config/projects/<ProjectID>/pluginData/plugin-settings.xml`:

- For Project-level chart: use tag `<project-graphs>`
- For Build Configuration-level chart: use tag `<buildType-graphs>`

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>
    <project-graphs>
        <graph title="Duration comparison" hideFilters="showFailed" seriesTitle="Some key" format="duration">
            <valueType key="BuildDuration" title="duration1" sourceBuildTypeId="my_first_configuration_id"/>
            <valueType key="BuildDuration" title="duration2" sourceBuildTypeId="my_second_configuration_id"/>
            <valueType key="customKey" title="Custom data" color="#ee0055" /> <!-- Will use data from build configuration my_second_configuration_id -->
        </graph>
    </project-graphs>
</settings>
```

This "Duration comparison" chart will only be shown on Statistics tab of the project where the `plugin-settings.xml` file is located

Custom build configuration-level charts in `plugin-settings.xml`

```

<settings>
  <buildtype-graphs>
    <graph title="Passed Test Count" seriesTitle="Configuration">
      <valueType key="PassedTestCount" title="This configuration" />
      <valueType key="PassedTestCount" title="Passed Test Count" />
    sourceBuildTypeId="my_configuration_id"/> <!-- 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>
  </buildtype-graphs>
</settings>

```

These charts will be shown on Statistics tabs of the Build Types of the project where the `plugin-settings.xml` file is located and all it's subprojects. In order to show a chart in all Build Types register it in the [Root Project](#).

 Note that when adding custom charts, the intermediate `project-graphs` or `buildType-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 <code><key></code> .
sourceBuildTypeId	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>sourceBuildTypeId</code> is set within the <code>valueType</code>, the data is taken from this build configuration even if it belongs to a different project. if the <code>sourceBuildTypeId</code> is not set within current the <code>valueType</code>, but it is set in the <code>valueType</code> above the current one within the chart, the data from the build configuration referenced above will be taken. See example for the <code>plugin-settings.xml</code> file above. if the <code>sourceBuildTypeId</code> is not set within current the <code>valueType</code> and is not set above, the chart will show data for the current build configuration, i.e. this chart will work only for build configurations.
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.

<valueTypes> : allows to show several series on the chart by a pattern

Attribute	Description
pattern	Pattern for names of the Value Types (or series) to be shown on the chart. * sign is allowed to filter Value Types. Value Type 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 Value Type key. Pattern group markers could be used - eg {1} stands for the first captured group in the pattern, {0} stands for the whole pattern.
sourceBuildTypeId	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>sourceBuildTypeId</code> is set within the <code>valueType</code>, the data is taken from this build configuration even if it belongs to a different project. if the <code>sourceBuildTypeId</code> is not set within current the <code>valueType</code>, but it is set in the <code>valueType</code> above the current one within the chart, the data from the build configuration referenced above will be taken. See example for the <code>plugin-settings.xml</code> file above. if the <code>sourceBuildTypeId</code> is not set within current the <code>valueType</code> and is not set above, the chart will show data for the current build configuration, i.e. this chart will work only for build configurations.

```

<graph title="Servers response time" seriesTitle="Server">
  <valueTypes pattern="Server:*" title="{1} response time" />
</graph>
<graph title="Build stages time" seriesTitle="Server">
  <valueTypes pattern="buildStageDuration:*" title="Stage: {1}" />
</graph>

```

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
axis.y.type	Logarithmic for the logarithmic Y axis scale, linear for the standard scale. The default is linear .
axis.y.includeZero	Whether the zero value is included on the Y axis (true) or not (false). The default is true .
axis.y.min	An integer value to start the Y axis from.
axis.y.max	An integer value to use as the maximum for the Y axis value .

Default Statistics Values Provided by TeamCity

The table below lists the predefined value providers that can be used to configure a custom chart. The values reported for each build differ depending on your build configuration settings.

You can view the all statistic values reported by the build on the **Build Results|Parameters|Reported statistic values** tab. For each of the values, a statistics chart is available on clicking the *View Trend* icon .

Key	Description	Unit
ArtifactsSize	The sum of all artifact file sizes in the artifact directory	Bytes
VisibleArtifactsSize	The sum of all artifact file sizes excluding hidden artifacts (those placed under <code>.teamcity</code> 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
TotalTestCount	The total number of tests in the build	int
PassedTestCount	The number of successfully passed tests in the build	int
FailedTestCount	The number of failed tests in the build	int
IgnoredTestCount	The number of ignored tests in the build	int
InspectionStatsE	The number of inspection errors in the build	int
InspectionStatsW	The number of inspection warnings in the build	int
SuccessRate	An indicator whether the build was successful	0 - failed, 1 - successful
TimeSpentInQueue	How long the build was queued	Milliseconds

Custom Build Metrics

If the predefined build metrics do not cover your needs, you can report custom metrics to TeamCity from your build script and use them to create a custom chart. There are two ways to report custom metrics to TeamCity:

- using [service messages](#) from your build,
- or using the [teamcity-info.xml](#) file.

Note that custom value keys should be unique and should not interfere with value keys predefined by TeamCity.

See also:

[Concepts: Code Coverage | Code Inspection | Code Duplicates](#)

[User's Guide: Statistic Charts](#)

[Extending TeamCity: Build Script Interaction with TeamCity | Custom Statistics](#)

Edit Custom Chart Limitations

Currently editing a custom chart from the TeamCity UI is limited:

- Only one source Build Configuration could be assigned to a chart
- It is impossible to move a chart to another project
- A chart cannot be created if it contains no data
- A statistic value can be added only if it contains some data
- The width and height of a chart are not configurable
- A patterned statistic value cannot be added or removed.

Developing TeamCity Plugins

TeamCity 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 need more information or have a question regarding the API, please do not hesitate to post your question into [TeamCity Plugins forum](#). Please use search before posting to avoid possible duplication of discussions.

Please refer to corresponding section for further details.

- [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
- [Bundled Development Package](#)
- [Open API Changes](#)
- [Plugin Types in TeamCity](#)
- [Plugins Packaging](#)
- [Server-side Object Model](#)
- [Agent-side Object Model](#)
- [Extensions](#)
- [Web UI Extensions](#)
- [Plugin Settings](#)
- [Development Environment](#)
- [Developing Plugins Using Maven](#)
- [Plugin Development FAQ](#)
- [Getting Started with Plugin Development](#)

Plugin Quick Start

See [Getting Started with Plugin Development](#) to create your first plugin with Maven. [Developing Plugins Using Maven](#) provides more details.

There are also several approaches to create plugins provided by third parties or existing out of the main TeamCity development line:

- [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.

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.

On this page:

- Server-side part of the runner
- Agent-side part of the runner
- Extending the Ant runner
- Your Build Runner Results in TeamCity
 - Build log

- Artifacts
- Reports
 - XML Report processing
 - HTML Report processing

A build runner can have various settings which must be edited by the user in the web UI and passed to the agent. These settings are called **runner parameters** (or **runner properties**) and provided as a Map<String, String> to the agent part of the runner.

 Hint: some build runners whose source code can be used as a reference:

- Rake Runner
- FxCop runner sources
- Other build runner plugins.

Server-side part of the runner

The main entry point for the runner on the server side is **jetbrains.buildServer.serverSide.RunType**. A build runner plugin must provide its' own RunType and register it in the **jetbrains.buildServer.serverSide.RunTypeRegistry**.

RunType has a **type** which must be unique among all build runners and correspond to the **type** returned by the agent-side part of the runner (see **jetbrains.buildServer.agent.AgentBuildRunnerInfo**).

The **getEditRunnerParamsJspFilePath** and **getViewRunnerParamsJspFilePath** methods return paths to JSP files for editing and viewing runner settings. These JSP files must be bundled with plugin in **buildServerResources** subfolder, [read more](#). The paths should be relative to the **buildServerResources** folder.

 Since TeamCity 5.1, the path to the build runner resources files should be a full path without context. This path could be either a path to a .jsp file or a path that is handled by a controller. The plugin class may use **PluginDescriptor#getPluginResourcesPath()** method to create a path to a .jsp file from the buildServerResources folder of the plugin.

 TeamCity 5.0.x and earlier uses the following rule to compute a full path to the runner's jsp:

```
<context path>/plugins/<runType>/<returned jsp path>
```

 Hint: before writing your own JSP for a custom build runner, take a look at the JSP files of the existing runners bundled with TeamCity.

When a user fills in your runner settings and submits the form, **jetbrains.buildServer.serverSide.PropertiesProcessor** returned by the **getRunnerPropertiesProcessor** method will be called. This processor will be able to verify user settings and indicate which of them are invalid.

Usually a JSP page is simple and does not provide much controls except for fields, checkboxes and so on. But if you need more control on how the page is processed on the server side, then you should register your own extension to the runner editing controller: **jetbrains.buildServer.controllers.admin.projects.EditRunTypeControllerExtension**.

And finally if you need to prefill some settings with default values, you can do this with the help of the **getDefaultRunnerProperties** method.

Agent-side part of the runner

The main interface for agent-side runners is **jetbrains.buildServer.agent.AgentBuildRunner**. However, if your custom runner runs an external process, it is simpler to use the following classes:

1. **jetbrains.buildServer.agent.runner.CommandLineBuildServiceFactory**
2. **jetbrains.buildServer.agent.runner.CommandLineBuildService**
3. **jetbrains.buildServer.agent.runner.BuildServiceAdapter**

You should implement the **CommandLineBuildServiceFactory** factory interface and make your class a Spring bean. The factory also provides some meta information about the runner via **jetbrains.buildServer.agent.AgentBuildRunnerInfo**.

CommandLineBuildService is an abstract class which simplifies external processes launching and allows listening for process events (output, finish and so on). Your runner should extend this class. Since TeamCity 6.0, we introduced the **jetbrains.buildServer.agent.runner.BuildServiceAdapter** class that extends **CommandLineBuildService** and provides utility methods to access build and runner context parameters.

AgentBuildRunnerInfo has two methods: **getType** which must return the same **type** as the one returned by the server-side part of the plugin, and **canRun** which is called to determine whether the custom runner can run on the agent (in the agent environment).

If the command line build service is not suitable for your needs, you can still implement the **AgentBuildRunner** interface and define it in the Spring context. Then it will be loaded automatically.

Extending the Ant runner

The TeamCity Ant runner, while being a plugin itself, can also be extended with the help of **jetbrains.buildServer.agent.ant.AntTaskExtension**. This extension works in the same JVM where Ant is running. Using this extension, you can watch for Ant tasks, modify/patch them and log various messages to the build log.

Your class implementing **AntTaskExtension** interface must be defined in the Spring bean and it will be picked up by the Ant runner automatically. You need to add a dependency to <teamcity>/webapps/ROOT/WEB-INF/plugins/ant/agent/antPlugin.zip!antPlugin/ant-runtime.jar jar.

Your Build Runner Results in TeamCity

Build log

Usually a build runner starts an external process, and logging is performed from that process. The simplest way to log messages in this case is to use **service messages**, [read more](#). In brief, a service message is a specially formatted text with attributes; when such text is logged to the process output, it is parsed and the associated processing is performed. With the help of these messages you can create a TeamCity hierarchical build log, report tests, errors and so on.

If an external process launched by your runner is Java and you can't use service messages, it is possible to obtain **jetbrains.buildServer.agent.BuildProgressLogger** in the class running in this JVM. For this, the following jar files must be added in the classpath of the external Java process: runtime-util.jar, server-logging.jar. Then you should use the **jetbrains.buildServer.agent.LoggerFactory** method to construct the logger: LoggerFactory.createBuildProgressLogger(parentClassLoader). Since this way is more involved, it is recommended to use service messages instead.

If logging of the messages is done in the agent JVM (not from within the external process started by your runner), you can obtain **jetbrains.buildServer.agent.BuildProgressLogger** from the **jetbrains.buildServer.agent.AgentRunningBuild#getBuildLogger** method.

Artifacts

You can instruct your build runner to publish the resulting artifacts to TeamCity using **service messages**. Note that artifacts are uploaded to the TeamCity server in the background, so to verify that your artifacts are uploaded, you'll have to wait until your build is finished.

Reports

XML Report processing

If your runner reports build results in a format supported by TeamCity, they can be displayed in the TeamCity web UI on the Build Results page. There are two ways to approach this:

- using the [XML Report Processing](#) build feature
- via [service messages](#)

HTML Report processing

If your build runner produces some static HTML content, it can be displayed in the TeamCity web UI. Configure a custom [report tab](#) to show the results on a project or build level.

Risk Tests Reordering in Custom Test Runner

In TeamCity, you can instruct the system to [run risk group tests before any others](#).

To implement the risk group tests reordering feature for your own custom test runner, TeamCity provides the following special 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 is specified, they are separated with a comma. This property is provided only if corresponding settings are selected on the build runner page.
- **teamcity.tests.recentlyFailedTests.file** — this property value contains the full path to a file with the recently failed tests. The property is provided only if the **recentlyFailed** group is selected. The file contains tests separated by a new line. For Java-like tests, full class names are stored in the file (without the test method name). In other cases, the full name of the test will be stored in the file as it was reported by the tests runner.
- **teamcity.build.changedFiles.file** — this property is useful if you want to support running of new and modified tests in your tests runner. This property contains the full path to a file with the information about changed files included in the build. You can use this file to determine whether any tests were modified and run them before others. The file contains new-line separated files: each line corresponds to one file and has the following format:

```
<relative file path>:<change type>:<revision>
```

where:

- <relative file path> is the path to a file relative to the current checkout directory.
 - <change type> is a type of modification and can have the following values: CHANGED, ADDED, REMOVED, NOT_CHANGED, DIRECTORY_CHANGED, DIRECTORY_ADDED, DIRECTORY_REMOVED
 - <revision> is a file revision in the repository. If the file is a part of change list started via the [remote run](#), then the <personal> string will be written instead of the file revision.
- **teamcity.build.checkoutDir** — this property contains the path to the build checkout directory. It is useful if you need to convert relative paths to modified files to absolute ones.



TeamCity will pass the **teamcity.tests.runRiskGroupTestsFirst**, **teamcity.tests.recentlyFailedTests.file** and **teamcity.build.changedFiles.file** properties to the build process, but if the process starts an additional JVM or other processes, these properties won't be passed to them automatically.

For example, if you are using an Ant runner, you will have access to these properties from the Ant build.xml. But if your build.xml starts a new JVM (or <junit/> task with `fork="yes"` attribute), and you want to access these properties from this JVM, you'll have to modify your build script and pass them explicitly.

Known Limitations

If you have a package specified in the `TestNG` xml suite, reordering will not work: in this case TestNG itself searches for classes in packages and TeamCity cannot affect the way it sorts these classes. However, reordering will work if you specify concrete Test classes in the xml suite. Also, if you have several xml suites, reordering will work on the per-suite basis.

Having single classes in the XML suite may impose some inconveniences, e.g. developers have to remember to include classes in the suites. At the same time, this should speedup the tests startup, as the process of the searching classes by package is not that fast.

Custom Build Trigger

An example of a trigger plugin can be found in [Url Build Trigger](#).

Build Trigger Service

Build trigger is a service whose purpose is to trigger builds (add builds to the queue). Build trigger must extend **jetbrains.buildServer.buildTriggers.BuildTriggerService** abstract class. Build trigger service is uniquely identified by trigger name (see `getName` method). There is no need to register `BuildTriggerService`, instead plugin should provide a class extending the **jetbrains.buildServer.buildTriggers.BuildTriggerService** defined as a Spring bean.

Build Trigger Settings

Build trigger settings is an object containing build trigger parameters specified by a user via the web UI. Build trigger settings are represented by **jetbrains.buildServer.buildTriggers.BuildTriggerDescriptor** class. The settings are contained within a map of string parameters. More than one instance of `jetbrains.buildServer.buildTriggers.BuildTriggerDescriptor` corresponding to the same trigger service can be added to the build configuration or a template. Instances of the `jetbrains.buildServer.buildTriggers.BuildTriggerDescriptor` with the same trigger name and parameters are considered equal. With help of `jetbrains.buildServer.buildTriggers.BuildTriggerService#isMultipleTriggersPerBuildTypeAllowed()` trigger service can allow or disallow multiple trigger settings per build configuration.

Each trigger service can provide an URL to a jsp or custom controller which will show the trigger web UI. The approach is similar to the one used for VCS roots and build runners.

Triggering Policy

Build trigger service must return a policy (`jetbrains.buildServer.buildTriggers.BuildTriggeringPolicy`) which will be used to add builds to the queue. Currently only one policy is available: `jetbrains.buildServer.buildTriggers.PolledBuildTrigger`. More policies can be added in the future.

Trigger returning `jetbrains.buildServer.buildTriggers.PolledBuildTrigger` policy will be polled by the server with regular intervals. Trigger will receive `jetbrains.buildServer.buildTriggers.PolledTriggerContext` object which contains all information necessary to make a decision whether a build must be triggered or not. Trigger can use `jetbrains.buildServer.serverSide.SBuildType#addToQueue(java.lang.String)` method to add builds to the queue. Note that `jetbrains.buildServer.buildTriggers.PolledTriggerContext` also provides access to the custom data storage. This storage can be used for build trigger state associated with a build configuration and trigger settings. Custom storage will be automatically persisted and restored upon server restart.

Extending Notification Templates Model

You can extend data model passed into [notification templates](#) when evaluating.

In your plugin, implement [jetbrains.buildServer.notification.TemplateProcessor](#) interface. The following example can be found in our sample plugin:

```
public class SampleTemplateProcessor implements TemplateProcessor {  
    public SampleTemplateProcessor() {}  
  
    @NotNull  
    public Map<String, Object> fillModel(@NotNull NotificationContext context) {  
        Map<String, Object> model = new HashMap<String, Object>();  
        model.put("users", context.getUsers());  
        model.put("event", context.getEventType());  
        return model;  
    }  
}
```

Issue Tracker Integration Plugin

TeamCity offers [integration](#) with several issue trackers and a custom plugin can provide support for other systems.

Issue tracker integration

To create a TeamCity plugin for custom issue tracking system (ITS), you have to implement the following interfaces (all from `jetbrains.buildServer.issueTracker` package):

- [jetbrains.buildServer.issueTracker.SIssueProvider](#) - represents a single provider
- [jetbrains.buildServer.issueTracker.IssueProviderFactory](#) - API for instantiation of issue tracker providers

The main entity is a *provider* (i.e. connection to the ITS), responsible for parsing, extracting and fetching issues from the ITS.

Here is a brief description of the strategy used in TeamCity in respect to ITS integration:

When the server is going to render the user comment (VCS commit, or build comment), it invokes all registered providers to parse the comment. This operation is performed by the `IssueProvider.getRelatedIssues()` method, which analyzes the comment and returns the list of the issue mentions ([jetbrains.buildServer.issueTracker.IssueMention](#)). `IssueMention` just holds the information that is enough to render a popup arrow near the issue id. When the user points the mouse cursor on the arrow, the server requests the full data for this issue calling `IssueProvider.findIssueById()` method, and then displays the data in a popup. The data can be taken from the provider's cache.

The provider has a number of parameters, configured from admin UI. These parameters are passed using the properties map (a map string -> string). Commonly used properties include provider name, credentials to communicate with ITS, or regular expression to parse issue ids. You don't have to worry about storing the properties in XML files, server does that.

Provider registration is done by the TeamCity administrator in the web UI, and the responsibility for it lies mostly on TeamCity server. The plugin must only provide a JSP used for creation/editing of the provider (see details below).

Plugin development overview

A brief summary of steps to be done to create and add a plugin to TeamCity.

- Implement factory and provider interfaces ([jetbrains.buildServer.issueTracker.SIssueProvider](#) and [jetbrains.buildServer.issueTracker.IssueProviderFactory](#))
- Create a JSP page for admin UI
- [Install](#) the plugin (to `.BuildServer/plugins`)

Reusing default implementation

Common code of Jira, Bugzilla and YouTrack plugins can be found in `Abstract*` classes in the same package:

- [jetbrains.buildServer.issueTracker.AbstractIssueProviderFactory](#)
- [jetbrains.buildServer.issueTracker.AbstractIssueProvider](#)
- [jetbrains.buildServer.issueTracker.AbstractIssueFetcher](#) - a helper entity which encapsulates fetch-related logic

`AbstractIssueProvider` implements a simple caching provider able to extract the issues from the string based on a regexp. In most cases you just need to derive from it and override few methods. A simple derived provider class can look like this:

```

public class MyIssueProvider extends AbstractIssueProvider {
    // Let's name the provider simple: "myName". The plugin name should be the same.
    public MyIssueProvider(@NotNull IssueFetcher fetcher) {
        super("myName", fetcher);
    }

    // Means that issues are in format "PREFIX-123", like in Jira or YouTrack.
    // The prefix is configured via properties, regexp is invisible for users.
    protected boolean useIdPrefix() {
        return true;
    }
}

```

Providers like Bugzilla might need to override `extractId` method, because the mention of issue id (in comment) and the id itself can differ. For instance, suppose the issues are referenced by a hash with a number, e.g. #1234; the regexp is "#(\d{4})" (configurable); but the issues in ITS are represented as plain integers. Then the provider must extract the substrings matching "#(\d{4})" and return the first groups only. You should implement it in `extractId` method:

```

@NotNull
protected String extractId(@NotNull String match) {
    Matcher matcher = myPattern.matcher(match);
    matcher.find();
    return matcher.group(1);
}

```

The factory code is very simple as well, for example:

```

public class MyIssueProviderFactory extends AbstractIssueProviderFactory {
    public MyIssueProviderFactory(@NotNull IssueFetcher fetcher) {
        // Type name usually starts with uppercase character because it is displayed in
        // UI, but not necessarily.
        super(fetcher, "MyName");
    }

    @NotNull
    public IssueProvider createProvider() {
        return new MyIssueProvider(myFetcher);
    }
}

```

`IssueFetcher` is usually the central class performing plugin-specific logic. You have to implement `getIssue` method, which connects to the ITS remotely (via HTTP, XML-RPC, etc), passes authentication, retrieves the issue data and returns it, or reports an error. Example:

```

public IssueData getIssue(@NotNull String host, @NotNull String id,
    @Nullable final Credentials credentials) throws Exception {
    final String url = getUrl(host, id);
    return getFromCacheOrFetch(url, new FetchFunction() {
        @NotNull
        public IssueData fetch() throws Exception {
            InputStream body = fetchHttpFile(url, credentials);
            IssueData result = null;
            if (body != null) {
                result = parseXml(body, url);
            }
            if (result == null) {
                throw new RuntimeException("Failed to fetch issue from " + url + ")");
            }
            return result;
        }
    });
}

```

You need to implement how to compose the server URL and how do you parse the data out of XML (HTML). `AbstractIssueFetcher` will take care about caching, errors reporting and everything else.

Plugin UI

The only mandatory JSP required by TeamCity is `editIssueProvider.jsp` (the full path must be `/plugins/myName/admin/editIssueProvider.jsp`, that is, the plugin should have the jsp available `/admin/editIssueProvider.jsp` of its resources). This JSP is requested when the user opens the dialog for editing (or creating) the issue provider. In most cases it just renders the provider properties, or returns the form for filling them.

You can see the example in `/plugins/yourtrack/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.CollectSingleStatePolicy#getCurrentVersion(jetbrains.buildServer.vcs.VcsRoot)`. The version here is an arbitrary text. It can be a representation of a transaction number, a revision number, a date, whatever suitable enough for getting a source snapshot in a particular VCS. Usually format of the version depends on a version control system, the only requirement which comes from TeamCity — it should be possible to sort changes by version in order of their happening (see `jetbrains.buildServer.vcs.VcsSupportConfig#getVersionComparator()`).

Version is used in several places:

- for changes collecting
- for patch construction
- when content of the file is retrieved from the repository
- for labeling / tagging

TeamCity does not show Versions in the UI directly. For UI TeamCity converts a Version to its display name using `jetbrains.buildServer.vcs.VcsSupportConfig#getVersionDisplayName(String,jetbrains.buildServer.vcs.VcsRoot)`.

A *Change* is an atomic modification of a single file within a source repository. In other words, a change corresponds to a single increment of a file version.

A *Modification* is a set of changes made by some user at a certain time interval. It most closely corresponds to a single checkin transaction (commit), when a user commits all his modifications made locally to the central repository. A Modification also contains the Version of the VCS Root right after the corresponding Changes have been applied.

A collection of Modifications is what TeamCity expects as a result when asking a VCS plugin for changes.

A *Patch* is a set of operations to convert the directory state from one modification to another (e.g. change/add/remove file, add/remove directory).

A *Checkout Rule* is a way of changing default file layout.

Checkout rules allow to map the path in repository to another path on agent or to exclude some parts of repository, [read more](#).

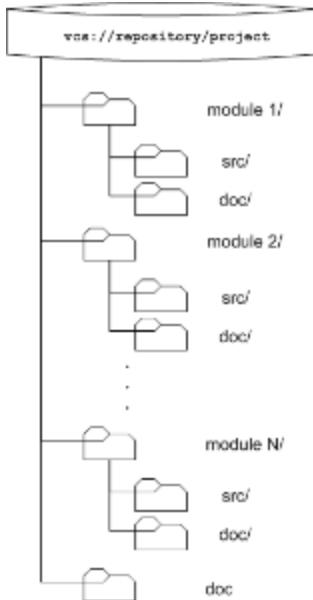
Checkout rules consist of include and exclude rules. Include rule can have "from" and "to" parts ("to" part allows to map path in repository to another path on agent). Mapping is performed by TeamCity itself and VCS plugin should not worry about it. However a VCS plugin can use checkout rules to speedup changes retrieval and patch building since checkout rules usually narrow a VCS Root to some its subset.

Server-Side Part

Patch Building and Change Collecting Policies

When implementing include rule policies it is important to understand how it works with Checkout Rules and paths. Let's consider an example with collecting changes.

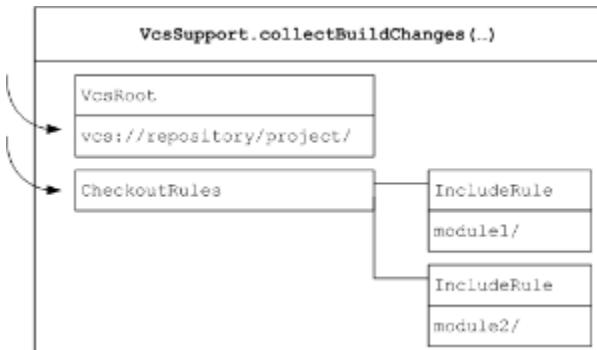
Suppose, we have a VCS Root pointing to `vcs://repository/project/`. The project root contains the following directory structure:



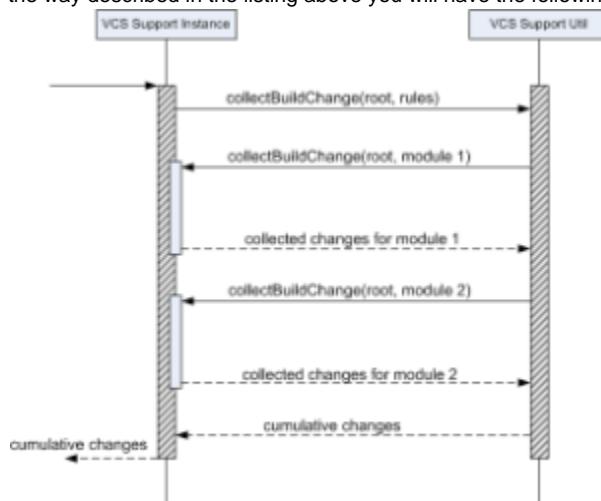
We want to monitor changes only in `module1` and `module2`. Therefore we've configured the following checkout rules:

