TeamCity Documentation

TeamCity 7.x

Welcome to Documentation space for TeamCity 7.x - distributed build management and continuous integration server. Here you will find description, instructions and notes about installing, configuring and using TeamCity, its features, options, and plugins.

> ℹ TeamCity 7.x contains a bunch of new features and improvements. To familiarize yourself with them, please refer to:
>
> - What's New in TeamCity 7.1
> - What's New in TeamCity 7.0
>
> Also, take a look on the list of Supported Platforms and Environments and on their visual representation.

Documentation Space Structure

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

Installation and Upgrade

Refer to the Installation section if you are installing TeamCity for the first time.
Review Upgrade Notes and see the Upgrade section if you are upgrading your existing TeamCity instance.

User's Guide

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

Administrator's Guide

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

Concepts

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

Misc

If you have a question on TeamCity it's worth checking out How To... section and pages under Troubleshooting.

Where to Find More Information and Send Feedback

To get general information about TeamCity and its features, please visit TeamCity Official Site.
On the Official TeamCity Blog and TeamCity Developers Blog you will find the latest news from our team, handy feature's descriptions, usage examples, tips and tricks.
You can also follow us on Twitter.

Feedback

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

Documentation for previous TeamCity versions