```
\+:module1
\+:module2
```

When `collectBuildChanges(...)` is invoked it will receive a `VcsRoot` instance that corresponds to `vcs://repository/project/` and a `CheckoutRules` instance with two `IncludeRules` — one for "module1" and the other for "module2".



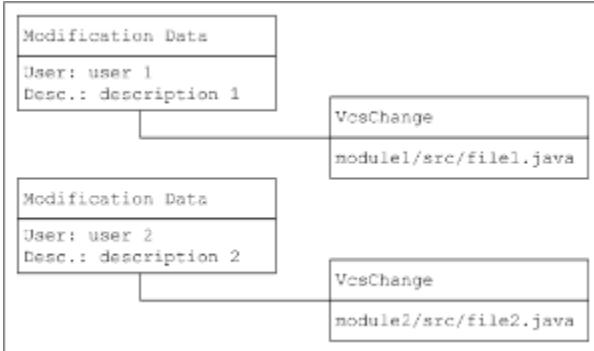
If `collectBuildChanges(...)` utilizes `VcsSupportUtil.collectBuildChanges(...)` it transforms the invocation into two separate calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. If you have implemented `CollectChangesByIncludeRule` in the way described in the listing above you will have the following interaction.



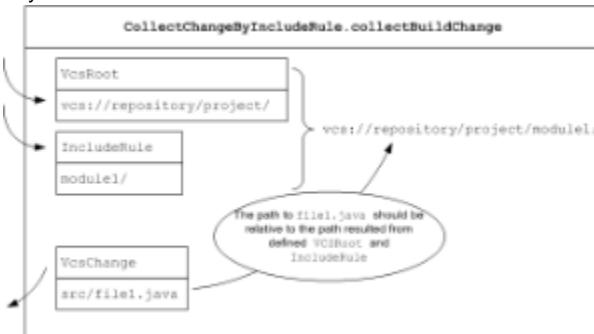
Now let's assume we've got a couple of changes in our sample repository, made by different users.



The collection of `ModificationData` returned by `VcsSupport.collectBuildChanges(...)` should then be like this:



But this is not a simple union of collections, returned by two calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. To see why let's have a closer look at the first call.

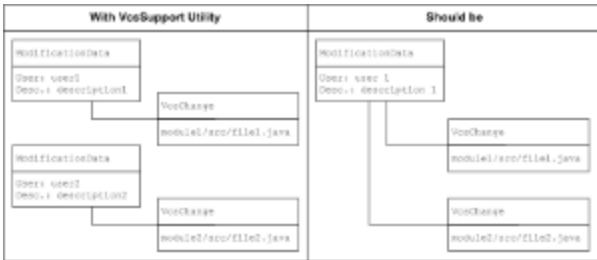


As you can see the paths in the resulting change must be relative to the path presented by the include rule, rather than the path in the VCS Root.

Then after collecting all changes for all the include rules `VcsSupportUtil` transforms the collected paths to be relative to the VCS Root's path.

Although being quite simple `VcsSupportUtil.collectBuildChanges(...)` has the following limitations.

Assume both changes in the example above are done by the same user within the same commit transaction. Logically, both corresponding `VcsChange` objects should be included into the same `ModificationData` instance. However, it's not true if you utilize `VcsSupportUtil.collectBuildChanges(...)`, since a separate call is made for each include rule.



Changes corresponding to different include rules cannot be aggregated under the same `ModificationData` instance even if they logically relate to the same commit transaction. This means a user will see these changes in separate change lists, which may be confusing. Experience shows that it's not very common situation when a user commits to directories monitored with different include rules. However, if the duplication is extremely undesirable an implementation should not utilize `VcsSupportUtil.collectBuildChanges(...)` and control `CheckoutRules` itself.

Another limitation is complete ignorance of exclude rules. As it said before this doesn't cause showing unneeded information in the UI. So an implementation can safely use `VcsSupportUtil.collectBuildChanges(...)` if this ignorance doesn't lead to significant performance problems. However, If an implementer believes the change collection speed can be significantly improved by taking into account include rules, the implementation must handle exclude rules itself.

All above is applicable to building patches using `VcsSupportUtil.buildPatch(...)` including the path relativity aspect.

Server-Side Caching

By default, the server caches clean patches created by VCS plugins, because clean patch construction can take significant time on large repositories. If clean patches created by your VCS plugin need not to be cached, you should return `true` from the method `VcsSupport#ignoreServerCachesFor(VcsRoot)`.

Agent-Side Part

Agent-Side Checkout

Agent part of VCS plugin is optional, if it is provided then checkout can also be performed on the agent itself. This kind of checkout usually works faster but it may require additional configuration efforts, for example, if VCS plugin uses command line client then this client must be installed on all of the agents.

To enable agent-side checkout, be sure to include `jetbrains.buildServer.agent.vcs.AgentVcsSupportContext` into agent plugin part and also enable agent-side checkout via `jetbrains.buildServer.vcs.VcsSupportConfig#isAgentSideCheckoutAvailable()`.

Version Control System Plugin (old style - prior to 4.5)

In TeamCity a plugin for Version Control System (VCS) is seen as an `jetbrains.buildServer.vcs.VcsSupport` instance. All VCS plugins must extend this class.

VCS plugin has a server side part and an optional agent side part. The server side part of a VCS plugin should support the following mandatory operations:

- collecting changes between versions
- building of a patch from version to version
- get content of a file (for web diff, duplicates finder, and some other places)

There are also optional parts:

- labeling / tagging
- personal builds

Personal builds require corresponding support in IDE. Chances are we will eliminate this dependency in the future.



You can use source code of the existing VCS plugins as a reference, for example:

- <http://www.jetbrains.net/confluence/display/TW/Mercurial>
- <http://www.jetbrains.net/confluence/display/TW/AccuRev>

Before digging into the VCS plugin development details it's important to understand the basic notions such as Version, Modification, Change, Patch, Checkout Rule, which are explained below.

Version

A *Version* is unambiguous representation of a particular snapshot within a repository pointed at by a VCS Root. The current version represents the head revision at the moment of obtaining.

The current version& is obtained from the **VcsSupport#getCurrentVersion(VcsRoot)**. The version here is arbitrary text. It can be transaction number, revision number, date and so on. Usually format of the version depends on a version control system, the only requirement which comes from TeamCity - it should be possible to sort changes by version in order of their appearance (see **VcsSupport#getVersionComparator()** method).

Version is used in several places:

- for changes collecting
- for patch construction
- when content of the file is retrieved from the repository
- for labeling / tagging

TeamCity does not show Versions in the UI directly. For UI, TeamCity converts a Version to its display name using **VcsSupport#getVersionDisplayName(String, VcsRoot)**.

Collecting Changes

A *Change* is an atomic modification of a single file within a source repository. In other words, a Change corresponds to a single increment of the file version.

A *Modification* is a set of Changes made by some user at a certain moment. It most closely corresponds to a single checkin transaction, when a user commits all his modifications made locally to the central repository. A Modification also contains the Version of the VCS Root right after the corresponding Changes have been applied.

TeamCity server polls VCS for changes on a regular basis. A VCS plugin is responsible for collecting information about Changes (grouped into Modifications) between two versions.

Once a VCS Root is created the first action performed on it is determining the current Version (**VcsSupport#getCurrentVersion(VcsRoot)**). This value is stored and used during the next checking for changes as the "from" Version (**VcsSupport#collectBuildChanges(VcsRoot, String, String, CheckoutRules)**). The current Version is obtained again to be used as the "to" Version. The Modifications collected are then shown as pending changes for corresponding build configurations. After the checking for changes interval passes the server requests for next portion of changes, but this time the "from" Version is replaced with the previous "current" Version. And so on.

Obtaining the current Version may be an expensive operation for some version control systems. In this case some optimization can be done by implementing interface **CurrentVersionIsExpensiveVcsSupport**. Its method **CurrentVersionIsExpensiveVcsSupport#collectBuildChanges(VcsRoot, String, 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.patches.PatchBuilder**.

When a build is about to start, the server determines for which Versions the patch is to be constructed and passes them to **VcsSupport#buildPatch(VcsRoot, String, String, PatchBuilder, CheckoutRules)**.

There are two types of patch: clean patch (if fromVersion is null) and incremental patch (if fromVersion is provided). Clean patch is just an export of files on the specified version, while incremental patch is a more complex thing. To create incremental patch you need to determine the difference between two snapshots including files and directories creations/deletions.

Checkout Rules

Checkout rules allow to map path in repository to another path on agent or to exclude some parts of repository, [read more](#).

Checkout rules consist of include and exclude rules. Include rule can have "from" and "to" parts ("to" part allows to map path in repository to another path on agent). Mapping is performed by TeamCity itself and VCS plugin should not worry about it. However a VCS plugin can use checkout rules to speedup changes retrieval and patch building since checkout rules usually narrow a VCS Root to some its subset.

In most cases it is simpler to collect changes or build patch separately by each include rule, for this VCS plugin can implement interface **jetbrains.buildServer.CollectChangesByIncludeRule** (as well as **jetbrains.buildServer.vcs.BuildPatchByIncludeRule**) and use **jetbrains.buildServer.vcs.VcsSupportUtil** as shown below:

```

public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion,
String currentVersion, CheckoutRules checkoutRules)
    throws VcsException {
    return VcsSupportUtil.collectBuildChanges(root, fromVersion, currentVersion,
checkoutRules, this);
}

public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion,
String currentVersion, IncludeRule includeRule)
    throws VcsException {
    ... changes collecting code ...
}

```

And for patch construction:

```

public void buildPatch(VcsRoot root, String fromVersion, String toVersion,
PatchBuilder builder, CheckoutRules checkoutRules)
    throws IOException, VcsException {
    VcsSupportUtil.buildPatch(root, fromVersion, toVersion, builder, checkoutRules,
this);
}

public void buildPatch(VcsRoot root, String fromVersion, String toVersion,
PatchBuilder builder, IncludeRule includeRule)
    throws IOException, VcsException {
    ... build patch code ...
}

```

If you want to share data between calls, this approach allows you to do it easily using anonymous classes:

```

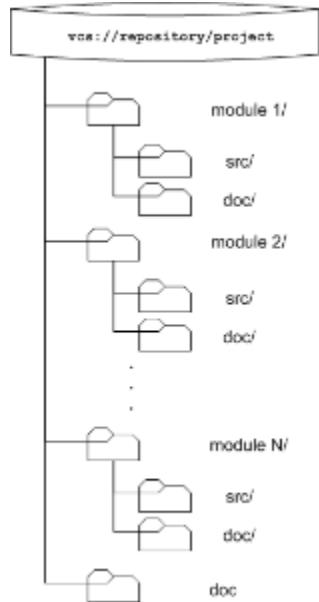
public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion,
String currentVersion, CheckoutRules checkoutRules)
    throws VcsException {
    final MyConnection conn = obtainConnection(root); // get a connection to the
repository
    return VcsSupportUtil.collectBuildChanges(root, fromVersion, currentVersion,
checkoutRules, new CollectChangesByIncludeRule {
        public List<ModificationData> collectBuildChanges(VcsRoot root, String
fromVersion, String currentVersion, IncludeRule includeRule)
            throws VcsException {
            doCollectChange(conn, includeRule); // use the same connection for all calls
        }
    });
}

public List<ModificationData> collectBuildChanges(VcsRoot root, String fromVersion,
String currentVersion, IncludeRule includeRule)
    throws VcsException {
    ... changes collecting code ...
}

```

When using `VcsSupportUtil` it is important to understand how it works with Checkout Rules and paths. Let's consider an example with collecting changes.

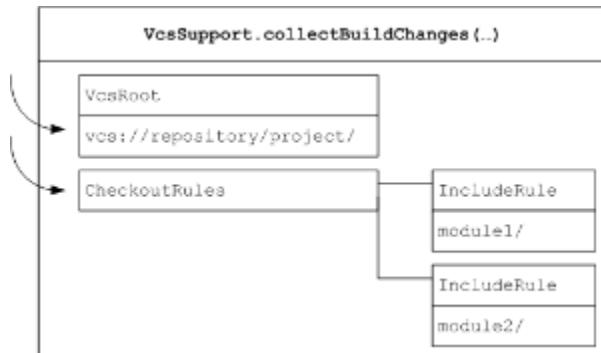
Suppose, we have a VCS Root pointing to `vcs://repository/project/`. The project root contains the following directory structure:



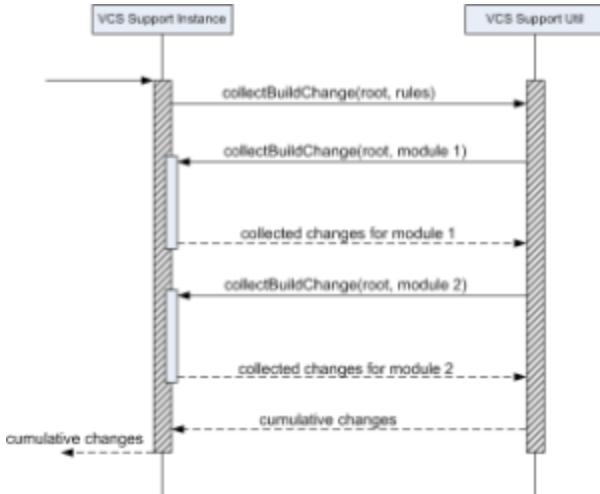
We want to monitor changes only in `module1` and `module2`. Therefore we've configured the following checkout rules:

```
\+:module1  
\+:module2
```

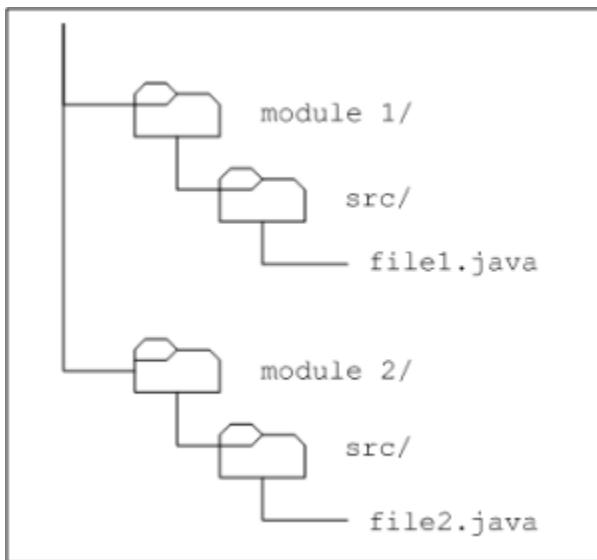
When `collectBuildChanges(...)` is invoked it will receive a `VcsRoot` instance that corresponds to `vcs://repository/project/` and a `{CheckoutRules` instance with two `IncludeRules` — one for "module1" and the other for "module2".



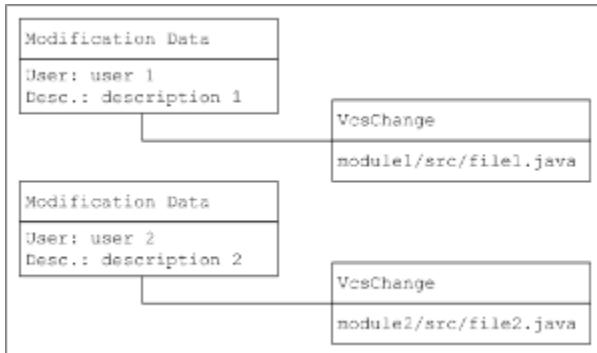
If `collectBuildChanges(...)` utilizes `VcsSupportUtil.collectBuildChanges(...)` it transforms the invocation into two separate calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. If you have implemented `CollectChangesByIncludeRule` in the way described in the listing above you will have the following interaction.



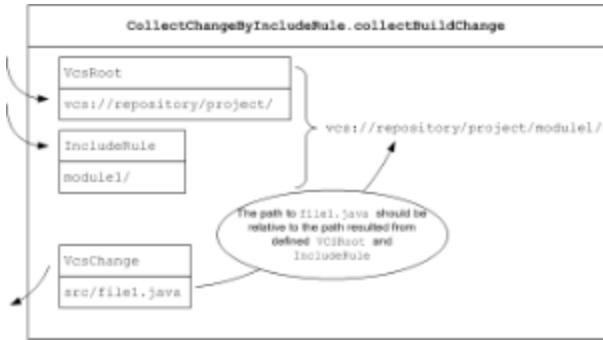
Now let's assume we've got a couple of changes in our sample repository, made by different users.



The collection of ModificationData returned by VcsSupport.collectBuildChanges(...) should then be like this:



But this is not a simple union of collections, returned by two calls of CollectChangesByIncludeRule.collectBuildChange(...). To see why let's have a closer look at the first calls.

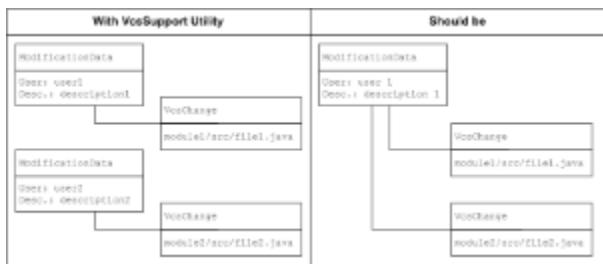


As you can see the paths in the resulting change must be relative to the path presented by the include rule, rather than the path in the VCS Root.

Then after collecting all changes for all the include rules `VcsSupportUtil` transforms the collected paths to be relative to the VCS Root's path.

Although being quite simple `VcsSupportUtil.collectBuildChanges(...)` has the following limitations.

Assume both changes in the example above are done by the same user within the same commit transaction. Logically, both corresponding `VcsChange` objects should be included into the same `ModificationData` instance. However, it's not true if you utilize `VcsSupportUtil.collectBuildChanges(...)`, since a separate call is made for each include rule.



Changes corresponding to different include rules cannot be aggregated under the same `ModificationData` instance even if they logically relate to the same commit transaction. This means a user will see these changes in separate change lists, which may be confusing. Experience shows that it's not very common situation when a user commits to directories monitored with different include rules. However if the duplication is extremely undesirable an implementation should not utilize `VcsSupportUtil.collectBuildChanges(...)` and control `CheckoutRules` itself.

Another limitation is complete ignorance of exclude rules. As it said before this doesn't cause showing unneeded information in the UI. So an implementation can safely use `VcsSupportUtil.collectBuildChanges(...)` if this ignorance doesn't lead to significant performance problems. However, If an implementer believes the change collection speed can be significantly improved by taking into account include rules, the implementation must handle exclude rules itself.

All above is applicable to building patches using `VcsSupportUtil.buildPatch(...)` including the path relativity aspect.

Registering In TeamCity

During the server startup all VCS plugins are required to register themselves in the VCS Manager (`jetbrains.buildServer.vcs.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. e.g. a derivative id will then be used
        // in "auth-config.xml" file
        return "custom";
    }

    @Override
    public String getDisplayName() {
        // return presentable name of this plugin
        return "My custom authentication plugin";
    }

    @Override
    public String getDescription() {
        // return human-readable description of this module type
        return "Authentication via custom login module";
    }

    public Class<? extends LoginModule> getLoginModuleClass() {
        // return our custom login module class
        return CustomLoginModule.class;
    }

    @Override
    public Map<String, ?> getJAASOptions(final Map<String, String> properties) {
        // return options which can be passed to our custom login module
        // for example, we could store reference to SBuildServer instance here
        return null;
    }
}
```

Finally we should create `build-server-plugin-ourUserAuth.xml` and zip archive with plugin classes as it is described [here](#) and write there `CustomLoginModuleDescriptor` bean:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-autowire="constructor">
    <bean id="customLoginModuleDescriptor"
class="some.package.CustomLoginModuleDescriptor"/>
</beans>
```

Now you should be able to [change authentication scheme](#) to your authentication module.

HTTP Authentication Module

To provide your own HTTP authentication module you should provide a class which must implement the interface **jetbrains.buildServer.controllers.interceptors.auth.HttpAuthenticationScheme** and register it in the **jetbrains.buildServer.serverSide.auth.LoginConfiguration**.

To make the authentication module active its type name can then be used during [Configuring Authentication Settings](#).

For example:

CustomHttpAuthenticationScheme.java

```
public class CustomHttpAuthenticationScheme extends
jetbrains.buildServer.controllers.interceptors.auth.HttpAuthenticationSchemeAdapter {
    // Spring framework will provide reference to LoginConfiguration when
    // the CustomLoginModuleDescriptor is instantiated
    public CustomHttpAuthenticationScheme(final LoginConfiguration loginConfiguration) {
        // register this scheme in the login configuration
        loginConfiguration.registerAuthModuleType(this);
    }

    @Override
    protected String doGetName() {
        // return unique name of this module type. e.g. a derivative id will then be used
        in "auth-config.xml" file
        return "custom";
    }

    @Override
    public String getDisplayName() {
        // return presentable name of this plugin
        return "My custom HTTP authentication plugin";
    }

    @Override
    public String getDescription() {
        // return human-readable description of this module type
        return "Authentication via custom HTTP scheme";
    }

    // main method to process HTTP authentication
    @Override
    public HttpAuthenticationResult processAuthenticationRequest(final
HttpServletRequest request,
                                                               final
HttpServletResponse response,
                                                               final Map<String,
String> properties) throws IOException {
        if (!isAttemptToAuthenticateViaThisHTTPPScheme(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 NotifierPropertyKey(<notifier type>,
<property name>));
```

Notifier can also provide custom UI for [Notifier rules](#) and [My Settings&Tools](#) pages. See [PlaceId.NOTIFIER_SETTINGS_FRAGMENT](#) and [PlaceId.MY_SETTINGS_NOTIFIER_SECTION](#).

Notifications are only delivered if there is at least one subscribed user for given event.

 Use source code of the existing plugins as a reference:

- <http://code.google.com/p/buildbunny/wiki/CreateTeamcityNotifier>
- <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 the [Custom Chart](#) page.

On this page:

- [Customize TeamCity Statistics Page](#)
 - [Add Chart](#)
 - [Add Custom Content](#)
- [Add Statistics to your Custom Pages](#)
 - [Customize Chart Appearance](#)
- [Add Custom Build Metrics](#)

Customize TeamCity Statistics Page

Add Chart

To add a chart to the **Statistics** tab for a project or build configuration, use the `ChartProviderRegistry` bean:

```

public MyExtension(ChartProviderRegistry registry, ProjectManager manager) {
    registry.getChartProvider("project-graphs").registerChart(manager.findProjectByExternalId("externalId"), createXmlGraphBean());
    // "project-graphs" for Project Statistics Tab
    // "buildtype-graphs" for Build Configuration Statistics Tab
}

public XmlGraphBean createXmlGraphBean() {
    // creates XmlGraphBean
}

```

Add Custom Content

To add custom content to the **Statistics** tab for a project or build configuration, use the following example [here](#) and the appropriate PlaceId:

- for Build Configuration Statistics tab, use PlaceId.BUILD_CONF_STATISTICS_FRAGMENT
- for Project Statistics tab, use PlaceId.PROJECT_STATS_FRAGMENT.

Add Statistics to your Custom Pages

To add charts to your custom JSP pages, use the `<buildGraph>` tag and a special controller accessible on `"/buildGraph.html"`. It requires the `jsp` attribute leading to your page:

```

new ModelAndView("/buildGraph.html?jsp=" +
myDescriptor.getPluginResourcesPath("sampleChartPage.jsp"))

```

To insert statistics chart into the `sampleChartPage.jsp`:

```

<%@taglib prefix="stats" tagdir="/WEB-INF/tags/graph"%>
<stats:buildGraph id="g1" valueType="BuildDuration" />

```

Customize Chart Appearance

Attribute	Description	Usage
width, height	modify the chart image size	Integer value
hideFilters	suppress filter controls	Comma separated filters names: 'all', 'series', 'average', 'showFailed', 'range', 'yAxisType', 'forceZero' or 'yAxisRange'
defaultFilter	default filter state	Comma separated names: 'showFailed', 'averaged', 'logYAxis', 'autoscale'
hints	chart style	Set to 'rendererB' for a bar chart

Add Custom Build Metrics

To add a custom build metric, in addition to [the built-in methods](#), you can extend `BuildValueTypeBase` to define your build metric calculation method, appearance, and key. After that you can reference this metric by its key in statistics chart/report tags.

See also:

[Extending TeamCity: Build Script Interaction with TeamCity](#)

Custom Server Health Report

To report custom server health items, do the following:

- Create your own reporter
- Create a custom page extension to render items reported by you

Reporting Server Health Items

To make a reporter, create a subclass of [jetbrains.buildServer.serverSide.healthStatus.HealthStatusReport](#).

Particularly, you must override method [jetbrains.buildServer.serverSide.healthStatus.HealthStatusReport#report\(jetbrains.buildServer.serverSide.healthStatus.HealthStatusScope, jetbrains.buildServer.serverSide.healthStatus.HealthStatusItemConsumer\)](#).

The items should be reported according to the analysis scope passed as a parameter to this method using the appropriate method of resultConsumer.

If you try to consume an object which is not in the scope of the current analysis, it will be filtered out by the consumer and will not appear in the report.

This method is always called with system privileges (in all permissions mode). Any permissions checks should be avoided here.

While reporting items, additional data required for further items presentation could be provided.

Presenting Server Health Items to User

To present reported items, provide a [custom page extension](#) and connect it to 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!

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 the `build.properties` file and set the value for `path.variable.teamcitydistribution` property to the path of `<TeamCity Home Directory>`
- Run `ant dist` in the plugin directory (Ant 1.7+ is recommended). The plugin distribution should be created in the `dist` directory.

Building sample plugin in IntelliJ IDEA

- Unpack `<TeamCity Home Directory>\devPackage\samplePlugin-src.zip` into a directory of your choice
- Open the project in IDEA (the `.idea` project should work OK in IntelliJ IDEA 9 and later (including IntelliJ IDEA 9.0 Community Edition))
- On prompt to add the path variable, set the "TeamCityDistribution" path variable to the directory where TeamCity with `devPackage` is installed (`<TeamCity Home Directory>`).
- Open Project Structure and ensure you have Project SDK with name "1.6" pointing to Sun JDK version 1.6

Running the server with plugin from IDEA

- Either edit the `build.properties` file to set the `path.variable.teamcitydistribution` property or regenerate the build script from IDEA (execute "Generate Ant Build" with the settings: single file, all other options unchecked).

If you use the Ultimate edition of IntelliJ IDEA, you can start TeamCity'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.

Open API Changes

Changes from 9.0 to 9.1

Issue Trackers integration API changes

- `IssueProviderType` has been extracted as a separate class. It serves as issue tracker integration descriptor and contains id, display name and URLs to controller and issue rendering pages
- `AbstractIssueProviderFactory` now takes `IssueProviderType` as an argument rather type as a String
To be compatible with 9.1, existing plugins must implement `IssueProviderType` and change the corresponding provider factory according to base class interface. [This change](#) in Github integration plugin can be taken as an example

Changes from 8.1.x to 9.0

Server API changes

- `jetbrains.buildServer.serverSide.dependency.Dependent#getDependencyReferences` and `jetbrains.buildServer.serverSide.dependency.Dependent#getNumberOfDependencyReferences` were moved to `jetbrains.buildServer.serverSide.SBuildType`
- `jetbrains.buildServer.issueTracker.IssueProviderFactory#getDisplayName` display name for issue tracker in UI.

Changes from 7.1.x to 8.0

External ID -related changes

- `jetbrains.buildServer.serverSide.dependency.DependencyFactory#createDependency` now accepts external ID instead of internal one, use `jetbrains.buildServer.serverSide.dependency.DependencyFactory#createDependencyByInternalId` for internal ID.
- `jetbrains.buildServer.serverSide.ArtifactDependencyFactory#createArtifactDependency` now accepts external ID

instead of internal one, use `jetbrains.buildServer.serverSide.ArtifactDependencyFactory#createArtifactDependencyByInternalId` for internal ID.

Common API changes

- `SimpleCommandLineProcessRunner.RunCommandEvent` => `SimpleCommandLineProcessRunner.ProcessRunCallback`
- added `jetbrains.buildServer.serverSide.CachePaths` for plugins to get cache directory on server
- `jetbrains.buildServer.serverSide.statistics.ValueType#getFormat` now returns String constant representing format style
- `jetbrains.buildServer.serverSide.statistics.ValueType#getColor` now returns String containing Web Color

Server API changes

- Added `jetbrains.buildServer.serverSide.SProject#getPluginDataDirectory` that returns per-project plugin data directory
- `jetbrains.buildServer.serverSide.BuildTypeSettings#addBuildRunner` not accepts `jetbrains.buildServer.serverSide.BuildRunnerDescriptor` instead of `*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.PatchBu`

- `jetbrains.buildServer.vcs.patches.PatchBuilderBase` All methods from `jetbrains.buildServer.vcs.patches.PatchBuilderBase` were moved into `jetbrains.buildServer.vcs.patches.PatchBuilder`
- `jetbrains.buildServer.vcs.patches.PatchBuilderEx#setTimeStamp` was removed, use `jetbrains.buildServer.vcs.patches.PatchBuilder#setLastModified`
- `jetbrains.buildServer.vcs.patches.PatchBuilder` code was covered with `@NotNull/@Nullable` annotations

Access to VCS Services

Vcs API is split into two parts: **VCS plugin api**, which is used to implement VCS services in TeamCity, and **VCS usage api**, or just **VCS API**, which is used to work with VCS services from within TeamCity.

- `jetbrains.buildServer.vcs.VcsManager#getAllVcs` is replaced with `jetbrains.buildServer.vcs.VcsManager#getAllVcsCore`
- Introduced `jetbrains.buildServer.vcs.VcsRootInstance#findService` method to obtain a `VcsService`
- `jetbrains.buildServer.vcs.VcsManager#getVcsUsernames` return type has changed from `VcsSupportContext` to `VcsSupportCore` in the key of the returned Map.

Changes from 7.0 to 7.1

- new API calls `AgentRunningBuild#stopBuild` and `AgentRunningBuild#getInterruptReason()`. (Those methods were in `AgentRunningBuildEx` since 6.5)
- Responsibility API changes:
 - Added:
 - `jetbrains.buildServer.responsibility.ResponsibilityEntry`
 - enum `RemoveMethod`
 - `jetbrains.buildServer.responsibility.ResponsibilityEntry`
 - `jetbrains.buildServer.serverSide.ResponsibilityInfo`
 - `jetbrains.buildServer.serverSide.ResponsibilityInfoData`
 - `jetbrains.buildServer.tests.TestResponsibilityData`
 - `getRemoveMethod()`
 - `jetbrains.buildServer.responsibility.ResponsibilityEntryFactory`
 - `createEntry(BuildType)`
 - `jetbrains.buildServer.responsibility.impl.BuildTypeResponsibilityEntryImpl`
 - `constructor(BuildType)`
 - `jetbrains.buildServer.web.functions.user.ResponsibilityFunctions`
 - `isUserResponsible(ResponsibilityEntry, User)`
 - Changed:
 - `jetbrains.buildServer.responsibility.impl.BuildTypeResponsibilityEntryImpl`
 - `constructor(BuildType, State, User, User, Date, String, RemoveMethod)`
 - `jetbrains.buildServer.responsibility.ResponsibilityEntryFactory`
 - `createEntry(BuildType, State, User, User, Date, String, RemoveMethod)`
 - `createEntry(TestName, long, State, User, User, Date, String, String, RemoveMethod)`
 - `jetbrains.buildServer.BuildType`
 - `getResponsibilityInfo()` now returns `ResponsibilityEntry`
 - `jetbrains.buildServer.serverProxy.RemoteBuildServer`
 - `updateResponsibility(Vector, String, String, String, String, String)`
 - Removed (deprecated):
 - `jetbrains.buildServer.serverSide.ResponsibilityInfo`
 - `createInactive()`
 - `createInactive(String, boolean, User)`
 - `getSince()`
 - `getUser()`
 - `getUserWhoPerformsTheAction()`
 - `isActive()`
 - `isFixed()`
 - `setUser(User)`
 - `jetbrains.buildServer.serverSide.ResponsibilityInfoData`
 - `isActive()`
 - `isFixed()`
 - `jetbrains.buildServer.BuildType`
 - `removeResponsible(boolean, User, String)`
 - `setResponsible(User, String)`
 - `jetbrains.buildServer.serverProxy.RemoteBuildServer`
 - `removeResponsible(String, boolean, String)`
 - `removeResponsible(String, boolean, String, String)`
 - `resetResponsible(String, String)`
 - `resetResponsible(Vector, String, boolean, String, String, String)`
 - `setIsFixed(String, String, String)`
 - `setResponsible(String, String, String)`
 - `setResponsible(String, String, String, String)`

- `setResponsible(Vector, String, String, String, String)`
- `jetbrains.buildServer.serverSide.BuildServerListener`
- `jetbrains.buildServer.serverSide.BuildServerAdapter`
 - `responsibleChanged(SBuildType, ResponsibilityInfo, ResponsibilityInfo, boolean)`
 - `jetbrains.buildServer.responsibility.SBuildTypeResponsibilityFacade`
 - `jetbrains.buildServer.responsibility.STestNameResponsibilityFacade`
- Removed:
 - `jetbrains.buildServer.serverSide.problems.BuildProblem` and all implementations
 - `jetbrains.buildServer.serverSide.problems.BuildProblemsProvider` and all implementations
 - `jetbrains.buildServer.serverSide.problems.BuildProblemsVisitor`
 - `jetbrains.buildServer.serverSide.SBuild`
 - `getBuildProblems()`
 - `visitBuildProblems(BuildProblemsVisitor)`
- JavaScript: Activator is now `BS.Activator` and its source file has been moved from `js/activation.js` to `js-bs/activation.js`

Changes from 6.5 to 7.0

- new API calls: `BuildStatistics.findTestBy(TestName)` and `BuildStatistics.getAllTests()`
- event-method `projectCreated` of `j.b.serverSide.BuildServerListener` and `j.b.serverSide.BuildServerAdapter` now receives two parameters: `projectId` and `user`.
- no longer publish `AntTaskExtension*`, `AntUtil`, `TestNGUtil`, `ElementPatch`, `JavaTaskExtensionHelper` classes to `openapi` package. Those classes can still be found in `<teamcity>/webapps/ROOT/WEB-INF/plugins/ant/agent/antPlugin.zip!antPlugin/ant-runtime.jar`
- Notificator interface: methods `notifyResponsibleChanged` and `notifyResponsibleAssigned` changed second parameter from `j.b.serverSide.ResponsibilityInfo` to `j.b.responsibility.ResponsibilityEntry` (due to `ResponsibilityInfo` deprecation).
- `j.b.serverSide.BuildServerListener` - we've deprecated `responsibleChanged` method which used `j.b.serverSide.ResponsibilityInfo` parameter and added a similar method which uses `j.b.responsibility.ResponsibilityEntry`
- new API calls: `j.b.agent.AgentRunningBuild.getBuildFeatures()` and `j.b.agent.AgentRunningBuild.getBuildFeaturesOfType(String)`. With help of these methods you can access build features enabled for the current build with all parameters properly resolved.
- new API calls: `j.b.serverSide.BuildTypeSettings.isEnabled(String)` and `j.b.serverSide.BuildTypeSettings.setEnabled(String, boolean)`. These calls allow to enable / disable a setting with specified id (build runner, trigger or build feature), or check if it is enabled.
- Classes from `serviceMessages.jar` no longer depend on `j.b.messages.Status` class. If you used some of the classes (for example, `j.b.messages.serviceMessages.BuildStatus` class) and want to make your code compatible with TeamCity versions 6.0 - 7.0, please use `j.b.messages.serviceMessages.ServiceMessage.asString(...)` methods.
- new API extension point to filter all build messages: `j.b.messages.BuildMessagesTranslator`
- `j.b.serverSide.BuildServerListener` - we've removed `beforeBuildFinish(SRunningBuild, boolean)` method which was deprecated since TeamCity 3.1, there is another method `beforeBuildFinish(SRunningBuild)` which can be used instead.

Changes from 6.0 to 6.5

- Classes `j.b.serverSide.TestBlockBean`, `j.b.serverSide.TestInProject`, `j.b.serverSide.FailedTestBean`, `j.b.TestNameBean` are removed from the Open API. Interfaces `j.b.serverSide.STest`, `j.b.serverSide.STestRun` should be used instead.
- `j.b.serverSide.ShortStatistics.getFailedTests()`, `j.b.serverSide.BuildStatistics.getIgnoredTests()` and `j.b.serverSide.BuildStatistics.getPassedTests()` return the list of `j.b.serverSide.STestRun` accordingly.
- Classes `j.b.tests.TestName` and `j.b.tests.SuiteTestName` are combined together into `j.b.tests.TestName`.

Changes from 5.1.2 to 6.0

- `j.b.vcs.TriggerRules` class was removed from Open API as part of API cleanup. Please let us know if your plugin is affected by the change.

New responsibility event methods added:

- `j.b.serverSide.BuildServerListener.responsibleChanged(SProject, Collection<SuiteTestName>, ResponsibilityEntry, boolean)`.
- `j.b.notification.Notificator.notifyResponsibleChanged(Collection<SuiteTestName>, ResponsibilityEntry, SProject, Set<SUser>)`, `j.b.notification.Notificator.notifyResponsibleAssigned(Collection<SuiteTestName>, ResponsibilityEntry, SProject, Set<SUser>)`.

- j.b.notification.NotificationEventListener.responsibleChanged(SProject, Collection<SuiteTestName>, ResponsibilityEntry, boolean).
- j.b.messages.ServiceMessageTranslator is reworked to allow binding to arbitrary message type by name instead of only known types

Most methods in j.b.agent.AgentLifeCycleListener interface were extended to receive j.b.agent.BuildRunnerContext.

j.b.agent.AgentLifeCycleListener#runnerFinished(...) method added. It is called after build step is finished.

j.b.agent.duplicates.DuplicatesReporter and j.b.duplicator.DuplicateInfo are added for reporting code duplicates on agent side.

Build Agent changes:

- j.b.agent.AgentRunningBuild does not extend j.b.agent.AgentBuildInfo, j.b.agent.ResolvedParameters. All methods from those interfaces were inlined into AgentRunningBuild interface.

Most methods from j.b.agent.AgentRunningBuild were splitted into j.b.agent.BuildRunnerContext and j.b.agent.BuildContext. We have added

Parameters required for build runner are represented with j.b.agent.BuildRunnerContext interface.

Every time AgentRunningBuild and BuildRunnerContext return resolved parameters back.

j.b.agent.BuildRunnerContext represents the context of current build runner. All add* methods modifies context for the runner. Those changes will be reverted when context is switched to next runner.

j.b.agent.AgentRunningBuild provides a context of a build (i.e. shared between all runners). All addShared* methods modifies the build context (and thus all build runner contexts).

j.b.agent.BuildAgentConfiguration now contains getBuildParameters() and getConfigParameters() methods to access parameters. Configuration parameters here are formed from properties from buildAgent.properties that does not start from 'system.' or 'env.' prefix. All parameters are returned with all references resolved.

j.b.agent.AgentBuildRunner#createBuildProcess method signature has been changed to receive j.b.agent.BuildRunnerContext.

j.b.agent.CommandLineBuildService#initialize(...) method signature has been changed to receive j.b.agent.BuildRunnerContext.

j.b.agent.CommandLineBuildService#getRunnerContext(...) added

j.b.agent.CommandLineBuildService#afterProcessSuccessfullyFinished() added

j.b.agent.BuildServiceAdapter is added to simplify as proposed base class for commandline base build runner service.

Changes from 5.0 to 5.1

Web extensions:

- deprecated method removed:
j.b.web.openapi.WebControllerManager.addPageExtension(final WebPlace addTo, final WebExtension extension, Anchor<WebExtension> anchor)
- deprecated class removed: j.b.serverSide.Anchor
- deprecated class removed: j.b.notification.TemplatePatternProcessor; j.b.notification.TemplateProcessor added instead, see [Extending Notification Templates Model](#)
- method removed: j.b.notification.TemplateMessageBuilder.setPatternProcessor()
- several methods in j.b.serverSide.SBuildType now return boolean instead of void. You will probably need to recompile your plugins that use the interface.

Changes from 4.5.5 to 5.0

Parameters

j.b.serverSide.parameters.AbstractBuildParameterReferencesProvider is renamed to j.b.serverSide.parameters.AbstractBuildParametersProvider
j.b.serverSide.parameters.BuildParameterReferencesProvider is renamed into j.b.serverSide.parameters.BuildParametersProvider
BuildParameterReferencesProvider.getParameters(@NotNull final SBuild build) changed signature to getParameters(@NotNull final SBuild build, final boolean emulationMode)
j.b.agent.BuildAgentConfiguration#getCacheDirectory now receives String as argument
j.b.serverSide.buildDistribution.StartBuildPrecondition#canStart second parameters
(Map<QueuedBuildInfo, BuildAgent>) may contain null values for some queued builds

Miscellaneous

Added new build server events:

```
j.b.serverSide.BuildServerListener.vcsRootRemoved(SVcsRoot),  
j.b.serverSide.BuildServerListener.responsibleChanged(SProject, TestNameResponsibilityEntry,  
TestNameResponsibilityEntry, boolean)
```

Added three notification methods:

```
j.b.notification.Notificator.notifyResponsibleAssigned(SBuildType, Set<SUser>),  
j.b.notification.Notificator.notifyResponsibleChanged(TestNameResponsibilityEntry,  
TestNameResponsibilityEntry, SProject, Set<SUser>), j.b.notification.Notificator.notifyResponsibleAssigned(T  
estNameResponsibilityEntry, TestNameResponsibilityEntry, SProject, Set<SUser>)
```

Changes prior to 4.5.5

Not documented

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](#).

Plugins Packaging

This page is intended for plugin developers and explains how to package TeamCity plugins and agent tools. See [Installing Additional Plugins](#) and [Installing Agent Tools](#) for installation instructions.

On this page:

- [Introduction](#)
- [Plugins Location](#)
- [Plugins Loading](#)
- [Server-Side Plugins](#)
 - [Plugin Structure](#)
 - [Web resources packaging](#)
- [Plugin Descriptor](#)
- [Agent-Side Plugins](#)
 - [Plugin Structure](#)
 - [Deprecated Plugin Structure](#)
 - [New Plugins](#)
 - [Plugin Descriptor](#)
 - [Plugins](#)
 - [Tools](#)
- [Plugin Dependencies](#)
- [Agent Upgrade on Updating Plugins](#)

Introduction

To write a TeamCity plugin, the knowledge of [Spring Framework](#) is beneficial.

There are [server-side](#) and [agent-side](#) plugins in TeamCity. Server-side and agent-side plugins are initialized in their own Spring containers; this means that every plugin needs a Spring bean definition file describing the main services of the plugin. Bean definition files are to be placed into the `META-INF` folder of the JAR archive containing the plugin classes.

There is a convention for naming the definition file:

- **build-server-plugin-<plugin name>*.xml** — for server-side plugins
- **build-agent-plugin-<plugin name>*.xml** — for agent-side plugins

where the asterisk can be replaced with any text, for example: **build-server-plugin-cvs.xml**.

 If you want to get started with an empty plugin quickly, try the template plugin in the JetBrains Subversion repository <http://svn.jetbrains.org/teamcity/plugins/template-plugin/templateProject>. Refer to `readme.txt` for instructions.

Plugins Location

TeamCity is able to load plugin from the following directories:

- `<TeamCity data directory>/plugins` — user-installed plugins
- `<TeamCity web application>/WEB-INF/plugins` — default directory for bundled TeamCity plugins

Plugins with the same name (for example, a newer version) located in `<TeamCity data directory>/plugins` will override the plugins in the `<TeamCity web application>/WEB-INF/plugins` directory.

Plugins Loading

TeamCity creates a child Spring Framework context per plugin. There are two options to load plugins classes: **standalone** and **shared**:

- Standalone classloading (**recommended**) allows loading every plugin to a separate classloader. This approach allows a plugin to have additional libraries without the risk of affecting the server or other plugins.
- Shared classloading allows loading all plugins into same classloader. It is not allowed to override any libraries here.

You may specify desired the classloading mode in the `teamcity-plugin.xml` file, see the [section below](#).

 Since TeamCity 9.0, the TeamCity plugin loader supports plugin dependencies, described [below](#).

 To load your plugin, the server must be restarted.

Server-Side Plugins

A server-side plugin may affect the server only, or may include a number of agent-side plugins that will be automatically distributed to all build agents.

Plugin Structure

A plugin can be a zip archive (**recommended**) or a separate folder.

If you use a *zip file*:

- TeamCity will use the name of the zip file as the plugin name
- The plugin zip file will be automatically unpacked to a temporary directory on the server start-up

If you use a *separate folder*:

- TeamCity will use the folder name as the plugin name

The plugin zip archive/directory includes:

- `teamcity-plugin.xml` containing meta information about the plugin, like its name and version, see the [section below](#).
- the `server` directory containing the server-side part of the plugin, i.e, a number of jar files.
- the `agent` directory containing `<agent plugin zip>` if your plugin affects agents too, see the [section below](#).

The plugin directory should have the following structure:

The server-only plugin:

```
server
  |
  --> <server plugin jar files>
teamcity-plugin.xml
```

The plugin affecting the server and agents:

```
agent
|
--> <agent plugin zip files> (see [below|#agentDirectory])
server
|
--> <server plugin jar files>
teamcity-plugin.xml
```

Web resources packaging

In most cases a plugin is just a number of classes packed into a JAR file.

If you wish to write a custom page for TeamCity, most likely you'll need to place images, CSS, JavaScript, JSP files or binaries somewhere. The files that you want to access via hyperlinks and JSP pages are to be placed into the `buildServerResources` subfolder of the plugin's .jar file. Upon the server startup, these files will be extracted from the archive. You may use `jetbrains.buildServer.web.openapi.PluginDescriptor` spring bean to get the paths to the extracted resources ([read more](#) on how to construct paths to your JSP files).

It is a good practice to put all resources into a separate.jar file.

Plugin Descriptor

The `teamcity-plugin.xml` file must be located in the root of the plugin directory or .zip file. You can refer to the XSD schema for this file which is unpacked to `<TeamCity data directory>/config/teamcity-plugin-descriptor.xsd`

An example of `teamcity-plugin.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<teamcity-plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:shemas-jetbrains-com:teamcity-plugin-v1-xml">
    <info>
        <name>PluginName</name> <!-- the name of plugin used in teamcity -->
        <display-name>This name may be used for UI</display-name>
        <description>Some description goes here</description>
        <version>0.239.42</version>
    </info>
    <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:shemas-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>
```

There is experimental ability (can be removed in the future versions!) to set executable bit to some files after unpacking on the agent. Watch [TW-21673](#) for proper solution.

To make some files of a tool executable, use the following `teamcity-plugin.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
    <teamcity-agent-plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="urn:schemas-jetbrains-com:teamcity-agent-plugin-v1-xml">
            <tool-deployment>
                <layout>
                    <executable-files>
                        <include name='path_to_a_file' />
                    </executable-files>
                </layout>
            </tool-deployment>
        </teamcity-agent-plugin>
```

where `<include name='path_to_a_file' />` specifies the list of files (relative to agent plugin directory) to be made executable on Linux/Unix/Mac. See [Installing Agent Tools](#) for installation instructions.

Plugin Dependencies

Since TeamCity 9.0, plugin dependencies are introduced on both the server and agent side.

Some components are separated from the core into separate bundled plugins: Ant runner, IDEA runner, .NET runners, JUnit and TestNG support.

If you need some functionality of one of these plugins, use the plugin dependencies feature.

To use plugin dependencies, add the `dependencies` tag into the plugin xml descriptor:

Example of the server-side plugin descriptor using plugin dependencies:

```
<?xml version="1.0" encoding="UTF-8"?>
<teamcity-plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:schemas-jetbrains-com:teamcity-plugin-v1-xml">
    <info>
        <name>Plugin Name</name>
        <!-- Some tags skipped -->
    </info>
    <deployment use-separate-classloader="true" />
    <dependencies>
        <plugin name="dotNetRunners" />
    </dependencies>
</teamcity-plugin>
```

Example of agent-side plugin descriptor:

```
<?xml version="1.0" encoding="UTF-8"?>
<teamcity-agent-plugin xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:schemas-jetbrains-com:teamcity-agent-plugin-v1-xml"
>
<plugin-deployment use-separate-classloader="true" />
<dependencies>
<tool name="ant" />
<plugin name="ant-runner" />
</dependencies>
</teamcity-agent-plugin>
```



Using separate classloader is required (and will be enforced) to use dependencies.
Transitive dependencies are not supported, you should specify all dependencies.

The names of the bundled tools and plugins are just the names of the corresponding folders in **TeamCity Home/webapps/ROOT/WEB-INF/plugins** for the server-side plugins and **<Agent home>/plugins/** or **<Agent home>/tools/** for the agent-side plugins and tools.



Some bundled plugin names may change in future releases.

Agent Upgrade on Updating Plugins

TeamCity server monitors all agent plugins .zip files for a change (plugin files changed, added or removed). Once a change is detected, agents receive the upgrade command from the server and download the updated files automatically. It means that if you want to deploy an updated agent part of your plugin without the server restart, you can put your agent plugin into this folder.

After a successful upgrade, your plugin will be unpacked into the **<Agent home>/plugins/** or **<Agent home>/tools/** folders. Note that if an agent is busy running a build, it will upgrade only after the build finishes. No new builds will start on the agent if it is to be upgraded.

Server-side Object Model

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.
jetbrains.buildServer.serverSide.GeneralDataCleaner	2.0	This extension is called when cleanup process is going to finish, plugins can clean their data with help of this extension.
jetbrains.buildServer.serverSide.DataCleaner	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.
jetbrains.buildServer.serverSide.ParametersPreprocessor	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 ParametersPreprocessor in a way that parameters added by BuildParametersProvider 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 BuildParametersProvider .
<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

<code>jetbrains.buildServer.web.openapi.WebControllerManager</code>	Maintains a collection of custom controllers, allows to register custom controllers
<code>jetbrains.buildServer.controllers.BaseController</code>	Base class for controllers

Plugin Settings

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 Debugging

You can debug your plugin in a running TeamCity just like a regular Java application debug: start TeamCity server with debug-enabling JVM options and then connect to a remote debug port from the IDE. If you start TeamCity server from outside of your IDE, in IntelliJ IDE you can use "Remote" run configuration, check related [external blog post](#). The JVM options for the server can be set via `TEAMCITY_SERVER_OPTS` environment variable.

Plugin Reloading

If you make changes to a plugin, you will generally need to shut down the server, update the plugin, and start the server again.

To enable TeamCity development mode, pass the `"teamcity.development.mode=true"` internal property. Using the option you will:

- Enforce application server to quicker recompile changed `.jsp` classes
- Disable JS and CSS resources merging/caching

The following hints can help you eliminate the restart in the certain cases:

- if you do not change code affecting plugin initialization and change only body of the methods, you can attach to the server process with a debugger and use Java hotswap to reload the changed classes from your IDE without web server restart. Note that the standard hotswap does not allow you to change method signatures.
- if you make a change in some resource (`.jsp`, `.js`, `images`) you can copy the resources to `webapps/ROOT/plugins/<plugin-name>` directory to allow Tomcat to reload them.
- change in build agent part of plugin will initiate build agents upgrade.

If you replace a deployed plugin `.zip` file with changed class files while TeamCity server is running, this can lead to `NoClassDefFound` errors. To avoid this, set `"teamcity.development.shadowCopyClasses=true"` internal property. This will result in:

- creating `".teamcity_shadow"` directory for each plugin `.jar` file;
- avoid `.jar` files update on plugin archive change.

See also:

[Extending TeamCity: Developing TeamCity Plugins | Plugins Packaging](#)

Developing Plugins Using Maven

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>9.0</version>
  <scope>provided</scope>
</dependency>
```

The agent-side API:

```
<dependency>
  <groupId>org.jetbrains.teamcity</groupId>
  <artifactId>agent-api</artifactId>
  <version>9.0</version>
  <scope>provided</scope>
</dependency>
```

i Note that API dependencies are used with the provided scope. This way you will avoid adding the API and its transitive dependencies to the target distribution.

There is also an artifact to support plugin tests:

```
<dependency>
  <groupId>org.jetbrains.teamcity</groupId>
  <artifactId>tests-support</artifactId>
  <version>9.0</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 9.0 TeamCity version:

```
mvn archetype:generate -DarchetypeRepository=http://repository.jetbrains.com/all  
-DarchetypeArtifactId=teamcity-server-plugin  
-DarchetypeGroupId=org.jetbrains.teamcity.archetypes -DarchetypeVersion=LATEST
```

Here is the Maven command that will generate a project that contains both, the server and agent parts of a plugin and depends on 9.0 TeamCity version:

```
mvn archetype:generate -DarchetypeRepository=http://repository.jetbrains.com/all  
-DarchetypeArtifactId=teamcity-plugin  
-DarchetypeGroupId=org.jetbrains.teamcity.archetypes -DarchetypeVersion=LATEST
```

Here is the Maven command that will generate a sample project on 9.0 TeamCity version:

```
mvn archetype:generate -DarchetypeRepository=http://repository.jetbrains.com/all  
-DarchetypeArtifactId=teamcity-sample-plugin  
-DarchetypeGroupId=org.jetbrains.teamcity.archetypes -DarchetypeVersion=LATEST
```

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.

Plugin Development FAQ

- [How to Use Logging](#)

How to Use Logging

The TeamCity code uses the Log4j logging library with a centralized configuration on the [server](#) and [agent](#).

Logging is usually done via a utility wrapper `com.intellij.openapi.diagnostic.Logger` rather than the default Log4j classes.

You can use the [jetbrains.buildServer.log.Loggers](#) class to get instances of the Loggers, e.g. use `jetbrains.buildServer.log.Loggers.SERVER` to add a message to the `teamcity-server.log` file.

For plugin-specific logging it is recommended to log into a log category matching the full name of your class. This is usually achieved by defining the logger field in a class as `private static Logger LOG = Logger.getInstance(YourClass.class.getName());`

If your plugin source code is located under the `jetbrains.buildServer` package, the logging will automatically go into `teamcity-server.l`

og.

If you use another package, you might need to add a corresponding category handling into the `conf/teamcity-server-log4j.xml` file (mentioned at [TeamCity Server Logs](#)) or the corresponding agent file.

For debugging you might consider creating a customized Log4j configuration file and put it as a logging preset into `<TeamCity Data Directory>\config_logging` directory. This way one will be able to activate the preset via the [Administration | Diagnostics](#) page, **Trouble shooting** tab.

Getting Started with Plugin Development

The use of plugins allows you to extend the TeamCity functionality. See the [list of existing TeamCity plugins](#) created by JetBrains developers and community.

This document provides information on how to develop and publish a server-side plugin for TeamCity [using Maven](#). The plugin will return the "Hello World" jsp page when using a specific URL to the TeamCity Web UI.

On this page:

- [Introduction](#)
- [Step 1. Set up the environment](#)
- [Step 2. Generate a Maven project](#)
 - [View the project structure](#)
- [Step 3. Edit the plugin descriptor](#)
- [Step 4. Create the plugin sources](#)
 - [A. Create the plugin web-resources](#)
 - [B. Create the controller and obtain the path to the JSP](#)
 - [C. Update the Spring bean definition](#)
- [Step 4. Build your project with Maven](#)
- [Step 5. Install the plugin to TeamCity](#)
- [Next Steps](#)

Introduction

A *plugin* in TeamCity is a `zip` archive containing a number of classes packed into a JAR file and [plugin descriptor](#) file.

The TeamCity Open API can be found in the JetBrains [Maven repository](#). The Javadoc reference for the API is available [online](#) and locally in `<TeamCity Home Directory>/devPackage/javadoc/openApi-help.jar`, after you install TeamCity.

Step 1. Set up the environment

To get started writing a plugin for TeamCity, set up the plugin development environment.

1. Download and install [Oracle Java](#). Set the `Java_Home` environment variable on your system. The 32-bit Java 1.7 and [since Teamcity 9.1](#) Java 1.8 is recommended, the 64-bit version [can be used](#).
2. Download and install [TeamCity](#) on your development machine. Since you are going to use this machine to test your plugin, it is recommended that this TeamCity server is of the same version as your production server. We are using TeamCity 9.0.2 installed on Windows in our setup.
3. Download and install a Java IDE; we are using [IntelliJ IDEA 14.0.3 Community Edition](#), which has a built-in Maven integration.
4. Download and install [Apache Maven](#). Maven 3.2.x is recommended. Set the `M2_HOME` environment variable. Run `mvn -version` to verify your setup. We are using Maven 3.2.5. in our setup.

Step 2. Generate a Maven project

We'll generate a Maven project [from an archetype](#) residing in JetBrains Maven repository. Executing the following command will produce a project for a server-side-only plugin depending on the latest TeamCity version:

```
mvn archetype:generate -DarchetypeRepository=http://repository.jetbrains.com/all  
-DarchetypeArtifactId=teamcity-server-plugin  
-DarchetypeGroupId=org.jetbrains.teamcity.archetypes -DarchetypeVersion=LATEST
```

You will be asked to enter the Maven `groupId`, `artifactId`, `version` and `package` name for your plugin.

We used the following values:

<code>groupId</code>	<code>com.demoDomain.teamcity.demoPlugin</code>
<code>artifactId</code>	<code>demoPlugin</code>
<code>version</code>	leave the default <code>1.0-SNAPSHOT</code>

packaging

leave the default package name

demoPlugin will be used as the internal name of our plugin.

When the build finishes, you'll see that the demoPlugin directory was created in the directory where Maven was called.

View the project structure

The root of the demoPlugin directory contains the following:

- the readme.txt file with minimal instructions to develop a server-side plugin
- the pom.xml file which is your Project Object Model
- the teamcity-plugin.xml file which is your **plugin descriptor** containing meta information about the plugin.
- the demoPlugin-server directory contains the plugin sources:
 - \src\main\java\zip contains the AppServer.java file
 - src\main\resources includes resources controlling the plugin look and feel.
 - src\main\resources\META-INF folder contains build-server-plugin-demo-plugin.xml, the bean definition file for our plugin. TeamCity plugins are initialized in their own Spring containers and every plugin needs a Spring bean definition file describing the main services of the plugin.
- the build directory contains the xml files which define how the project output is aggregated into a single distributable archive.

Step 3. Edit the plugin descriptor

Open the teamcity-plugin.xml file in the project root folder and add details, such as the plugin display name, description, vendor, and etc. by modifying the corresponding attributes in the file.

Step 4. Create the plugin sources

Open the pom.xml from the project root folder with IntelliJ IDEA.

We are going to make a controller class which will return Hello.jsp via a specific TeamCity URL.

A. Create the plugin web-resources

The plugin web resources (files that are accessed via hyperlinks and JSP pages) are to be placed into the buildServerResources subfolder of the plugin's resources.

1. First we'll create the directory for our jsp: go to the demoPlugin-server\src\main\resources directory in IDEA and create the buildServerResources directory.
2. In the newly created demoPlugin-server\src\main\resources\buildServerResources directory, create the Hello.jsp file, e.g.

```
<html>
<body>
Hello world
</body>
</html>
```

B. Create the controller and obtain the path to the JSP

Go to \demoPlugin\demoPlugin-server\src\main\java\com\demoDomain\teamcity\demoPlugin and open the AppServer.java file to create a custom controller:

1. We'll create a simple controller which extends the TeamCity **jetbrains.buildServer.controllers.BaseController** class and implements the BaseController.doHandle(HttpServletRequest, HttpServletResponse) method.
2. The TeamCity open API provides the **jetbrains.buildServer.web.openapi.WebControllerManager** which allows registering custom controllers using the path to them: the path is a part of URL starting with a slash / appended to the URL of the server root.
3. Next we need to construct the path to our JSP file. When a plugin is unpacked on the TeamCity server, the paths to its resources change. To obtain valid paths to the files after the plugin is installed, use the **jetbrains.buildServer.web.openapi.PluginDescriptor** class which implements the `getPluginResourcesPath` method; otherwise TeamCity might have difficulties finding the plugin resources.

```

package com.demoDomain.teamcity.demoPlugin;

import jetbrains.buildServer.controllers.BaseController;
import jetbrains.buildServer.web.openapi.PluginDescriptor;
import jetbrains.buildServer.web.openapi.WebControllerManager;
import org.jetbrains.annotations.Nullable;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class AppServer extends BaseController {
    private PluginDescriptor myDescriptor;

    public AppServer (WebControllerManager manager, PluginDescriptor descriptor)
    {
        manager.registerController("/demoPlugin.html",this);
        myDescriptor=descriptor;
    }

    @Nullable
    @Override
    protected ModelAndView doHandle(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse) throws Exception {
        return new
    ModelAndView(myDescriptor.getPluginResourcesPath("Hello.jsp"));
    }
}

```

C. Update the Spring bean definition

Go to the demoPlugin-server\src\main\resources\META-INF directory and update build-server-plugin-demo-plugin.xml to include our AppServer class.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans default-autowire="constructor">
    <bean class="com.demoDomain.teamcity.demoPlugin.AppServer"></bean>
</beans>

```

Step 4. Build your project with Maven

Go to the root directory of your project and run

```
mvn package
```

The target directory of the project root will contain the <demoPlugin>.zip file. It is our plugin package, ready to be installed.

Step 5. Install the plugin to TeamCity

1. Copy the plugin zip to <TeamCity Data Directory>/plugins directory.
2. Restart the server and locate the TeamCity Demo Plugin in the **Administration|Plugins List** to verify the plugin was installed correctly.

The screenshot shows the 'External plugins' section of the TeamCity Administration interface. It lists one plugin: 'Hello World' by 'TeamCity'. The table includes columns for Name, Version, Vendor, and Home Path. The Home Path is listed as 'TeamCity Data Directory/.../helloWorldPlugin'.

Name	Version	Vendor	Home Path
Hello World	1.0	TeamCity	TeamCity Data Directory/.../helloWorldPlugin

The Hello World page is available via <TeamCity server URL>/demoPlugin.html.

Next Steps

[Read more](#) if you want to extend the TeamCity pages with custom elements.

The detailed information on TeamCity plugin development is available [here](#).

You may also use the [plugin](#) allowing you to control a TeamCity instance from the command line and to install a new/updated plugin created from a Maven archetype.

How To...

In this section:

- Choose OS/Platform for TeamCity Server
- Estimate Hardware Requirements for TeamCity
- Retrieve Administrator Password
- Estimate External Database Capacity
- Estimate the Number of Required Build Agents
- Setup TeamCity in Replication/Clustering Environment
- TeamCity Security Notes
- Configure Newly Installed MySQL Server
 - InnoDB database engine
 - max_connections
 - innodb_buffer_pool_size and innodb_log_file_size
 - innodb_file_per_table
 - innodb_flush_log_at_trx_commit
 - log files on different disk
 - Setting The Binary Log Format
 - Enable additional diagnostics
- Configure Newly Installed PostgreSQL Server
 - shared_buffers
 - checkpoint_segments
 - synchronous_commit
- Set Up TeamCity behind a Proxy Server
 - Apache
 - Nginx
 - Other servers
 - Tomcat settings
- Configure TeamCity to Use Proxy Server for Outgoing Connections
- Install Multiple Agents on the Same Machine
- Change Server Port
- Test-drive Newer TeamCity Version before Upgrade
- Create a Copy of TeamCity Server with All Data
- Move TeamCity Projects from One Server to Another
- Move TeamCity Installation to a New Machine
- Move TeamCity Agent
- Share the Build number for Builds in a Chain Build
- Make Temporary Build Files Erased between the Builds
- Clear Build Queue if It Has Too Many Builds due to a Configuration Error
- Automatically create or change TeamCity build configuration settings
- Attach Cucumber Reporter to Ant Build
- Get Last Successful Build Number
- Set up Deployment for My Application in TeamCity
- Use an External Tool that My Build Relies on
- Integrate with Build and Reporting Tools
- Restore Just Deleted Project
- Transfer 3 Default Agents to Another Server
- Import coverage results in TeamCity
- Recover from "Data format of the data directory (NNN) and the database (MMM) do not match" error
- Debug a Build on a Specific Agent
- Debug a Part of the Build (a build step)
- Vulnerabilities
 - Heartbleed, ShellShock
 - POODLE

- GHOST
- FREAK
- Watch Several TeamCity Servers with Windows Tray Notifier
- TeamCity Release Cycle

Choose OS/Platform for TeamCity Server

Once the server/OS fulfills the [requirements](#), TeamCity can run on any system.500 agents

Please also review the [requirements](#) for the integrations you plan to use, for example the following functionality requires or works better when TeamCity server is installed under Windows:

- VCS integration with TFS
- VCS integration with VSS
- Windows domain logins (can also work under Linux, but may be less stable), especially NTLM HTTP authentication
- NuGet feed on the server (can also work under Linux, but may be less stable)
- Agent push to Windows machines

If you have no preference, Linux platforms may be more preferable due to more effective file system operations and the level of required general OS maintenance.

Final Operating System choice should probably depend more on the available resources and established practices in your organization.

If you choose to install 64 bit OS, TeamCity can run under 64 bit JDK (both server and agent).

However, unless you need to provide more than 1Gb memory for TeamCity, the recommended approach is to use 32 bit JVM even under 64 bit OS. Our experience suggests that using 64 bit JVM does not increase performance a great deal. At the same time it does increase memory requirements to almost the scale of 2. See a [note](#) on memory configuration.

Estimate Hardware Requirements for TeamCity

The hardware requirements differ for the server and the agents.

The **agent** hardware requirements are basically determined by the builds that are run. Running TeamCity agent software introduces 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 (number of builds, number of tests, etc.) An active server database usage can be estimated at several gigabytes of data per year.

Overview on the TeamCity hardware resources usage:

- CPU: TeamCity utilizes multiple cores of the CPU, so increasing number of cores makes sense. For non-trivial TeamCity usage at least 4 CPU cores are recommended.
- Memory: See a [note](#) on memory usage. Consider also that required memory may depend on the JVM used (32 bit or 64 bit). Generally, you will probably not need to dedicate more than 4G of memory to TeamCity server if you do not plan to run more then 100 concurrent builds (agents) and more then 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.

The latest TeamCity version is known to work well with up to 300 build agents (300 concurrently running builds actively logging build run-time data). In synthetic tests the server was functioning OK with as many as 500 concurrent builds (the server with 8 cores, 32Gb of total memory running under Linux, and MySQL server running on a separate comparable machine). The load on the server produced by each build depends on the amount of data the build logs (build log, tests number and failure details, inspections/duplicates issues number, etc.). Keeping the amount of data reasonably constrained (publishing large outputs as build artifacts, not printing those into standard output; tweaking inspection profiles to report limited set of the most important inspection hits, etc.) will help scale the server to handle more concurrent builds. If you need much more agents/parallel builds, it is recommended to 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 post on the [maximum number of agents which TeamCity can handle](#)

See also a related post: [description of a substantial TeamCity setup](#).

Retrieve Administrator Password

On the first start with empty database TeamCity displays the Administrator Setup page. A TeamCity installation should always have a user with the System Administrator role.

If there is no user account with the System Administrator role in the current authentication scheme, you can use the `http://<your_TeamCity_server>/setupAdmin.html` URL to setup an administrator account.

If there is an administrator account already, the page is not available. To regain the access to the system you need to either log in with existing administrator account credentials or log in as a [super user](#) and change the existing administrator account password or create a new account with System Administrator role.

Other options (less recommended):

If you forgot the Administrator password and your TeamCity uses an internal database, you can reset the password using the [instructions](#). Otherwise you can use [REST API](#) to add the System Administrator role to any existing user.

There are also [instructions](#) to patch roles directly in the database provided by a user.

Estimate External Database Capacity

It is quite hard to provide the exact numbers when setting up or migrating to an external database, as the required capacity varies greatly depending on how TeamCity is used.

The database size and database performance are crucial aspects to consider.

Database Size

The size of the database will depend on:

- how many builds are started every day
- how many test are reported from builds
- [clean-up](#) rules (retention policy)
- cleanup schedule

We recommend the initial size of data spaces to be 4 GB. When migrating from the internal database, we suggest at least doubling the size of the current internal database. For example, the size of the external database (without the Redo Log files) of the internal TeamCity server in JetBrains is about 50 GB. Setting your database to grow automatically helps to increase file sizes to a pre-determined limit when necessary, which minimizes the effort to monitor disk space.

Allocating 1 GB for the redo log (see the table [below](#)) and undo files is sufficient in most cases.

Database Performance

The following factors are to be taken into account:

- type of database (RDBMS)
- number of agents (which actually means the number of builds running in parallel)
- number of web pages opened by all users
- [clean-up](#) rules (retention policy)

It is advised to place the [TeamCity Data directory](#) and database data files on physically different hard disks (even when both the TeamCity server and RDBMS share the same host).

Placing redo logs on a separate physical disk is also recommended especially in case of the high number of agents (50 and more).

Database-specific considerations

 The redo log (or a similar entity) naming for different RDBMS:

RDBMS	Log name
Oracle	Redo Log
MS SQL Server	Transaction Log
PostgreSQL	WAL (write ahead log)
MySQL + InnoDB and Percona	Redo Log

PostgreSQL: We recommend using version 9.2+, which has a lot of query optimization features. Also see the information on the write-ahead-log (WAL) in the [PostgreSQL documentation](#)

Oracle: it is recommended to keep statistics on: all automatically gathered statistics should be enabled (since Oracle 10.0, this is the default set-up). Also see the information on redo log files in the [Oracle documentation](#).

MS SQL Server: it is NOT recommended to use the JTDS driver: it does not work with `nchar/nvarchar`, and to preserve unicode streams it may cause queries to take a long time and consume a lot of IO. Also see the information on redo log in the [Microsoft Knowledge base](#)

MySQL: the query optimizer might be inefficient: some queries may get a wrong execution plan causing them to take a long time and consume huge IO.

Estimate the Number of Required Build Agents

There are no precise data and the number of required build agents depends a lot on the server usage pattern, type of builds, team size, commitment of the team to CI process, etc.

The best way is to start with the default 3 agents and see how that plays with the projects configured, then estimate further based on that.

You might want to increase the number of agents when you see:

- builds waiting for an idle agent in the build queue;
- more changes included into each build than you find comfortable (e.g. for build failures analysis);
- necessity for different environments.

We've seen patterns of having an agent per each 20 build configurations (types of builds). Or a build agent per 1-2 developers.

See also [notes](#) on maximum supported number of agents.

Setup TeamCity in Replication/Clustering Environment

TeamCity does not provide specific support for replication/redundancy/high availability or clustering solutions.

However to address fast disaster recovery scenarios 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.

As to data, TeamCity server uses both database and file storage (Data Directory). You can browse through [TeamCity Data Backup](#) and [TeamCity Data Directory](#) pages in to get more information on TeamCity data storing.

Basically, both TeamCity data directory on 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.

Ensure that the distribution of the failover/backup server is of exactly the same version as the main server. It is also important to ensure the same server environment/startup options like memory settings, etc.

TeamCity agents farm can be reused between the main and the failover servers. Agents will automatically connect to the new server if you make the failover server to be resolved via the old server DNS name and agents connect to the server using the DNS name. See also information on [switching](#) from one server to another.

If appropriate, the agents can be replicated just as the server. However, there is no need to replicate any TeamCity-specific data on the agents except for the `conf/buildAgent.properties` file as all the rest of the data can typically be renewed from the server. In case of replicated agents farm, the replica agents just need to be connected to the failover server.

In case of two servers installations for redundancy purposes, they can use the same set of licenses as only one of them is running at any given moment.

TeamCity Security Notes

For a list of disclosed security-related issues see our [public issue tracker](#) and "Security" section in the release notes.

It is recommended to upgrade to the newly released TeamCity versions as soon as they become available as they can contain security-related fixes.

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, publish any files as artifacts for the builds built on the agent (which means the files can be then displayed in TeamCity web UI and expose web vulnerabilities or can be used in other builds), 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. Also, the users can do everything which user with "View build configuration settings" permission can do.
- users with "View build configuration settings" permission ("Project developer" TeamCity role by default) can view all the projects on the server, but since TeamCity 9.0 there is a way to restrict this, see details in the corresponding issue [TW-24904](#).
- users with "Edit project" permission ("Project Administrator" TeamCity role by default) in one project can retrieve artifacts and trigger builds from any build configuration they have only view permission for.
- users with "Change server settings" permission ("System Administrator" TeamCity role by default): 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.
 - When [storing settings in VCS](#) is enabled, any user who can access the settings repository (including users with "View file content" permission for the build configurations using the same VCS root) can see the settings and retrieve the actual passwords based on their stored scrambled form. **Since TeamCity 9.1**, when option is enabled to allow per-build settings from personal builds, any user who can run a personal build, in fact gets permissions like a project administrator of a side project on the server.

Configure Newly Installed MySQL Server

If MySQL server is going to be used with TeamCity in addition to the [basic setup](#), you should review and probably change some of the MySQL server settings.

If MySQL is installed on Windows, the settings are located in `my.ini` file which usually can be found under MySQL installation directory. For Unix-like systems the file is called `my.cnf` and can be placed somewhere under `/etc` directory. Read more about configuration file location in [MySQL documentation](#). Note: you'll need to restart MySQL server after changing settings in `my.ini` | `my.cnf`.

The following settings should be reviewed and/or changed:

InnoDB database engine

Make sure you're using InnoDB database engine for tables in TeamCity database. You can check what engine is used with help of this command:

```
show table status like '<table name>';
```

or for all tables at once:

```
show table status like '%';
```

max_connections

You should ensure `max_connections` parameter has bigger value than the one specified in TeamCity `<TeamCity data directory>/config/database.properties` file.

innodb_buffer_pool_size* and *innodb_log_file_size

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. After increasing buffer pool size you should also change size of the *innodb_log_file_size* setting (it's value can be calculated as *innodb_buffer_pool_size*/N, where N is the number of log files in group (2 by default)):

```
innodb_log_file_size=1024M
```

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.

innodb_flush_log_at_trx_commit

If TeamCity is the only application using MySQL database then you can improve performance by setting *innodb_flush_log_at_trx_commit* variable to 2 or 0:

```
# If set to 1, InnoDB will flush (fsync) the transaction logs to the
# disk at each commit, which offers full ACID behavior. If you are
# willing to compromise this safety, and you are running small
# transactions, you may set this to 0 or 2 to reduce disk I/O to the
# logs. Value 0 means that the log is only written to the log file and
# the log file flushed to disk approximately once per second. Value 2
# means the log is written to the log file at each commit, but the log
# file is only flushed to disk approximately once per second.
innodb_flush_log_at_trx_commit=2
```

Note: it is not important for TeamCity that database offers full ACID behavior, so you can safely change this variable.

log files on different disk

Placing the MySQL log files on different disk sometimes helps improving performance. You can read about it in [MySQL documentation](#).

Setting The Binary Log Format

If the default MySQL binary logging format is not MIXED (it depends on the [version of MySQL](#) you are using), then it should be explicitly set to **MIXED**:

```
binlog-format=mixed
```

Enable additional diagnostics

To get additional diagnostics data in case of some database-specific errors, grant more permissions for a TeamCity database user via SQL command:

```
GRANT PROCESS ON *.* TO <teamcity-user-name>;
```

Configure Newly Installed PostgreSQL Server

For better TeamCity server performance it is recommended to change some of the parameters of the newly installed PostgreSQL server. You can read more about PostgreSQL performance optimizations in [PostgreSQL Wiki](#).

Parameters below can be changed in `postgresql.conf` file which can be found in PostgreSQL's data directory.

shared_buffers

Default value of `shared_buffers` parameter is too small and should be increased.

```
shared_buffers=512MB
```

checkpoint_segments

For write intensive applications such as TeamCity it makes sense to change some of the `checkpoint` related parameters.

```
checkpoint_segments=32  
checkpoint_completion_target=0.9
```

synchronous_commit

If TeamCity is the only application using PostgreSQL database, we recommend disabling `synchronous_commit` parameter:

```
synchronous_commit=off
```

Set Up TeamCity behind a Proxy Server

This section covers the recommended setup of proxy servers when installed before TeamCity server web UI

Consider the example:

TeamCity server is installed at URL: <http://teamcity.local:8111/tc>

It is visible to the outside world as URL: <http://teamcity.public:400/tc>

You need to set up a reverse proxy. The settings below ensure requests to <http://teamcity.public:400/tc> are redirected to <http://teamcity.local:8111/tc> and the redirect URLs sent back to the clients are correctly mapped by the proxy server. If you need to use different protocols (e.g. enable https on the proxy server but not in TeamCity), you should also follow instructions in the [#Other servers](#) section.

Note: An internal TeamCity server should work under the **same context** as it is visible from outside by an external address. See also [context changing instructions](#).

Apache

Versions 2.4.5+ are recommended. Earlier versions do not support the WebSocket protocol, so use the settings noted in [the previous documentation version](#).

```

LoadModule proxy_module
/usr/lib/apache2/modules/mod_proxy.so
LoadModule proxy_http_module
/usr/lib/apache2/modules/mod_proxy_http.so
LoadModule headers_module
/usr/lib/apache2/modules/mod_headers.so
LoadModule proxy_wstunnel_module
/usr/lib/apache2/modules/mod_proxy_wstunnel.so

ProxyRequests Off

ProxyPass /tc/app/subscriptions
ws://teamcity.local:8111/tc/app/subscriptions
connectiontimeout=240 timeout=1200
ProxyPassReverse /tc/app/subscriptions
ws://teamcity.local:8111/tc/app/subscriptions

ProxyPass /tc http://teamcity.local:8111/tc
connectiontimeout=240 timeout=1200
ProxyPassReverse /tc http://teamcity.local:8111/tc

```

Please note the order of [ProxyPass rules](#), you should sort conflicting ProxyPass rules starting with the longest URLs first.

i Note that by default Apache allows only a limited number of parallel connections that may be insufficient when using the WebSocket protocol. For instance, it can result in the TeamCity server not responding when a lot of clients open the Web UI. To fix it, you may need to fine-tune the Apache configuration.

For example, on Unix you should switch to [mpm_worker](#) and configure the maximum number of simultaneous connections:

```

<IfModule mpm_worker_module>
    ServerLimit 100
    StartServers 3
    MinSpareThreads 25
    MaxSpareThreads 75
    ThreadLimit 64
    ThreadsPerChild 25
    MaxClients 2500
    MaxRequestsPerChild 0
</IfModule>

```

On Windows you may need to increase the [ThreadsPerChild](#) value as described [in the Apache documentation](#).

Nginx

Versions 1.3+ are recommended. Earlier versions do not support the WebSocket protocol, so use the settings noted in the previous documentation version .

```

map $http_upgrade $connection_upgrade {
    default upgrade;
    '' '';
}

server {
    listen      400;
    server_name teamcity.public;

    location /tc {
        proxy_pass          http://teamcity.local:8111/tc;
        proxy_http_version 1.1;
        proxy_set_header   X-Forwarded-For $remote_addr;
        proxy_set_header   Host $server_name:$server_port;
        proxy_set_header   Upgrade $http_upgrade;
        proxy_set_header   Connection $connection_upgrade;
    }
}

```

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.

Other servers

Generic notes: Make sure to use a productive proxy with due (high) limits on request (upload) and response (download) size and timeouts. It is recommended to use proxy capable to work with WebSocket protocol.

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, in addition to proxy configuration use the following approach:

Set up a proxying server to redirect all requests to `teamcity.public:400` to a dedicated port on the 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, use the `secure="true"` and `scheme="https"` attributes.
This is also described in the [comment](#).

Tomcat settings

The TeamCity server must know the original remote address of the client. This is especially important for agents, because the server tries to establish a 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 the Nginx or Apache proxy server.

Configure TeamCity to Use Proxy Server for Outgoing Connections

TeamCity can use proxy server for certain outgoing HTTP connections made by the TeamCity server to other services like issues trackers, etc. To point TeamCity at your proxy server, you need to pass additional JVM options to the TeamCity server on the start up:

JVM arguments:

```
-Dproxyset=true  
-Dhttp.proxyHost=proxy.domain.com  
-Dhttp.proxyPort=8080  
-Dhttp.nonProxyHosts=domain.com  
-Dhttps.proxyHost=proxy.domain.com  
-Dhttps.proxyPort=8080  
-Dhttps.nonProxyHosts=domain.com
```

Install Multiple Agents on the Same Machine

See the [corresponding section](#) under agent installation documentation.

Change Server Port

See [corresponding section](#) in server installation instructions.

Test-drive Newer TeamCity Version before Upgrade

It's advised to try 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. When you start the test server do not forget to change the Server URL, disable Email and Jabber notifiers as well as [other features](#) on the new server.

Create a Copy of TeamCity Server with All Data

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. entire "artifacts" directory) from [.BuildServer/system](#) from the original to the copied server. This copying should occur before you create a backup to make sure newer directories do not appear on the server copy.

If you do not want to use bundled backup functionality or need manual control over the process, here is a description of the general steps one would need to perform to manually create copy of the server:

1. create a [backup](#) so that you can restore it if anything goes wrong,

2. ensure the server is not running,
3. either perform clean [installation](#) or copy the TeamCity binaries ([TeamCity Home Directory](#)) into a new place (the `temp` and `work` subdirectories can be omitted during copying). ⚠ Use exactly the same TeamCity version. If you plan to upgrade after copying, perform the upgrade only after you have the existing version up and running.
4. 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/artifacts` (optional) if you want build artifacts and build logs (including tests failure details) preserved on the new server
 - `.BuildServer/system/changes` (optional) if you want personal changes preserved on the new server
 - `.BuildServer/system/pluginData` (optional) if you want to preserve state of various plugins, build triggers and settings audit diff
 - `.BuildServer/system/caches` and `.BuildServer/system/caches` (optional) are not necessary to copy to the new server, they will be recreated on startup, but can take some time to be rebuilt (expect some slow down).
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. do not forget to update [Server URL](#) on [Administration | Global Settings](#) page and change [other settings](#) to prevent the copy server to clash with the original one.
3. check that VCS servers, issue tracker servers, email and Jabber server and other server-accessed systems are accessible from the new installation.
4. check that any systems configured to push events to TeamCity server (like VCS hooks, automated build triggering, monitors, etc.) are updated to know about the new server (if necessary)
5. install new agents (or select some from the existing ones) and configure them to connect to the new server (using the new server URL)

If you are creating a test server you need to ensure that the users and production systems are not affected. Typically, this means you need to:

- ensure the server has correct (changed) [Server URL](#);
- disable Email, Jabber (in "Administration > Notifier" sections) and may be also custom notifiers or change their settings to prevent new server from sending out notifications;
- ensure the same license keys are not used on several servers ([more on licensing](#));
- be sure not to run any builds which change (e.g. deploy to) production environments. This also typically includes Maven builds deploying to non-local repositories. You can prevent any builds from starting by pausing a [build queue](#);
- disable any plugins which push data into other non-copied systems based on TeamCity events (like commit status publishing);
- disable cloud integration (so that it does not interfere with the main server);
- disable functionality to [store project settings in VCS](#): set `teamcity.versionedSettings.enabled=false` internal property;
- consider significantly increasing [VCS checking for changes interval](#) (server-wide default and overridden in the VCS roots) or changing settings of the VCS roots to prevent them from contacting production servers.

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. If you are only going to run the server for testing purposes, 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/for production purposes, you need to purchase separate licenses for the second server.

Move TeamCity Projects from One Server to Another

If you need to move data to a fresh server without existing data, it is recommended to [move the server](#) or [copy](#) it and then delete the data which is not necessary on the new server.

If you need to join the data with already existing set, there is a dedicated feature to move projects with most of the associated data from one server to another: [Projects Import](#).

❖ [Notes are on manual move of the settings in case you even want to perform it](#)

Since TeamCity 8.0 it is possible to move *settings* of a project or a build configuration to another server with simple file copying. For earlier TeamCity versions see the [comment](#).

The two TeamCity servers (source and target) should be of exactly the same version (same build).

All the **identifiers** throughout all the projects, build configurations and VCS roots of both servers should be unique. If they are not, you can change them via web UI.

If entities with the same id are present on different servers, the entities are assumed to be the same. For example this is useful for having global set of VCS roots on all the servers.

To move settings of the project and all its build configuration from one server to another:

From the TeamCity [TeamCity Data Directory](#), copy the directories of corresponding projects (`.BuildServer\config\projects\<id>`) and all its parent projects to `.BuildServer\config\projects` of the target server.

This moves project settings, build configuration settings, VCS roots defined in the projects preserving the links between them.

If there are same-named files on the target server as those copied, this can happen in case of

- a) id match: same entities already exist on the target server, in which case the clashing files can be excluded from copying, or
- b) id clash: different entities happen to have same ids. In this case it should be resolved either by changing entity id on the source or target server to fulfill the uniqueness requirement.

The set of parent projects is to be identified manually based on the web UI or the directory names on disk (which by default will have the same prefix).

Note: It might make sense to keep the settings of the root project synchronized between all the servers (by synchronizing content of `.BuildServer\config\projects_Root` directory). For example, this will ensure same settings for the default cleanup policy on all the servers.

Further steps after projects copying might be:

- delete unused data in the copied parent projects (if any) on the target server
- use "server health" reports to identify duplicate VCS roots appeared in result of copying, if any
- archive the projects on the source server and adjust cleanup rules (to be able to see build's history, if necessary)

What is *not* copied by the approach above:

- pausing comment and user of the paused build configurations
- archiving user of the archived projects
- global server settings (e.g. Maven settings.xml profiles, tools (e.g. handle.exe), external change viewers, build queue priorities, issue trackers). These are stored under various files under `.BuildServer\config` directory and should be synchronized either on the file level or by configuring the same settings in the server administration UI.
- project association with agent pools
- templates from other projects which are not parents of the copied one. This configuration is actually deprecated in TeamCity 8.0 and is only supported as legacy. Templates used in several projects should be moved to the common parent project or root project.
- no data configured for the agents (build configurations allowed to run on the agent).
- no user-related or user group-related settings (like roles and notification rules)
- no state-related data like mutes and investigations, etc.

Move TeamCity Installation to a New Machine

If you need to move 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. You might need to reduce DNS server cache/lease time in advance before the change to make the clients "understand" the change fast.

However, if you need to use another server domain address, you will need:

- 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 set to off.

[Read more](#) about dependency properties.

Please [watch/comment](#) the issue related to sharing a build number [TW-7745](#).

Make Temporary Build Files Erased between the Builds

Update your build script to use path stored in `${teamcity.build.tempDir}` (Ant's style name) property as the temp directory. TeamCity agent creates the [directory](#) before the build and deletes it right after the build.

Clear Build Queue if It Has Too Many Builds due to a Configuration Error

Try pausing the build configuration that has the builds queued. On build configuration pausing all its builds are removed from the queue. Also there is an ability to delete many builds from the build queue in a single dialog.

Automatically create or change TeamCity build configuration settings

If you need a level of automation and web administration UI does not suite your needs, there several possibilities:

- use [REST API](#)
- change configuration files directly on disk (see more at [TeamCity Data Directory](#))
- write a TeamCity Java plugin that will perform the tasks using open API.

Attach Cucumber Reporter to Ant Build

If you use Cucumber for Java applications testing you should run cucumber with `--expand` and special `--format` options. 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/rake-runner/rb/patch/bdd"/>
        <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](#)

Set up Deployment for My Application in TeamCity

TeamCity has enough features to handle orchestration part of the deployments with the actual deployment logic configured in the build script / build runner. TeamCity supports a variety of generic build tools, so any specific tool can be run from within TeamCity. To ease specific tool usage, it is possible to wrap it into a meta-runner or write a custom plugin for that.

In general, setup steps for configuring deployments are:

1. Write a build script that will perform the deployment task for the binary files available on the disk. (e.g. use Ant or MSBuild for this. For FTP/SSH tasks check the [Deployer plugin](#)) 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 in the deploying build configuration 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
- If necessary, use parameter with "prompt" display mode to ask for "confirmation" on running a build
- [Change title](#) of the build configuration "Run" button

Related section on the official site: [Continuous Deployment with TeamCity](#)

Use an External Tool that My Build Relies on

If you need to use specific external tool to be installed on a build agent to run your builds, you have the following options:

- Install and register the tool in TeamCity:
 1. Install the tool on all the agents that will run the build. This can be done manually or via an automated script. For simple file distribution also see [Installing Agent Tools](#)
 2. Add a property into `buildAgent.properties` file (or add environment variable to the system) with the tool home location as the value.
 3. Add agent requirement for the property in the build configuration.
 4. Use the property in the build script.
- Check in the tool into the version control and use relative paths.
- Add environment preparation stage into the build script to get the tool from elsewhere.
- Create a separate build configuration with a single "fake" build which would contain required files as artifacts, then use artifact dependencies to send files to the target build.

Integrate with Build and Reporting Tools

If you have a build tool or a tool that generates some report/provides code metrics which is not yet [supported by TeamCity](#) or any of the [plugins](#), most probably you can use it in TeamCity even without dedicated integration.

The integration tasks involved are collecting the data in the scope of a build and then reporting the data to TeamCity so that they can be presented in the build results or in other ways.

Data collection

The easiest way for a start is to modify your build scripts to make use of the selected tool and collect all the required data.

If you can run the tool from a command line console, then you can run it in TeamCity with a [command line runner](#). This will give you detection of the messages printed into standard error output. The build can be marked as failed if the exit code is not zero or there is output to standard error via [build failure condition](#).

If the tool has launchers for any of the supported build scripting engines like Ant, Maven or MSBuild, then you can use corresponding runner in TeamCity to start the tool.

See also [#Use an External Tool that My Build Relies on](#) for the recommendations on how to run an external tool.

You can also consider creating a [Meta Runner](#) to let the tool have dedicated UI in TeamCity.

For an advanced integration a custom [TeamCity plugin](#) can be developed in Java to ease tool configuration and running.

Presenting data in TeamCity

The build progress can be reported to TeamCity via [service messages](#) and build status text can also be [updated](#).

For testing tools, if they are not yet [supported](#) you can report tests progress to TeamCity from the build via [test-related service messages](#) or generate one of the supported [XML reports](#) in the build and let it be imported via a service message of configured XML Report Processing build feature.

To present the results for a generic report, the approach might be to generate HTML report in the build script, pack it into archive and publish as a build artifact. Then configure a [report tab](#) to display the HTML report as a tab on build's results.

A metrics value can be published as TeamCity statistics via [service message](#) and then displayed in a [custom chart](#). You can also configure [build failure condition](#) based on the metric.

If the tool reports code-attributing information like Inspections or Duplicates, TeamCity-bundled report can be used to display the results. A custom plugin will be necessary to process the tool-specific report into TeamCity-specific data model. Example of this can be found in [XML Test Reporting plugin](#) and [FXCop plugin](#) (see a link on [Open-source Bundled Plugins](#)).

See also [#Import coverage results in TeamCity](#).

For advanced integration, a custom plugin will be necessary to store and present the data as required. See [Developing TeamCity Plugins](#) for more information on plugin development.

Restore Just Deleted Project

TeamCity moves settings files of deleted projects under `TeamCity Data Directory/config/_trash` directory. To restore project you should find the directory on the server and move it into regular projects settings directory: `<TeamCity Data Directory>/config/projects`. 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. All the associated data (builds and test history, changes, etc.) is removed during the next cleanup after 24 hours timeout elapses.

Transfer 3 Default Agents to Another Server

This is not possible.

Each TeamCity server (Professional and Enterprise) allows using 3 or more agents bound to the server without any agent licenses. In case of the Professional server, by default 3 agents are bound to the server instance: users do not pay for these agents, there is no license key for them.

In case of the Enterprise server, the number of agents depends on your package and the agents are bound to the server license key.

So, the agents bound to the server cannot be transferred to another server.

If you need more build agents that are included with your TeamCity server, you can purchase additional build agent licenses and connect more agents in addition to those that come bound with the server.

See [more](#) on licensing.

Import coverage results in TeamCity

TeamCity comes bundled with IntelliJ IDEA/Emma and, JaCoCo coverage engines for Java and dotCover/NCover/PartCover for .NET. However, there are plenty of other coverage tools out there, like [Cobertura](#) and others which are not directly supported by TeamCity.

In order to achieve similar experience with these tools you can:

- publish coverage HTML report as TeamCity artifact: most of the tools produce coverage report in HTML format, you can publish it as artifact and [configure report tab](#) to show it in TeamCity. If published artifact name is `coverage.zip` and there is `index.html` file in it, report tab will be shown automatically. As to running an external tool, check [#Integrate with Build and Reporting Tools](#).
- extract coverage statistics from coverage report and publish [statistics values](#) to TeamCity with help of [service message](#): if you do so, you'll see coverage chart on build configuration Statistics tab and also you'll be able to fail a build with the help of a build failure condition on a metric change (for example, you can fail build if the coverage drops).



Percentage values

You should not publish values `CodeCoverageB`, `CodeCoverageL`, `CodeCoverageM`, `CodeCoverageC` standing for block/line/method/class coverage percentage. TeamCity will calculate these values using their absolute parts. E.g. `CodeCoverageL` will be calculated as `CodeCoverageAbsLCovered` divided by `CodeCoverageAbsLTotal`.

You could publish these values but in this case they will lack decimal parts and will not be useful.

Recover from "Data format of the data directory (NNN) and the database (MMM) do not match" error

If you get "Data format of the data directory (NNN) and the database (MMM) do not match." error on starting TeamCity, it means either the database or the TeamCity Data Directory were recently changed to an inconsistent state so they cannot be used together.

Double-check the database and data directory locations and change them if they are not those where the server used to store the data.

If they are right, most probably it means that the server was upgraded with another database or data directory and the [consistent upgrade](#) requirement was not met for your main data directory and the database.

To recover from the state you will need backup of the consistent state made prior to the upgrade. You will need to restore that backup, ensure the right locations are used for the data directory and the database and perform the TeamCity upgrade.

Debug a Build on a Specific Agent

In case a build fails on some agent, it is possible to debug it on this very agent to investigate agent-specific issues. Do the following:

- Go to the [Agents](#) page in the TeamCity Web UI and [select the agent](#).
- [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.
- [Select the build](#) to debug.
- Open the [Custom Run](#) dialog and specify the following options:
 - In the [Agent](#) drop-down, select the disabled agent.
 - It is recommended to select the [run as Personal Build](#) option to avoid intersection with regular builds.
- When debugging is complete, enable the agent manually if automatic re-enabling has not been configured.

You can also perform remote debugging of tests on an agent via the [IntelliJ IDEA plugin](#) for TeamCity.

Debug a Part of the Build (a build step)

If a build containing several steps fails at a certain step, it is possible to debug the step that breaks. Do the following:

1. Go to the build configuration and disable the build steps up to the one you want to debug.
2. Select the build to debug.
3. Open the [Custom Run](#) dialog and select the **put the build to the queue top** to give you build the priority.
4. When debugging is complete, re-enable the build steps.

Vulnerabilities

This section describes effect and necessary protection steps related to recently announced security vulnerabilities.

Heartbleed, ShellShock

TeamCity distributions provided by JetBrains do not contain software/libraries and do not use technologies affected by Heart bleed and Shell shock vulnerabilities.

What might still need assessment is the specific TeamCity installation implementation which might use the components behind those provided/recommended by JetBrains and which can be vulnerable to the mentioned exploits.

POODLE

If you configured HTTPS access to the TeamCity server, inspect the solution used for HTTPS as that might be affected (e.g. Tomcat seems to be affected). At this time none of TeamCity distributions include HTTPS access by default and investigating/eliminating HTTPS-related vulnerability is out of scope of TeamCity.

Depending on the settings used, TeamCity server (and agent) can establish HTTPS connections to other servers (e.g. Subversion). Depending on the server settings, those connections might fall back to using SSL 3.0 protocol. The recommended solution is not TeamCity specific and it is to disable SSLv3 on the target SSL-server side.

GHOST

CVE-2015-0235 vulnerability is found in glibc library which is not directly used by TeamCity code. It is used by the Java/JRE used by TeamCity under *nix platforms. As Java is not bundled with TeamCity distributions, you should apply the security measures recommended by the vendor of the Java you use. At this time there are no related Java-specific security advisories released, so updating the OS should be enough to eliminate the risk of the vulnerability exploitation.

FREAK

CVE-2015-0204 vulnerability is found in OpenSSL implementation and TeamCity does not bundle any parts of OpenSSL product and so is not vulnerable. You might still need to review the environment in which TeamCity server and agents are installed as well as tools installed in addition to TeamCity for possible vulnerability mitigation steps necessary.

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.

TeamCity Release Cycle

The information below can be used for reference purposes only.

"major" release below means any release with a change in first or second version number (e.g. X in X.X.Z)
"bugfix" release means releases with a change in the third version number (e.g. Z in X.X.Z)

Release stages that we generally have are:

Available under EAP (Early Access Program) - usually available only for major releases, starts several months after previous major release and usually months before the next major release. Typically new EAP releases are published on monthly or bi-monthly basis.

General Availability - as a rule, there are two major releases each year. There are multiple bugfix releases following the major release. Bugfix releases and support patches for critical issues (if applicable) are provided until "End of Sale" of the release.

End of Sale - occurs with the release of a new major version. After this time no bugfix updates or patches are usually provided (Exceptions are critical issues without workaround which allow for relatively simple fix and inability for the customer to upgrade for an important reason). Only limited support is provided for these versions.

End of Support - occurs with the release of two newer major versions. At this point we stop providing email support for the release.

Dates for the previous releases can be seen at [Previous Releases Downloads](#).

Troubleshooting

When a problem occurs, you have a number of places to look for information after you've found out the problem isn't in setup:

- Check the [Known Issues](#) and [Common Problems](#) sections, collect relevant information using the [Reporting Issues](#) guidelines.
- [The TeamCity Forum](#) - Search the forum to see if anyone else has experienced your problem. Our forum's user base is quite active and is a good place to find support. If you cannot find any relevant information either in the forums or in tracker and you are not sure whether you faced a bug or it's just a result of misconfiguration, the right way to start is to create a new thread in the forums.
- [TeamCity's Issue Tracker](#) - Browse the issue tracker to see if somebody has already reported on your problem. If the same issue exists, please vote for the issue. If you are sure you have faced a bug, please [collect](#) the relevant data about the problem and post a new issue into the tracker. Be sure to include the TeamCity build number, describe where exactly you see the problem, what your previous actions were if relevant and also please describe your environment (OS, Web Server, TeamCity distribution used, how TeamCity is set up, etc.)
- [Contact us](#) to report an issue or ask a question using the general guidelines described.
- If you own Enterprise TeamCity license and need to submit information that is not meant to be public, you can also contact the development team via [Online Form](#) or [Feedback email](#).



When contacting us make sure to:

- include the affected TeamCity version;
- include detailed exact error messages, logs, screenshots;
- mention all related postings on the topic.

See also:

[Troubleshooting: Known Issues](#) | [Reporting Issues](#)

Common Problems

- Most frequently used documentation sections
- Build 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
 - "Unique key violations" or "Duplicates found" error on restore from backup

- Character set/collation-related problems
- 'This driver is not configured for integrated authentication' error with MS SQL database
- Protocol violation error (Oracle only)
- Common Maven issues
- "Critical error in configuration file" errors
- TeamCity installation problems
- Problems with TeamCity NuGet Feed

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. If the reason is "Agent has unregistered (will upgrade)", then wait for several more minutes
- the agent is under Unauthorized - all the agents connected to the server for the first time should be authorized by a server administrator

If the agent stays in the state for more than 10 minutes and you have a fast network connection between the agent and the server, please:

- check the agent process is running and the `serverURL` in `conf/buildAgent.properties` is correct;
- check that all the [requirements](#) are met;
- check [agent logs](#) (`teamcity-agent.log`, `launcher.log`, `upgrade.log`) for any related messages/errors;
- check [server logs](#) (`teamcity-server.log`) for any messages/errors mentioning agent name or IP.

If you cannot find the cause of the delayed agent upgrade in the logs, [contact us](#) and provide the full agent and server logs. Please also check/include the state of the agent processes (java ones) on the agent machine.

[Back to top](#)

Artifacts of a build are not cleaned

If you encounter a case when artifacts are preserved while they should have been removed by the server cleanup process, please check:

- the cleanup rules of the build configuration in question, artifacts cleanup section
- presence of the icon "This build is used by other builds" in the build history line (prior to Pin action/icon on Build History)
- build's Dependencies tab, "Delivered Artifacts" section. For every build configuration, check whether "Prevent dependency artifacts clean-up" is turned ON (this is default value). If it is, then the build's artifacts are not cleaned because of the setting.
Read more on [cleanup settings](#).

[Back to top](#)

Database-related issues

"out of memory" error with internal (HSQLDB) database

If during the TeamCity server start-up you encounter errors like:

- "error in script file line: ... out of memory"
- "java.sql.SQLException: out of memory",
perform the following:
- try [increasing server memory](#). If this does not help, most probably this means that you have encountered [internal database corruption](#). You can try to deal with this corruption using the [notes](#) based on the HSQLDB documentation.

A way to attempt a manual database restore:

- stop the TeamCity server
- backup the `<TeamCity Data Directory>/system/buildserver.data` file
- remove the `<TeamCity Data Directory>/system/buildserver.data` file and replace it with zero-size file of the same name
- start the TeamCity server

However, if the database does not recover automatically, chances that it can be fixed manually are minimal.

The internal (HSQL) database is not stable enough for production use and we highly recommend using an [external database](#) for TeamCity non-evaluation usage.

If you encountered database corruption, you can restore the last good backup or drop builds history and users, but preserve the settings, see [Migrating to an External Database#Switching to Another Database](#).

The transaction... log is full

This error can occur with an MS SQL or Sybase database. In this case we recommend increasing the transaction log for the TeamCity database. The log size can be 1 - 16 GB depending on the number of build agents in the system and the number of tests all agents report daily.

The table 'table_name' is full

This error can occur with a MySQL database. The error indicates that the database has run out of free space either on the disk where the database files are located or in the temp directory.

Unable to extend ... segment ... in tablespace ...

This error can occur with an Oracle database. The error indicates that Oracle could not obtain more space for a table or an index as the database has run out of space on the disk or the specified quotas are insufficient.

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.

"Unique key violations" or "Duplicates found" error on restore from backup

TeamCity server restoration from a backup may fail with errors like "Unique key violation" or "Duplicates found" if the character sets (or collations) of the source and destination databases are not the same, and the cardinality of the destination character set is less than that of the source database.

To resolve the problem, select and set up the proper character set (and collation) for the destination database.

As for case sensitivity, the possible transitions are:

CS CS

CI CS

CI CI

However, it is recommended to always use CS.

If the source character set is Unicode or UTF, the destination one must also be Unicode or UTF.

If the source character set is 8-bit non-UTF, the destination one can be the same or Unicode/UTF.

This applies to TeamCity 6.0 and above.

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

'This driver is not configured for integrated authentication' error with MS SQL database

During TeamCity installation, the following error might occur when connecting and creating the MS SQL database with integrated security: "SQL error when doing: Taking a connection from the data source: This driver is not configured for integrated authentication."

The most common reason for the problem is the different bitness of the `sqljdbc_auth.dll` MS SQL shared library and the JRE used by TeamCity.

To solve the problem, do the following:

1. Make sure you use the MS SQL native driver (downloadable from [the Microsoft Download Center](#)).
2. Use the right JRE bitness — ensure that you are running TeamCity using Java with the same bitness as your `sqljdbc_auth.dll` MS SQL shared library.

By default, TeamCity uses the 32-bit Java. However, both 32-bit and 64-bit Java versions [can be used](#).

To run TeamCity with the required JRE, do one of the following:

- either set the `TEAMCITY_JRE` environment variable
- or remove the JRE bundled with TeamCity from `<TeamCity home>/jre` and set `JAVA_HOME`.



Note that on upgrade, TeamCity will overwrite the existing JRE with the default 32-bit version, so you'll have to update to the 64-bit JRE again after upgrade.

See also this related [external posting](#).

[Back to top](#)

Protocol violation error (Oracle only)

This error can occur when the Oracle JDBC driver is not compatible with the Oracle server. For example, Oracle JDBC driver version 11.1 is not compatible with Oracle server version 10.2.

In order to resolve the problem, use the Oracle JDBC driver from your Oracle server installation, or [download the driver](#) of the same version as the Oracle server.

Common Maven issues

There are two kinds of Maven-related issues commonly seen in the TeamCity build configurations:

- Error message on "Maven" tab of build configuration: "An error occurred during collecting Maven project information ... "
- Error message in build configuration with Maven dependencies trigger activated: "Unable to check for Maven dependency Update ..."

If the build configuration produces successful builds despite displaying such error messages, these errors are likely to be caused by the **server-side Maven misconfiguration**.

To collect information for the **Maven** tab, or to perform Maven dependencies check (for the trigger), TeamCity runs the embedded Maven. The execution is performed on the server machine, and any *agent-side* maven settings are **not accessible**. TeamCity resolves the `settings.xml` files on the server-side separately, as described [on this documentation page](#).

It makes sense to check if the `server-side settings.xml` files contain correct information about remote repositories, proxies, mirrors, profiles, credentials etc.

[Back to top](#)

"Critical error in configuration file" errors

If you encounter the error, it means the settings stored in the TeamCity Data Directory are in inconsistent state. This can occur after manual change of the files or if newer version of TeamCity starts to report the inconsistencies.

To resolve the issue, you can edit the file noted in the message on the server file system. (make sure to create backup copy of the file before any manual edits). Usually server restart is not necessary for the changes to take effect.

VCS root with id "XXX" does not exist

The build configuration or template reference a VCS root which is not defined in the system.

Remedy actions: Restore the VCS root or create a new VCS root with the id noted or edit the file noted in the message to remove the reference to the VCS root.

[Back to top](#)

TeamCity installation problems

If the TeamCity Web UI cannot be accessed after installation, you might be running TeamCity on a port that is already in use by another program. Check and configure your TeamCity installation.

[Back to top](#)

Problems with TeamCity NuGet Feed

If you are experiencing issues with partial TeamCity NuGet Feed, i.e. missing NuGet packages etc., you might have to reindex the TeamCity NuGet Feed.

Since TeamCity 9.0, to force TeamCity to reindex all available packages and reset the NuGet package list, navigate to the server **Administration** | **Diagnostics** | **Caches** and use the [buildsMetadata Reset](#) link.

For earlier versions, refer to [this section](#).

[Back to top](#)

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
 - 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
- SSL problems when connecting to HTTPS from TeamCity (handshake alert: unrecognized_name)

Agent running as Windows Service Limitations

When a TeamCity build agent is installed as a Windows service, there may appear various "Permission denied" or "Access denied" errors during the build process, see details below.

Security-related issues

The user account used by the service is required to have sufficient permissions to perform the build and [manage the service](#). If you run the TeamCity agent service under the SYSTEM account, do the following:

1. Change SYSTEM for a usual user account with necessary permissions granted.
2. Restart the service.

Windows service limitations

As a Windows service, the TeamCity agent and the build processes are not able to access network shares and mapped drives.

To overcome these restrictions, run TeamCity agent [via console](#).

Issues with automated GUI and browser testing

These problems include errors running tests headless, issues with the interaction of the TeamCity agent with the Windows desktop, etc.

To resolve/ avoid these:

1. Run TeamCity agent [via console](#).
2. Configure the build agent machine not to launch a screensaver locking the desktop.



Note that there is a Windows limitation to accessing a remote computer via mstsc: the desktop of the remote machine will be locked on RDP disconnect, which will cause issues running tests. The VNC protocol allows you to remote control another machine without locking it.

3. Configure the TeamCity agent to start automatically (e.g. configure an automatic user logon on Windows start and then configure the TeamCity agent start ([via agent.bat start](#)) on the user logon).



An unsupervised computer with a running desktop permanently logged into a user session might be considered a network security threat, as access to it can be difficult to trace. Therefore, it is recommended to run automated GUI and browser tests on a virtual machine isolated from sensitive corporate network resources, e.g. on a machine not included in a Windows domain.

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. e.g. see [the issue](#).

ESET antivirus can also slow down Ant/IntelliJ IDEA project builds a great deal (slowing down TCP connections to localhost on agent).

If you run antivirus on the TeamCity server or agent machines and get disk access errors or experience degraded performance, please disable or better completely uninstall the antivirus software before investigating the issue and reporting the issue to JetBrains.

It is recommended to exclude entire TeamCity server home and [TeamCity Data Directory](#) from the background checks and perform periodical checks there in the well-known maintenance window so that those do not affect server performance much.

On TeamCity agent, it is recommended to exclude TeamCity agent home from the background checks.

Please disable various indexing services. e.g. there might be problems with Windows Indexing Service. See [issue](#) for more details. Windows System Restore Feature might also need disabling.

Please also do not install software with background indexing like WinCVS, TortoiseCVS, TortoiseSVN and other Tortoise* products. This applies to server and also to agents if you use agent-side checkout.

Skype software is known to:

- use port 80 on the system so you might not be able to use TeamCity server using default 80 port.
- corrupt layout of pages displayed in Internet Explorer. Internet Explorer Skype plugin is to blame. ([TW-13052](#)).

[Back to top](#)

Subversion issues

svn: E175002: Received fatal alert: bad_record_mac

Please add system property -Dsvnkit.http.sslProtocols=SSLv3,TLS on the build server (see [Configuring TeamCity Server Startup Properties](#)). If you use checkout on agent, add this property [on build agent](#) as well.

Subversion-related JVM Crashes

If JVM crashes while executing SVN-related code (e.g. under org.tmatesoft.svn package), you can try to disable it by either:

- Passing -Dsvnkit.useJNA=false JVM option to the crashing process (server or agent), or
- Making NTLM support less prioritative by passing -Dsvnkit.http.methods=Basic,Digest,NTLM JVM option.

Anyway, upgrading the JVM used to the [latest available version](#) is recommended.

[Back to top](#)

NUnit 2.4.6 Performance

Due to an issue in NUnit 2.4.6, its performance may be slower than NUnit 2.4.1. For additional information, please refer to the corresponding issue in our issue tracker: [TW-4709](#)

[Back to top](#)

StarTeam Performance

Using StarTeam SDK 9.0 instead of StarTeam SDK 9.3 on the TeamCity server can significantly improve VCS performance when there is a slow connection between TeamCity and StarTeam servers.

[Back to top](#)

Perforce 2009.2 Performance on Windows

If you run Perforce 2009.2 on Windows you may experience significant slow down. This is an issue with P4 server running on Windows. Please refer to corresponding [section](#) in Perforce documentation.

Wrong times for build scheduled triggering (Timezone issues)

Please make sure you use the latest 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.1ck` file from the [TeamCity Data Directory](#)\`system` directory, and then start TeamCity again.

[Back to top](#)

Slow download from TeamCity server

If you experience slow speed when downloading artifacts from TeamCity, try checking the speed on the server machine, downloading from localhost.

If the speed is OK for the localhost, the issue can be in the network configuration or OS/hardware settings when combined with TeamCity(Tomcat) settings.

Please also make sure <[TeamCity Home](#)>\conf\server.xml file corresponds to the file included in TeamCity distribution (can be checked in .tar.gz distribution).

If you have the following "Connector" node (ports numbers can be different):

```
<Connector port="8111" protocol="HTTP/1.1"
           connectionTimeout="60000"
           redirectPort="8543"
           useBodyEncodingForURI="true"
        />
```

change it to:

```
<Connector port="8111" protocol="org.apache.coyote.http11.Http11NioProtocol"
           connectionTimeout="60000"
           redirectPort="8543"
           useBodyEncodingForURI="true"
           socket.txBufSize="64000"
           socket.rxBufSize="64000"
           tcpNoDelay="1"
        />
```

and restart TeamCity server.

If this does not help with the download speed, to investigate the case you might need to find an administrator with appropriate network-related issues investigation skills to look into the case.

Failure to publish artifacts to server behind IIS reverse proxy

This problem is only relevant to configurations that involve IIS reverse proxy between build server and agents.

Sometimes a build agent can be found in infinite loop trying to publish build artifacts to server. Build log looks like this:

```
[11:15:05]Publishing artifacts
[11:15:05][Publishing artifacts] Collecting files to publish: [toZip/** =>
artifact.zip]
[11:15:06][Publishing artifacts] Creating archive artifact.zip (9s)
[11:15:06][Creating archive artifact.zip] Creating
C:\BuildAgent\temp\buildTmp\ZipPreprocessor2847146024236637664\artifact.zip
[11:15:15][Creating archive artifact.zip] Archive was created, file size 32142324
bytes
[11:15:15][Publishing artifacts] Sending toZip/**
[11:15:25][Publishing artifacts] Sending toZip/**
[11:15:39][Publishing artifacts] Sending toZip/**
[11:15:48][Publishing artifacts] Sending toZip/**
[11:16:01][Publishing artifacts] Sending toZip/**
[11:16:16][Publishing artifacts] Sending toZip/**
```

meanwhile teamcity-agent.log is filled with 404 responses from IIS:

```
[ 2012-08-01 12:04:55,514]    WARN -      jetbrains.buildServer.AGENT - <!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
<title>404 - File or directory not found.</title>
<style type="text/css">
<!--
body{margin:0;font-size:.7em;font-family:Verdana, Arial, Helvetica,
sans-serif;background:#EEEEEE;}
fieldset{padding:0 15px 10px 15px;}</style>
```

The most common cause for this is **maxAllowedContentLength** setting (in IIS) either

- is set to too small value
- is left unconfigured and so defaults to 30000000 bytes (<30 Mb)

So any artifact larger than maxAllowedContentLength is discarded by IIS
Check the settings value and try to rerun your build

SSL problems when connecting to HTTPS from TeamCity (handshake alert: unrecognized_name)

This problem may happen when changing JVM from 1.6 to 1.7 and connecting some incorrectly configured HTTPS servers.
The problem and workaround for it are described in this issue: <http://youtrack.jetbrains.com/issue/TW-30210>

Please try running with antivirus software uninstalled before reporting the issue to JetBrains. e.g. see [the issue](#).

Reporting Issues

If you experience problems running TeamCity and believe they are related to the software, please [contact us](#) with a detailed description of the issue.

To fix a problem, we may need a wide range of information about your system as well as various logs. The section below explains how to collect such information for different issues.

In this section:

- Best Practices When Reporting Issues

- Slowness, Hangings and Low Performance
 - Server Thread Dump
 - Collecting CPU Profiling Data on Server
 - Agent Thread Dump
 - Taking Thread Dump
 - Database-related Slowdowns
- OutOfMemory Problems
- "Too many open files" Error
- Agent does not connect to the server
- Logging events
 - Version Control debug logging
- Patch Application Problems
- Logging for .NET Runners
- Remote Run Problems
- Logging in IntelliJ IDEA/Platform-based IDEs
 - Open in IDE Functionality Logging
 - No Suitable Build Configurations Found for Remote Run
- Logging in TeamCity Eclipse plugin
- TeamCity Visual Studio Addin issues
 - TeamCity Addin logging
 - Visual Studio logging
- dotCover Issues
- JVM Crashes
- Build Log Issues
- Uploading Large Data Archives

Best Practices When Reporting Issues

Following these guidelines will ensure timely response and effective issue resolution. Check [Feedback](#) for appropriate ways to contact us.



- note the **TeamCity version** in use, including the build number (can be found in the footer and the teamcity-server.log). Consider checking if the issue is still relevant in the most recent TeamCity version;
- note all previous postings on the same topic you have made or found;
- do not combine several issues into one posting;
- note the pattern of issue occurrence (first time, recurring), how it was mitigated before, whether there were any recent environment changes;
- describe expected and actual behavior;
- include related **screenshots** (always include the entire page and the browser URL in the capture);
- include related text messages as text (not as image), include the messages with all the details;
- attach/upload archive with TeamCity **server logs** (entire <server home>\logs directory is the best); if related to the build-time or agent behavior, attach entire <agent home>\logs directory;
- for performance/slowness/delays issues take a set of server or agent **thread dumps** during the issue occurrence;
- when sending files greater than 500Kb in size or more than three files, package them into a single archive;
- when replacing/masking data in logs, note the replacements patterns used;
- note if there is an anti-virus installed and if there is a network proxy;
- when relevant, note OS, versions of any manually installed components like Java, used TeamCity distribution (.exe, .tar.gz)
- note any customization/not standard environment settings;
- when reporting build issues, note whether they are **reproducing** when the build is run without TeamCity on the agent machine;
- when suggesting an improvement or feature or asking settings advice, detail *why* you need the feature and what is the **original goal** you want to achieve. Suggestions as to *how* you would like to see the feature are welcome too;
- check the sections below for common cases and specific information to collect and send to us.

Slowness, Hangings and Low Performance

If TeamCity is running slower than you would expect, please use the notes below to locate the slow process and send us all the relevant details if the process is a TeamCity one.

Determine Which Process Is Slow

If you experience a slow TeamCity web UI response, checking for changes process, server-side sources checkout, long cleanup times or other slow server activity, your target should be the machine where the TeamCity server is installed.

If the issue is related only to a single build, you will 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 that is loading the CPU/IO or there is no substantial CPU/IO load and everything runs just fine except for TeamCity, then this is to be investigated further.

Please check if you have any [Conflicting Software](#) like anti-virus running on the machine and disable/uninstall it.

If you have a substantial TeamCity installation, please check you have appropriate [memory settings](#) as the first step.

Collect Data

During the slow operation, take several thread dumps of the slow process (see below for thread dump taking approaches) with 5-10 seconds interval. If the slowness continues, please take several more thread dumps (e.g. 3-5 within several minutes) and then repeat after some time (e.g. 10 minutes) while the process is still being slow.

Then [send](#) us a detailed description of the issue accompanied with the thread dumps and full server (or agent) [logs](#) covering the issue. Unless it is undesirable for some reason, the preferred way is to file an issue into our [issue tracker](#) and let us know via feedback email. Please include all the relevant details of investigation, including the CPU/IO load information, what specifically is slow and what is not, note URLs, visible effects, etc. For large amounts of data, please use [our FTP](#).

Server Thread Dump

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 adjust the `teamcity.diagnostics.requestTime.threshold.ms=30000` [internal property](#) to change the timeout after which a thread dump is automatically created in the `threadDumps-<date>` directory under TeamCity logs whenever there is a user-originated web request taking longer than timeout.

Collecting CPU Profiling Data on Server

If you experience degraded server performance and the TeamCity server process is producing a large CPU load, take a CPU profiling snapshot and send it to us accompanied with the detailed description of what you were doing and what your system setup is.

You can take CPU profiling and memory snapshots by installing the [server profiling plugin](#) and following the instructions on the plugin page.

Here are some hints to get the best results from CPU profiling:

- after starting the server, wait for some time to allow it to "warm up". This can take from 5 to 20 minutes depending on the data volume that TeamCity stores.
- when a CPU usage increase is found on the server, please try to indicate what actions cause the load.
- start CPU profiling and repeat the action several times (5 - 10).
- capture a snapshot.
- archive the snapshot and send it to us including the description of the actions that cause the CPU load.

Agent Thread Dump

It is recommended that you take an agent thread dump from the Web UI: go to the Agent page, **Agent Summary** tab, and use the **Dump threads on agent** action.

If the UI is not accessible, you can take the dump thread manually using the approaches described [below](#). Note that the TeamCity agent consists of two `java` processes: the launcher and agent itself. The agent is triggered by the launcher. You will usually be interested in the agent (nested) process and not the launcher one.

Taking Thread Dump

These can help if you are unable to take a thread dump from the TeamCity web UI.

To take a thread dump:

Under Windows

You have several options:

- To take a server thread dump if the server is run from the console, press **Ctrl+Break** in the console window (this will not work for an agent, since its console belongs to the launcher process).
- Alternatively, run `jstack <pid_of_java_process>` in the `bin` directory of the Java installation used to by the process (the Java home can be looked up in the process command line. If the installation does not have `jstack` utility, you might need to get the java version via `java -version` command, download full JDK of the same version and use "`jstack`" utility form there). You might also need to supply "-F" flag to the command.
- Yet another approach is to use TeamCity-bundled agent thread dump tool (can be found in the agent's plugins). Run the command:

```
<TeamCity  
agent>\plugins\stacktracesPlugin\bin\x86\JetBrains.TeaмCity.Injector.exe  
<pid_of_java_process>
```

Note that if the hanging process is run as a service, the thread dumping tool must be run from a console with elevated permissions (using Run as Administrator). If the service is run under System account, you might also need to launch the thread dumping tools via "`PsExec.e xe -s <path to the tool>\<tool> <options>`". When the service is run under a regular user, wrapping the tool invocation in `PsExec.exe -u <user> -p <password> <path to the tool>\<tool> <options>` might also help.

If neither of these work for the server running as a service, try [running the server](#) from console and not as a service. This way the first (Ctrl+Break) option can be used.

Under Linux

- run `jstack <pid_of_java_process>` (using `jstack` from the Java installation as used by the process) or `kill -3 <pid_of_java_process>`. In the latter case output will appear in `<TeamCity Home>/logs/catalina.out` or `<TeamCity agent home>/logs/error.log`.

See also [Server Performance](#) section 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). You can track memory and CPU usage by TeamCity with the charts on the [Administration | Server Administration | Diagnostics](#) page of your TeamCity web UI.
- If the server is to blame, please check you have increased memory settings from the default ones for using the server in production (see [this section](#)).
- If the build process is to blame, set "JVM Command Line Parameters" settings in the build runner. Increase value for '`-Xmx`' JVM option, like `-Xmx1200m`. If the error message is "java.lang.OutOfMemoryError: PermGen space", increase the value in `-XX:MaxPermSize=270m` JVM option. e.g. Java Inspections builds may specifically need to increase `-Xmx` value;
- If the TeamCity server is to blame and increasing the memory size does not help, please report the case for us to investigate. For this, while the server is high on memory consumption, take several server thread dumps as described [above](#), get the memory dump (see below) and all the server logs, archive the results and [send them](#) to us for further analysis. If you have increased the `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, the `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 a memory dump manually is to run the TeamCity server with JDK 1.6+ and use the `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 the process limits on the file handles (on Linux use `ulimit -n`) and increase them if necessary. Please note that default Linux 1024 handles per process is way too small for a server application like TeamCity. Please increase the number to at least 16000. Please check the actual process limits after the change as there are different settings in the OS for settings global and per-session limits (e.g. see [the post](#))

If the number of files is large and looks suspicious and the locking process is a TeamCity one (the TeamCity agent or server with no other web applications running), then, while the issue is still occurring, grab the list of open handles several times with several minutes interval and send the result to us for investigation together with the relevant details.

Please note that you will most probably need to reboot the machine with the error after the investigation to restore normal functioning of the applications.

Agent does not connect to the server

Please refer to [Common Problems#Started Build Agent is not available on the server to run builds](#)

Logging events

The TeamCity server and agent create logs that can be used to investigate issues.



How to enable DEBUG logging on server?

Before reproducing the problem it makes sense to enable 'DEBUG' log level for TeamCity classes.

On the server side, go to the [Administration | Server Administration | Diagnostics](#) page and select logging preset ('debug-all', 'debug-vcs', etc).

After that, DEBUG messages will go to `teamcity-* .log` files ([read more](#)).

For detailed information, please refer to the corresponding sections:

[TeamCity Server Logs](#)
[Viewing Build Agent Logs](#)

[Back to top](#)

Version Control debug logging



To enable VCS logging on the server side, switch logging preset to "debug-vcs" in administration web UI and then retrieve `logs/teamcity-vcs.log` log file.

Most VCS operations occur on the TeamCity server, but if you're using the [agent-side checkout](#), VCS checkout occurs on the build agents.

For the agent and the server, you can change the Log4j configuration manually in [<TeamCity Home>/conf/teamcity-server-log4j.xml](#) or [<BuildAgent home>/conf/teamcity-agent-log4j.xml](#) files to include the following fragment:

```

<category name="jetbrains.buildServer.VCS" additivity="false">
    <appender-ref ref="ROLL.VCS" />
    <appender-ref ref="CONSOLE-ERROR" />
    <priority value="DEBUG" />
</category>

<category name="jetbrains.buildServer.buildTriggers.vcs" additivity="false">
    <appender-ref ref="ROLL.VCS" />
    <appender-ref ref="CONSOLE-ERROR" />
    <priority value="DEBUG" />
</category>

```

Please also update the `<appender name="ROLL.VCS" />` node to increase the number of the files to store:

```
<param name="maxBackupIndex" value="30" />
```

If there are separate logging options for specific version controls, they are described below.

Subversion debug logging

 To enable SVN logging on the server side, switch the logging preset to "debug-SVN" in the administration web UI and then retrieve the `logs/teamcity-vcs.log` and `logs/teamcity-svn.log` files.

An alternative manual approach is also necessary for agent-side logging.

First, please enable the generic VCS debug logging, as described [above](#).

Uncomment the SVN-related parts (the `SVN.LOG` appender and `javasvn.output` category) of the Log4j configuration file on the server and on the agent (if the [agent-side checkout](#) is used). The log will be saved to the `logs/teamcity-svn.log` file. Generic VCS log should be also taken from `logs/teamcity-vcs.log`

ClearCase

Uncomment the Clearcase-related lines in the [`<TeamCity Home>/conf/teamcity-server-log4j.xml`](#) file. The log will be saved to `logs/teamcity-clearcase.log` directory.

Patch Application Problems

In case the [server-side checkout](#) is used, the "patch" that is passed from the server to the agent can be retrieved by:

- add property `system.agent.save.patch=true` to the build configuration.
- trigger the build.

the build log and the agent log will contain the line "Patch is saved to file \${file.name}"
Get the file and provide it with the problem description.

[Back to top](#)

Logging for .NET Runners

To investigate process launch issues for [.Net-related runners](#), enable debugging as described below. The detailed information will then be printed into the build log. It is recommended not to have the debug logging for a long time and revert the settings after investigation.

Add the `teamcity.agent.dotnet.debug=true` configuration parameter in the build configuration or on the agent and run the build.

▼ Alternative way to enable the logging

1. Open the `<agent home>/plugins/dotnetPlugin/bin` folder.
2. Make a backup copy of `teamcity-log4net.xml`
3. Replace `teamcity-log4net.xml` with the content of `teamcity-log4net-debug.xml`



 After a debug log is created, it is recommended to roll back the change.
The change in the `teamcity-log4net.xml` will be removed on the build agent autoupgrade.

[Back to top](#)

Remote Run Problems

The changes that are sent from the IDE to the server on a **remote run** can be retrieved from the server `.BuildServer/system/changes` directory. Locate the `<change_number>.changes` file that corresponds to your change (you can pick the latest number available or deduce the URL of the change from the web UI).

The file contains the patch in the binary form. Please provide it with the problem description.

[Back to top](#)

Logging in IntelliJ IDEA/Platform-based IDEs

To enable debug logging for the **IntelliJ Platform-based IDE plugin**, include the following fragment into the Log4j configuration of the `<IDE home>/bin/log.xml` file:

```
<appender name="TC-FILE" class="org.apache.log4j.RollingFileAppender">
    <param name="MaxFileSize" value="10Mb"/>
    <param name="MaxBackupIndex" value="10"/>
    <param name="file" value="$LOG_DIR$/idea-teamcity.log"/>
    <layout class="org.apache.log4j.PatternLayout">
        <param name="ConversionPattern" value="%d [%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`

the logging related to the **open in IDE** functionality will appear in the IDE *console*.

No Suitable Build Configurations Found for Remote Run

First of all, check that your **VCS settings** in IDEA correspond to the **VCS settings** in TeamCity.

If they do not, change them and it should fix the problem.

Secondly, check that the build configurations you expect to be suitable with your IDEA project has either [server-side VCS checkout mode](#) or [agent-side checkout](#) and NOT manual VCS checkout mode (it is not possible to apply a personal patch for a build with the manual checkout mode because TeamCity must apply that patch after the VCS checkout is done, but it does not know or manage the time when it is performed).

If the settings are the same and you do not use the manual checkout mode but the problem is there, do the following:

- Provide us your IDEA VCS settings and TeamCity VCS settings (for the build configurations you expect to be suitable with your IDEA project)
- Enable debug logs for the TeamCity IntelliJ plugin (see [above](#))
- Enable the TeamCity server debug logs (see [above](#))
- In the [TeamCity IntelliJ plugin](#), try to start a remote run build
- Provide us debug logs from the TeamCity IntelliJ plugin and from the TeamCity server.

[Back to top](#)

Logging in TeamCity Eclipse plugin

To enable tracing for [the plugin](#), run Eclipse IDE with the `-debug <filename>` command line parameter. The `<filename>` 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/performance/vcs = true
jetbrains.teamcity.core/performace/teamcity = true
```

Read more about Eclipse Debug mode [Gathering Information About Your Plug-in](#) and built-in Eclipse help.

[Back to top](#)

TeamCity Visual Studio Addin issues

TeamCity Addin logging

To capture logs from the TeamCity Visual Studio Addin, please run Microsoft Visual Studio executable (devenv.exe) with additional command line arguments:

- For TeamCity VS Add-in as a part of [ReSharper Ultimate](#) use `/ReSharper.LogFile <PATH_TO_FILE>` and `/ReSharper.LogLevel <Normal|Verbose|Trace>` switches
- For Legacy version of TeamCity VS Add-in, use `/TeamCity.LogFile <PATH_TO_FILE>` and `/TeamCity.LogLevel <Normal|Verbose|Trace>` switches

Visual Studio logging

To troubleshoot common Visual Studio problems please run Visual Studio executable file with [/Log](#) command Line switch and send us resulting log file.

[Back to top](#)

dotCover Issues

To collect additional logs generated by [JetBrains dotCover](#), add the `teamcity.agent.dotCover.log` configuration parameter to the build configuration with a path to an empty directory on the agent.

All dotCover log files will be placed there and TeamCity will publish zipped logs as hidden build artifact `.teamcity/.NETCoverage/dotCoverLogs.zip`.

JVM Crashes

On a rare occasion of the TeamCity server or agent process terminating unexpectedly with no apparent reason, it can happen that this is caused by a Java runtime crash.

If this happens, the Oracle JVM creates a file named `hs_err_pid*.log` in the working directory of the process. The working directory 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 Oracle documentation, [the Fatal Error Log section](#).

Please send this file to us for investigation and consider updating the JVM for [the server](#) (or for agents) to the latest version available.

If you get the "There is insufficient memory for the Java Runtime Environment to continue. Native memory allocation (malloc) failed to allocate ..." message with the crash or in the crash report file, make sure to [switch to 64 bits JVM](#) or reduce -Xmx setting not to increase 1024m, see details in the [memory configuration section](#).

Build Log Issues

While investigating issues related to a build log, we might need the raw binary build log as stored by TeamCity.

It can be downloaded via the Web UI from the Build Log build's tab: select "Verbose" log detail and use the "raw messages file" link at the top-right.

Uploading Large Data Archives

Files under 10 MB in size can be attached right into the [tracker issue](#) (if you do not want the attachments to be publicly accessible, limit the attachment visibility to "teamcity-developers" user group only).

You can also send small files (up to 2 MB) via email: teamcity-support@jetbrains.com or via online form (up to 20 MB). Please do not forget to mention your TeamCity version and environment and archive the files before attaching.

FTP

If the file is over 10 MB, you can upload the archived files to <ftp://ftp.intellij.net/uploads> and let us know the exact file name. If you receive the permission denied error on an upload attempt, please rename the file. It's OK that you do not see the file listing on the FTP.

The FTP accepts standard anonymous credentials: username: "anonymous", password: "<your e-mail>".

In addition to usual, unencrypted connections, TLS ones are also supported.

In case of access issues, time-out errors, etc. please try using passive FTP mode.

HTTP

You can upload a file via <https://uploads.jetbrains.com/> form and let us know the exact file name.

[Back to top](#)

Applying Patches

Microsoft Visual Source Safe Integration

To apply 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

- r
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.
2. 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 tfs-native.exe:

1. Shutdown TeamCity server
2. Open <TeamCity Server>/webapps/root/WEB-INF/plugins/tfs/ or <TeamCity Server>/webapps/root/WEB-INF/lib/
3. Backup tfs-support.jar
4. Inside the tfs-support.jar file, replace /bin/tfs-native.exe with the new one
5. Start the server

To apply full TFS plugin patch:

1. Shutdown TeamCity server
2. Open <TeamCity Home>/webapps/root/WEB-INF/plugins/tfs/ or <TeamCity Home>/webapps/root/WEB-INF/lib/
3. Back up tfs-support.jar
4. Replace tfs-support.jar with the new one
5. Start the server

Capturing logs from TFS-native

To enable creating logs from TFS-native:

1. Locate tfs-native.exe under TeamCity temp folder. File name should look like TC-TFS-NATIVE-<digits>.exe
2. Create a copy of the file in any other folder.
3. 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:

1. Open <TeamCity Server>/webapps/root/update/plugins/
2. Copy dotNetPlugin.zip to temporary folder
3. Back up dotNetPlugin.zip
4. Extract dotNetPlugin.zip
5. Replace contents of /bin folder with new files.
6. Pack files again. Make sure there are no files in the root of the archive.
7. Replace dotNetPlugin.zip file on the server. All build agents will upgrade automatically.
8. Run the builds.

To enable logging from .NET runners:

1. Open <TeamCity Server>/webapps/root/update/plugins/
2. Copy dotNetPlugin.zip to temporary folder
3. Back up dotNetPlugin.zip

4. Extract dotNetPlugin.zip
5. Copy /bin/teamcity-log4net-debug.xml to /bin/teamcity-log4net.xml
6. You may patch Log4NET config file if you need.
7. Pack files again. Make sure there is no files in the root of the plugin archive.
8. Replace the dotNetPlugin.zip file on the server.
9. All build agents should upgrade automatically.
10. 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.

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

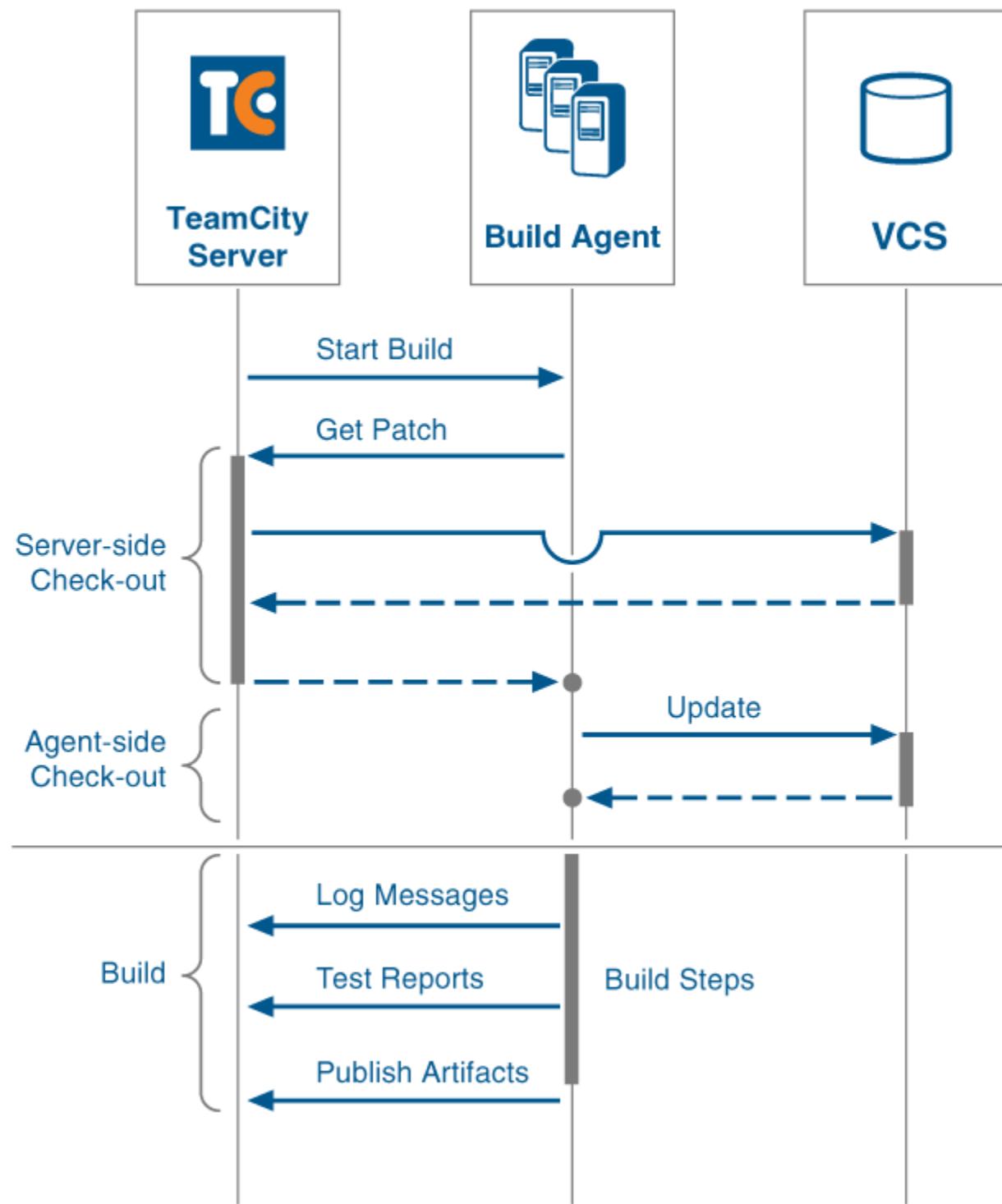
- [Installing and Configuring the TeamCity Server](#)
- [Build Agent](#)
- [Build Grid](#)
- [Setting up and Running Additional Build Agents](#)

Build Lifecycle in TeamCity

To demonstrate a build lifecycle in TeamCity, we need to introduce another important term – Version Control System:

i 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 needed for the build.

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.

The screenshot shows the TeamCity interface with a navigation bar at the top. The main content area is titled 'Getting started with TeamCity'. It features a large blue button labeled 'Create a project' with a right-pointing arrow icon. Below the button, there is a message: 'To start running builds, create projects and build configurations first.' Underneath this message, there is a section titled 'You may also want to:' followed by three bullet points: 'configure email and Jabber settings to enable notifications.', 'manage licenses, and', and 'add more users to TeamCity.'

There are no projects to show. Possible reasons:

- You should configure visible projects

- There are no projects/build configurations yet. Please, [create them](#).

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 built (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.



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.

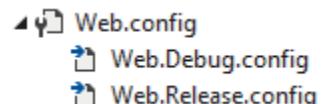
What if there are leftovers of unnecessary or old assemblies on that workstation we're xcopying from? What if we forget to change the database connection string in *Web.config* and mess up that release? How do we quickly roll back if that happens? The .NET stack has a solution to this: Configuration 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:

```
Original Web.config
13  </connectionStrings>
14  <appSettings>
15    <add key="webpages:Version" value="2.0.0.0" />
16    <add key="webpages:Enabled" value="true" />
17    <add key="ClientValidationEnabled" value="true" />
18    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
19  </appSettings>
20  <system.web>
21    <authentication mode="None" />
22    <compilation debug="true" targetFramework="4.0" />
23    <httpRuntime targetFramework="4.0" />
24  </system.web>
25  <system.webServer>
26    <modules>
27      <remove name="FormsAuthentication" />
28    </modules>
29  </system.webServer>
30</configuration>
```

```
Transformed Web.config (transforms applied: Web.Release)
13  </connectionStrings>
14  <appSettings>
15    <add key="webpages" value="2.0.0.0" />
16    <add key="webpages" value="false" />
17    <add key="ClientValidation" value="true" />
18    <add key="UnobtrusiveJavaScript" value="true" />
19  </appSettings>
20  <system.web>
21    <authentication mode="None" />
22    <compilation targetFramework="4.0" />
23    <httpRuntime targetFramework="4.0" />
24  </system.web>
25  <system.webServer>
26    <modules>
27      <remove name="FormsAuthentication" />
28    </modules>
29  </system.webServer>
30</configuration>
```

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.

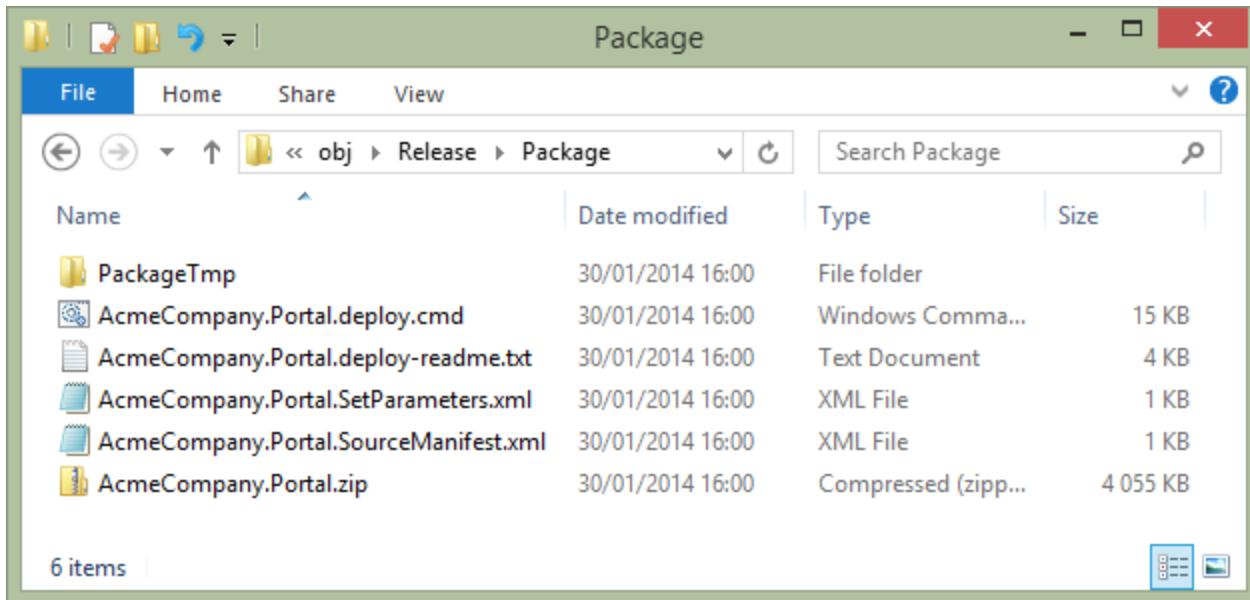
But before we deploy, let's first see how we can create a deployment package. And just so we learn about the package format, let's first do this manually by invoking msbuild.

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.

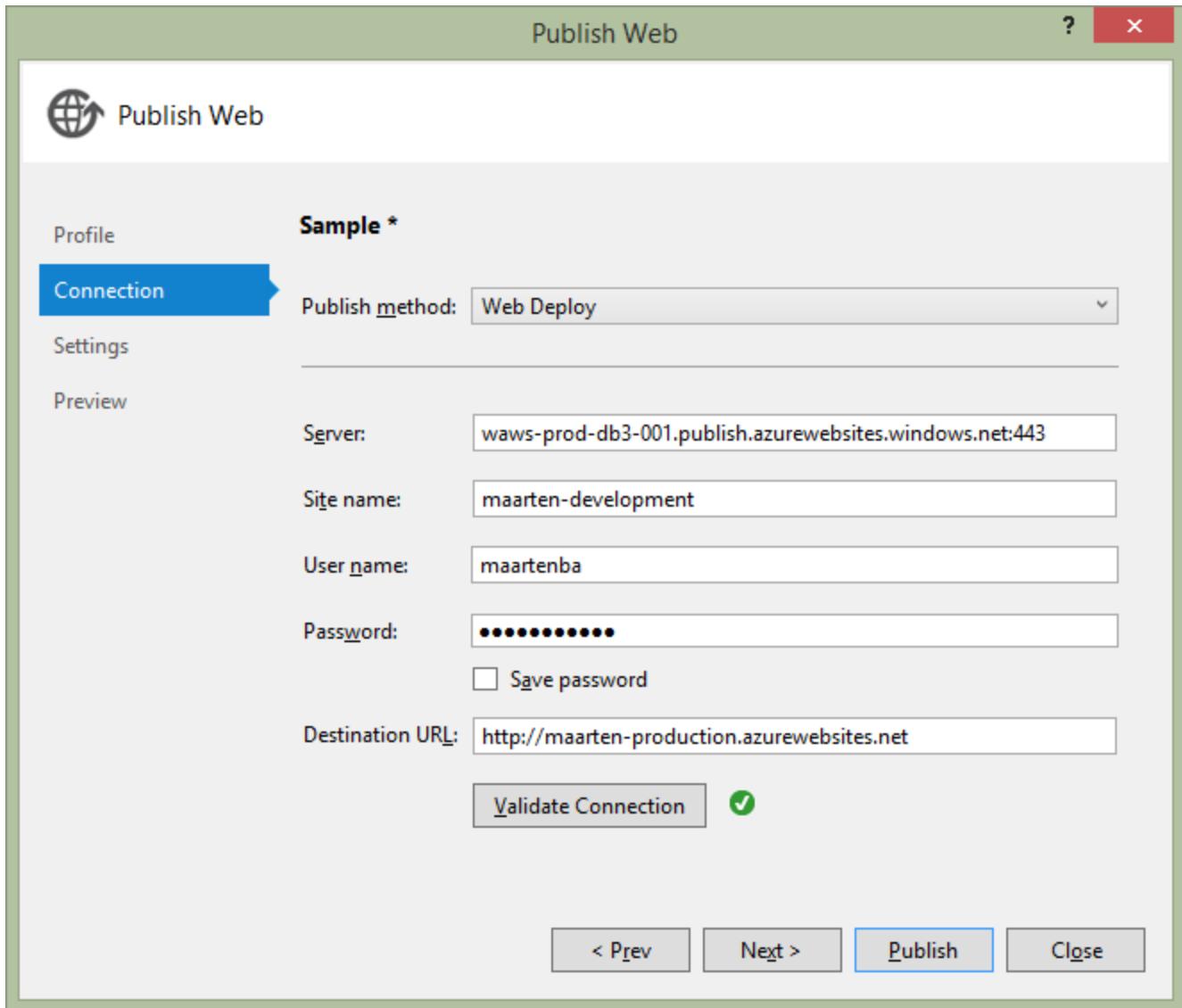
For the remainder of this tutorial, we will be covering deployment to Windows Azure Web Sites using WebDeploy, which is identical to how it works on IIS.

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.

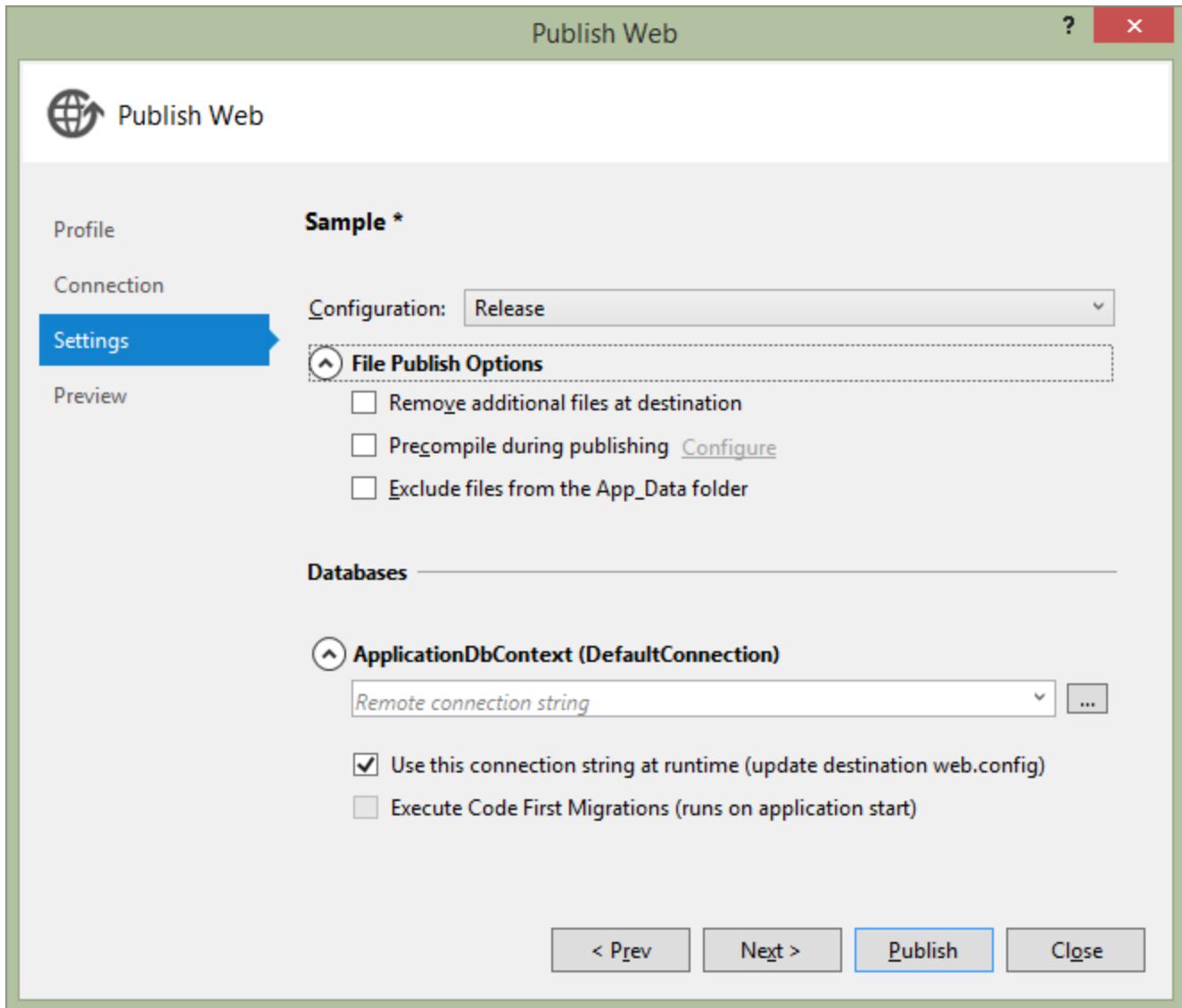
In the first step, we have to specify destination server details. This would typically be the HTTPS endpoint to the WebDeploy host (or FTP or file share details if that option was selected). After providing all details, we can validate the connection to see if it works.



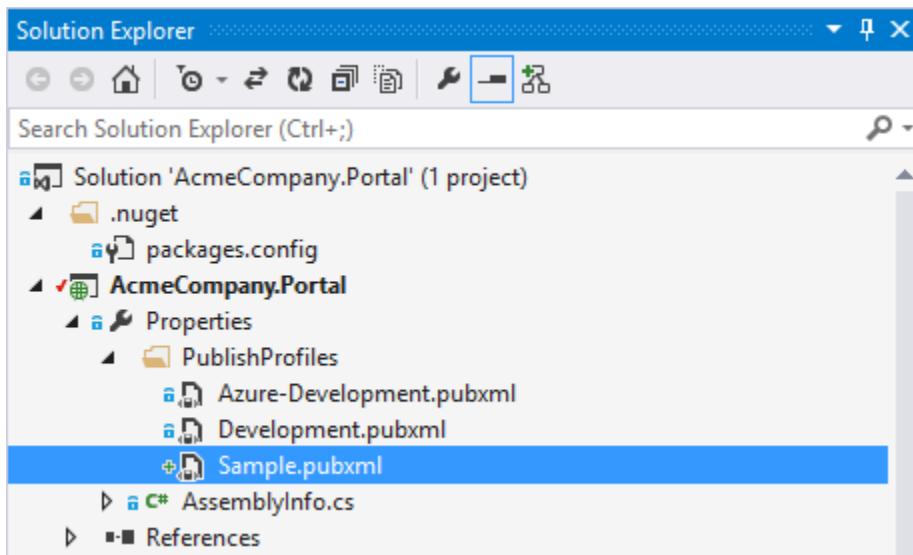
Note that instead of going through the entire wizard, Windows Azure Web Sites tooling allows importing the publish profile from the Windows Azure Management Portal. I'm showing the entire process here for when deploying to IIS.

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 `<Deskt
opBuildPackageLocation>` 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:DeployOnBuild=True  
/p:PublishProfile="Development"  
/p:ProfileTransformWebConfigEnabled=False
```

Enter additional command line parameters to MSBuild.exe.

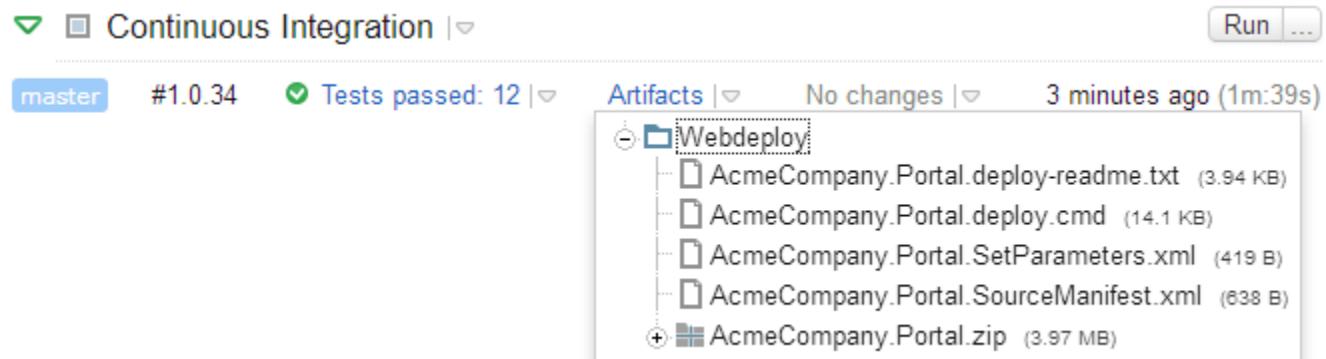
 Hide advanced options

Here's what these parameters do:

- `/p:DeployOnBuild=True` - triggers WebDeploy packaging
- `/p:PublishProfile="Development"` - specifies the deployment profile to use when packaging
- `/p:ProfileTransformWebConfigEnabled=False` - let's discuss this one in detail!

 Standard configuration transformations are run in an early stage, but WebDeploy runs another transformation using the `<LastUsedBuildConfiguration>` setting from our publish profile. This causes earlier configuration transformations to be overwritten, which we don't want to happen. Disabling the `ProfileTransformWebConfigEnabled` parameter avoids running this additional configuration transformation.

If we now run the build again (having specified `artifacts\webdeploy\Development => Webdeploy` as the artifact path, which is the path we configured in the publish profile earlier on), we will see a familiar set of files published as artifacts:



The screenshot shows the TeamCity interface for a build named "Continuous Integration". The build number is "#1.0.34" and it has "Tests passed: 12". The "Artifacts" section shows a folder named "Webdeploy" containing several files: "AcmeCompany.Portal.deploy-readme.txt" (3.94 KB), "AcmeCompany.Portal.deploy.cmd" (14.1 KB), "AcmeCompany.Portal.SetParameters.xml" (419 B), "AcmeCompany.Portal.SourceManifest.xml" (638 B), and a file "AcmeCompany.Portal.zip" (3.97 MB) which is expanded.

Now let's see if we can set up the actual deployment as well!

Step 3: Setting up the deployment on TeamCity

The strategy we'll be using for our deployments is described in the [How To....](#). We will be creating a new build configuration for every target environment we want to deploy to. These new build configurations will:

- Run the build
- Perform the deployment

What we want to achieve is this nice waterfall, where we can promote our build from CI to development to staging to production, or whichever environments we have in between CI and production.



From the TeamCity Administration, copy the CI build configuration and name it differently, for example "Deploy to Windows Azure Web Sites - Development". Next, we will make some changes to the build configuration.

Let's start by specifying build dependencies. Under the build configuration's *Dependencies*, add a new snapshot dependency on our CI build. This will ensure that deployment will only be possible if a matching CI build has passed completely, and that the deployment will be based on the exact same VCS revision as we built during CI.

Snapshot Dependencies

Build configurations linked by a snapshot dependency will use the same snapshot of the sources. The build dependencies are built. If necessary, the dependencies will be triggered automatically. ?

[+ Add new snapshot dependency](#)

<input type="checkbox"/>	<input checked="" type="checkbox"/> Edit Snapshot Dependency	X
<input type="checkbox"/>	Depend on: AcmeCorp.Portal :: Continuous Integration	▼
Artifact	Options:	<input checked="" type="checkbox"/> Do not run new build if there is a suitable one <input checked="" type="checkbox"/> Only use successful builds from suitable ones <input type="checkbox"/> Run build even if dependency has failed <input type="checkbox"/> Run build on the same agent
Artifact		Save Cancel

We want to be able to identify the build numbers throughout the entire chain of deployments. For example, if CI build 1.0.0 is deployed to staging, we want to be sure that this is actually version 1.0.0 and not some intermediate version. Under *General Settings*, change the build number format to use the same version number as the originating CI build. The build number format will have to be similar to `%dep.WebAcmeCorpPortal_ContinuousIntegration.build.number%`, duplicating the version number from the CI build.

Our CI build was building the default configuration for our solution. Since we are now deploying to a different environment and we've created deployment configurations (and configuration transforms) for Development and Production, let's change the build configuration through the Visual Studio build step.

Configuration: Development

Enter solution configuration to build. Debug or Release are supported in default solution file. Leave blank to use default

Now comes the actual deployment step! Up until now, we have built our project but we haven't really done anything to ship it to an actual server. Let's change that by adding a new build step based on a Command Line runner. As the build script, enter the following:

```
"C:\Program Files\IIS\Microsoft Web Deploy V3\msdeploy.exe"
    -source:package='artifacts\WebDeploy\<target environment>\AcmeCompany.Portal.zip'
    -dest:auto,
        computerName="https://<windows azure web site web publish
URL>:443/msdeploy.axd?site=<windows azure web site name>",
        userName="<deployment user name>",
        password="<deployment password>",
        authType="Basic",
        includeAcls="False"
    -verb:sync
    -disableLink:AppPoolExtension -disableLink:ContentExtension
    -disableLink:CertificateExtension
    -setParamFile:"msdeploy\parameters\<target
environment>\AcmeCompany.Portal.SetParameters.xml"
```

That's quite a bit, right? Let's go through this command:

- "C:\Program Files\IIS\Microsoft Web Deploy V3\msdeploy.exe" is the path to the msdeploy.exe which has to be available on the build agent.
- `-source:package='artifacts\WebDeploy\<target environment>\AcmeCompany.Portal.zip'` specifies the deployment package we want to upload
- `-dest:auto,computerName="https://<windows azure web site web publishURL>:443/msdeploy.axd?site=<windows azure web site name>,userName="<deployment user name>",password="<deployment password>,authType="Basic",includeAcls="False"` specifies the URL to the deployment service. For Windows Azure Web Sites, this will be in the aforementioned format. For IIS, this may be different (see Sayed Ibrahim Ashimi's excellent post on [WebDeploy parameters](#))
- `-verb:sync` tells WebDeploy to synchronize only changed files (this will drastically reduce deployment time as not all files will be uploaded for every deployment)
- `-disableLink:AppPoolExtension -disableLink:ContentExtension -disableLink:CertificateExtension` are used to disable certain configuration steps on the remote machine. These may be different for your environment, see [MSDN for a complete list](#).
- `-setParamFile:"msdeploy\parameters\<target environment>\AcmeCompany.Portal.SetParameters.xml"` is an important one. It specifies the WebDeploy parameters that will be replaced in the deployed Web.config file on the remote server, for example the connection string. More on this 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 connections strings a secret to everyone but TeamCity.

VCS Roots

In this section you can configure how project source code is retrieved from VCS. [?](#)

[+ Attach VCS root](#)

Name

(jetbrains.git) AcmeCorp.Portal belongs to AcmeCorp.Portal

(jetbrains.git) AcmeCorp.Portal (msdeploy) belongs to AcmeCorp.Portal

We can repeat these steps to create a build configuration for staging, for QA, for production and so on. Since we want to promote builds over this entire chain, these configurations should all have a snapshot dependency on the previous environment.

Here's what this could look like: 3 different build configurations, denoting different versions that are deployed to each target environment:

AcmeCorp.Portal | AcmeCorp Portal no hidden | [x](#)

Continuous Integration | [Run ...](#) [x](#)

master #1.1.3 Tests passed: 12 | Artifacts | No changes | one minute ago (2m:55s)

Deploy to Windows Azure Web Sites - Development | [Run ...](#) [x](#)

<default> #1.1.0 Tests passed: 12 | Artifacts | No changes | moments ago (3m:56s)

Deploy to Windows Azure Web Sites - Production | [Run ...](#) [x](#)

<default> #1.0.33 Tests passed: 12 | Artifacts | No changes | 4 hours ago (5m:22s)

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.

Run ... Actions [Edit Configuration Settings](#)

Comment... [Build](#)

Pin... [Build](#)

Tag... [Build](#)

Promote... [Build](#)

Mark as failed...

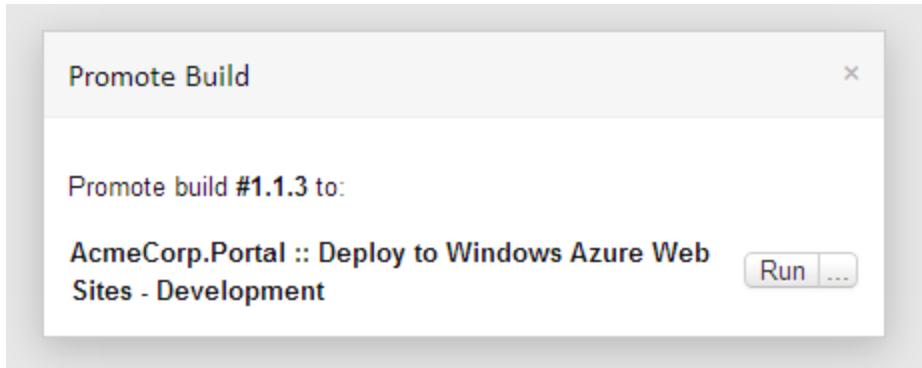
Label this build sources...

Merge this build sources...

Remove...

100% 18/18

Having configured the snapshot dependencies for our build configurations, TeamCity knows what the next environment should be: development.



This will trigger a new build that will deploy version 1.1.3 to the development environment. Once validated, we can navigate to that build's results and promote the build to the next environment.

Because of the snapshot dependencies we created, we can now also go to any build's *Dependencies* tab and see the environments where it has been deployed to. Here's build 1.1.3 as seen from development. We can see a CI build has been made, deployment to development has been done and deployment to production is still running:

Overview Changes Tests Build Log Parameters Dependencies Artifacts

Snapshot dependencies

This build is part of 1 build chain. [?](#)

Page 1 of 1 (1 build chain [?](#))

[↑](#) [↓](#)

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Production

AcmeCorp.Portal :: Continuous Integration | [▼](#) [▶](#)
master ✓ #1.1.3 Tests passed: 12 | [▼](#)

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Development | [▼](#) [▶](#)
<default> ✓ #1.1.3 Tests passed: 12 | [▼](#)

AcmeCorp.Portal :: Deploy to Windows Azure Web Sites - Production | [▼](#) [▶](#)
<default> ✨ #1.1.3 Tests passed: 12 | [▼](#)

- For a build configuration with snapshot dependencies, we can enable showing of changes from these dependencies using the **Show changes from snapshot dependencies** version control setting. This enables us to see exactly which changes are deployed. See [Build Dependencies Setup - Changes from Dependencies](#) for more information.

Conclusion

By thinking of a deployment as a chain of builds, doing deployments from TeamCity is not too hard. In this tutorial, we've used WebDeploy as an example means of transferring build artifacts to a target environment, but this could also have been another solution (like xcopy).

Using VCS labeling, it's also possible to label sources when a specific deployment happens. By pinning builds (optionally through the TeamCity API), we can make sure that build cleanup does not remove certain builds and artifacts.

Getting started with PHP

In this tutorial:

- [Introduction](#)

- Setting up the project
- Adding build steps
 - Unit tests
 - Running Phing
 - Code coverage
- Exploring build results

Introduction

TeamCity supports your Continuous Integration (CI) process in many technologies. In this tutorial, we'll configure a Continuous Integration (CI) process for a PHP project. We will be using the open-source PHP project [PHPExcel](#) as a sample project we want to provide CI for. This project features a large amount of code, PHPUnit tests and uses Phing to create build artifacts. Using TeamCity, we will automate the build process and make it ready for immediate feedback once the source code on GitHub changes.

This tutorial assumes you already have a PHP environment with PEAR, PHPUnit and Phing installed. If not, now is the time. You can find more info on configuring your PHP environment [through this blog post](#).

Setting up the project

We'll start by creating a new project and build configuration in TeamCity. The build number for this new build configuration will be "1.7.6.{0}" since PHPExcel is currently in the 1.7.6.x version range.

PHPExcel comes with a build script that creates build artifacts under the *build/release/** folder, which means we can already add that path as the artifact path TeamCity monitors.

http://localhost:8080/admin/createBuildConfiguration

Log in to TeamCity -- Tea...

TC Projects | My Changes | Agents 1 | Build Queue 0

Administration > PHPExcel - CI Project > Create Build Configuration

General Settings

Name: *

Description:

Build number format: Format may include '{0}' as a placeholder for build counter value, for example 1.{0}. It may reference to any available parameter, for example, VCS revision number: %build.vcs.number%. Note: maximum length of a build number after all substitutions is 256 characters.

Build counter: * [Reset counter](#)

Artifact paths: [Edit artifact paths:](#)

```
build/release/*%system.build.number%*  
unitTests/codeCoverage => coverage.zip
```

New line or comma separated paths to build artifacts. Ant-style wildcards like dir/**/*, directories like *.zip => winFiles, unix/distro.tgz => linuxFiles, where linuxFiles are target directories are supported.

Build options:

enable hanging builds detection
 enable status widget

Limit the number of simultaneously running builds (0 — unlimited)

[VCS settings >](#)

The PHPEXcel project has a GitHub repository at <https://github.com/maartenba/PHPEXcel.git>, a URL which we can configure in the Version Control System (VCS) settings.

← → TC http://localhost:8080/admin/editVcsf ⚙ ⚡ TC New VCS Root -- TeamCity ×

Log in to TeamCity -- Tea...

TC Projects My Changes Agents 1 Build Queue 0

Administration > Create Build Configuration > New VCS Root

Type of VCS

Type of VCS: Git

VCS Root Name

VCS Root Name: maartenba/PHPExcel - develop

Enter a unique name to distinguish this VCS root from other roots. If not specified, the name will be generated automatically.

General Settings

Fetch URL: * https://github.com/maartenba/PHPExcel.git

It is used for fetching data from repository.

Push URL:

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

Default Branch: * develop

Branch to be used if no branch from Branch Specification is set

Branch Specification:  Edit branch specification:
[branch specification input field]
Branches to monitor in addition to default one. Newline-delimited set of rules in the form of
+|-branch name (with optional * placeholder) ?

Adding build steps

A build configuration consists of several build steps which perform the actions we desire during build.

Unit tests

Since PHP is an interpreted language, we don't need a compilation step and can immediately start with unit tests: we want to make sure all tests are green every time source code is changed. Whenever there is a failing unit test, we want to fail the entire build and not provide any build artifacts. TeamCity comes with a number of predefined build steps for Java and .NET, but since we're on PHP we'll have to select the Command

Line build runner.

The screenshot shows the TeamCity interface for creating a build configuration. The top navigation bar includes links for 'Log in to TeamCity', 'Projects', 'My Changes', 'Agents', and 'Build Queue'. The current page is 'Administration > PHPExcel - CI Project > Create Build Configuration'. The main content area is titled 'New Build Step' and contains the following configuration details:

- Runner type:** Command Line (Simple command execution)
- Step name:** Run tests with coverage (You can specify a build step name to distinguish it from other steps.)
- Execute step:** Only if all previous steps were successful (You can specify step execution policy)
- Working directory:** (Optional, set if differs from the checkout directory)
- Run:** Custom script
- Custom script:** `php c:\teamcity-php\PHPUnit-TC.php -c %teamcity.build.checkoutDir%\unitTests\phpunit.xml`

Below the script editor, a note states: "Platform-specific script which will be executed as a .cmd file on Windows or as a shell script in Unix-like environments". At the bottom right are buttons for "VCS settings" and "Save".

We want to run the PHPUnit configuration that's provided with PHPExcel source code. Invoking this can be done using the following command line script:

```
phpunit \-c phpunit.xml
```

By default, TeamCity will import the test results provided by PHPUnit. However we can also report real-time test results to TeamCity, so that we

can already see results during a build run before it's even finished. Using a wrapper around PHPUnit which uses service messages to report build results. Locate the wrapper somewhere on the build agent or have the build agent download it from the above GitHub repository directly using a second VCS root.

The screenshot shows the TeamCity interface for the PHPExcel - CI project. The main navigation bar includes 'Projects' (selected), 'My Changes', 'Agents 1', 'Build Queue 0', 'Overview', 'Changes 0' (selected), 'Tests', 'Build Log', 'Build Parameters', and 'Artifacts'. The build configuration is 'Main' with build #1.7.6.2 (07 Jan 13 08:59). The build status is 'Stop' (red). The 'Tests' tab is selected, showing 66 failed tests (66 new). The 'Group by: package/suite' dropdown is set to 'package/suite'. The test results are listed under 'All tests':

- PHPExcel Unit Test Suite: DateTimeTest: DateTimeTest::testDATEDIF (9)
 - testDATEDIF with data set #2
 - testDATEDIF with data set #52
 - testDATEDIF with data set #89
 - testDATEDIF with data set #90
 - testDATEDIF with data set #91
 - testDATEDIF with data set #92
 - testDATEDIF with data set #93
 - testDATEDIF with data set #94
 - testDATEDIF with data set #95
- PHPExcel Unit Test Suite: DateTimeTest: DateTimeTest::testDATEVALUE (7)
 - testDATEVALUE with data set #24
 - testDATEVALUE with data set #30

We can already invoke our build configuration and should be getting unit test results displayed. But we're not finished yet, we want to add some more build steps.

Running Phing

Our next build step will be invoking **Phing**, a PHP project build system or build tool based on Apache Ant. PHPExcel comes with a Phing build

script which we'll invoke after all unit tests have passed. Let's add a new Command Line build step which uses Phing's command-line tool and pass some parameters to it:

```
phing \-f build\build.xml \-DpackageVersion=%system.build.number%
\‐DreleaseDate=CIbuild \-DdocumentFormat=doc release-standard
```

PHPExcel has four build targets defined (*release-standard*, *release-documentation*, *release-pear* and *release-phar*), all providing different build artifacts. The *release-standard* target which we've now specified at the command line is the default build for PHPExcel which generates a ZIP file containing all source code and phpDocumentor output.

We are also passing Phing the current build number from TeamCity using Phing's -D command line switches. The build script can use these to create the correct file names.

If you haven't configured the artifact paths while creating our build project, it's best to do so now (see [Setting up the project](#)). We want to make sure that the ZIP file generated in this build script is available from TeamCity's web interface.

When we run another build, we'll now see that unit tests are being run and afterwards the Phing build script is being run. Once the entire build is completed, we can find the ZIP file generated by the Phing build script as a downloadable build artifact.

The screenshot shows the TeamCity interface for a build of the PHPExcel CI project. The top navigation bar includes links for 'Projects' (selected), 'My Changes', 'Agents 1', 'Build Queue 0', and 'Log in to TeamCity'. The main title is 'PHPExcel - CI :: Main > #1.7...'. Below the title, the build number '#1.7.6.4 (07 Jan 13 09:07)' is displayed with a warning icon. A horizontal menu bar offers tabs for 'Overview', 'Changes 0', 'Tests', 'Build Log', 'Build Parameters', 'Artifacts' (selected), and 'Code Coverage'. Under the 'Artifacts' tab, two files are listed: 'coverage.zip (2.41Mb)' and 'PHPExcel_1.7.6.4_doc.zip (5.79Mb)'. A note indicates a total size of '8.21Mb'. There is also a link to 'Show hidden artifacts'. At the bottom, there are 'Help' and 'Feedback' links, and the text 'TeamCity Professional 7.1.3 (build 24266)'.

Code coverage

The first build step we created was running unit tests using PHPUnit. The nice thing about PHPUnit is that it can provide code coverage information as well, nicely formatted as an HTML report. TeamCity can display HTML reports on a custom tab in the build results.

Let's first make sure code coverage is enabled. Edit the first build step (running PHPUnit) and make sure it uses PHPExcel's `phpunit-cc.xml` configuration file for configuring PHPUnit. This configuration file which is specific to PHPExcel outputs its code coverage report in the `unitTests/codeC` coverage folder. While it is possible to add all generated files to TeamCity as a build artifact, it's cleaner to ZIP that entire folder and make it available as one single file. We can have TeamCity create this ZIP file for us by using a special artifact path pattern! Edit the build artifacts again and make sure the following two artifact paths are specified:

```
build/release/*%system.build.number%*  
unitTests/codeCoverage => coverage.zip
```

We can let TeamCity create a ZIP archive from the *unitTests/codeCoverage* path by simply using => and specifying a target artifact name.

Run the build again. Once it completes, the *Artifacts* tab should contain the *coverage.zip* file we've just created. Next to that, there should now be an additional tab *Code Coverage* available which displays code coverage results. Since we're using a TeamCity convention for reporting code coverage, namely creating a *coverage.zip* build artifact, TeamCity will automatically display the coverage results in a new report tab.

← → TC http://localhost:8080/viewLog.html?l ↗ ⌂ TC PHPExcel - CI :: Main > #1.7... X

Log in to TeamCity -- Tea...

TC Projects My Changes Agents 1 Build Queue 0

PHPExcel - CI > Main > ! #1.7.6.4 (07 Jan 13 09:07)

Overview Changes 0 Tests Build Log Build Parameters Artifacts Code Coverage

C:\TeamCity\buildAgent\work\271472a647ed9c6c\Classes (Dashboard)

Class Coverage Distribution

Coverage Range (%)	Count
0%	~105
0-10%	~25
10-20%	~5
20-30%	~5
30-40%	~5
40-50%	~5
50-60%	~5
60-70%	~5
70-80%	~5
80-90%	~5
90-100%	~5
100%	~5

Class Cor

X	Y
1400	1400
750	750
550	550
100	100
50	50
10	10
5	5
2	2
1	1
0.5	0.5

Top Project Risks

- PHPExcel_Reader_Excel5 (1614170)
- PHPExcel_Calculation_Statistical (486506)

Least Tes

- PHPExcel_Worksh
- PHPExcel_Worksh

Help Feedback TeamCity Professional 7.1.3 (build 24266)

The screenshot shows the TeamCity web interface for a PHPExcel CI build. At the top, there are navigation icons (back, forward, search, refresh) and a URL bar showing <http://localhost:8080/viewLog.html?l>. The main header includes the TeamCity logo, the project name "PHPExcel - CI :: Main > #1.7...", and a build status indicator (#1.7...).

The main content area shows the build history for "Main" with build #1.7.6.4 (07 Jan 13 09:07). Below this, a navigation bar includes tabs for Overview, Changes (0), Tests, Build Log, Build Parameters, Artifacts, and Code Coverage. The "Changes" tab is selected.

The code editor displays the PHP source code for the `_isLeapYear` and `_dateDiff360` functions. The code is color-coded for syntax highlighting:

```
54     public static function _isLeapYear($year) {  
55         return (((($year % 4) == 0) && ((($year % 100) != 0) || ((($year  
56     } //     function _isLeapYear()  
57  
58  
59     private static function _dateDiff360($startDay, $startMonth, $sta  
60         if ($startDay == 31) {  
61             --$startDay;  
62         } elseif ($methodUS && ($startMonth == 2 && ($startDay == 29  
63             $startDay = 30;  
64         }  
65         if ($endDay == 31) {  
66             if ($methodUS && $startDay != 30) {  
67                 $endDay = 1;  
68                 if ($endMonth == 12) {
```

It's possible to add additional build reporting and display results from PHP mess detector, [PHPLint](#) or even have a tab available which displays phDocumentor 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, and a 'containing:' input field. A table lists test results with columns for 'Status' and 'Test'. The first two rows are 'Failure': 'testXIRR with data set #2' and 'testXIRR with data s'. The third row is 'OK': 'testToString'. The fourth row is 'OK': 'testCurrency with da'. A context menu is open over the first failure entry, with the 'Open in IDE' option highlighted by a red box. Other options in the menu include 'Test Details', 'Investigate / Mute...', 'toFilter', and 'Show in Build Log'.

Status	Test
Failure	testXIRR with data set #2 (PHPExcel Unit Test Su)
Failure	testXIRR with data s (PHPExcel Unit Test Su)
OK	testToString (PHPExcel Unit Test Su)
OK	testCurrency with da (PHPExcel Unit Test Su)
...	AdvancedValueBind (PHPExcel Unit Test Su)

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!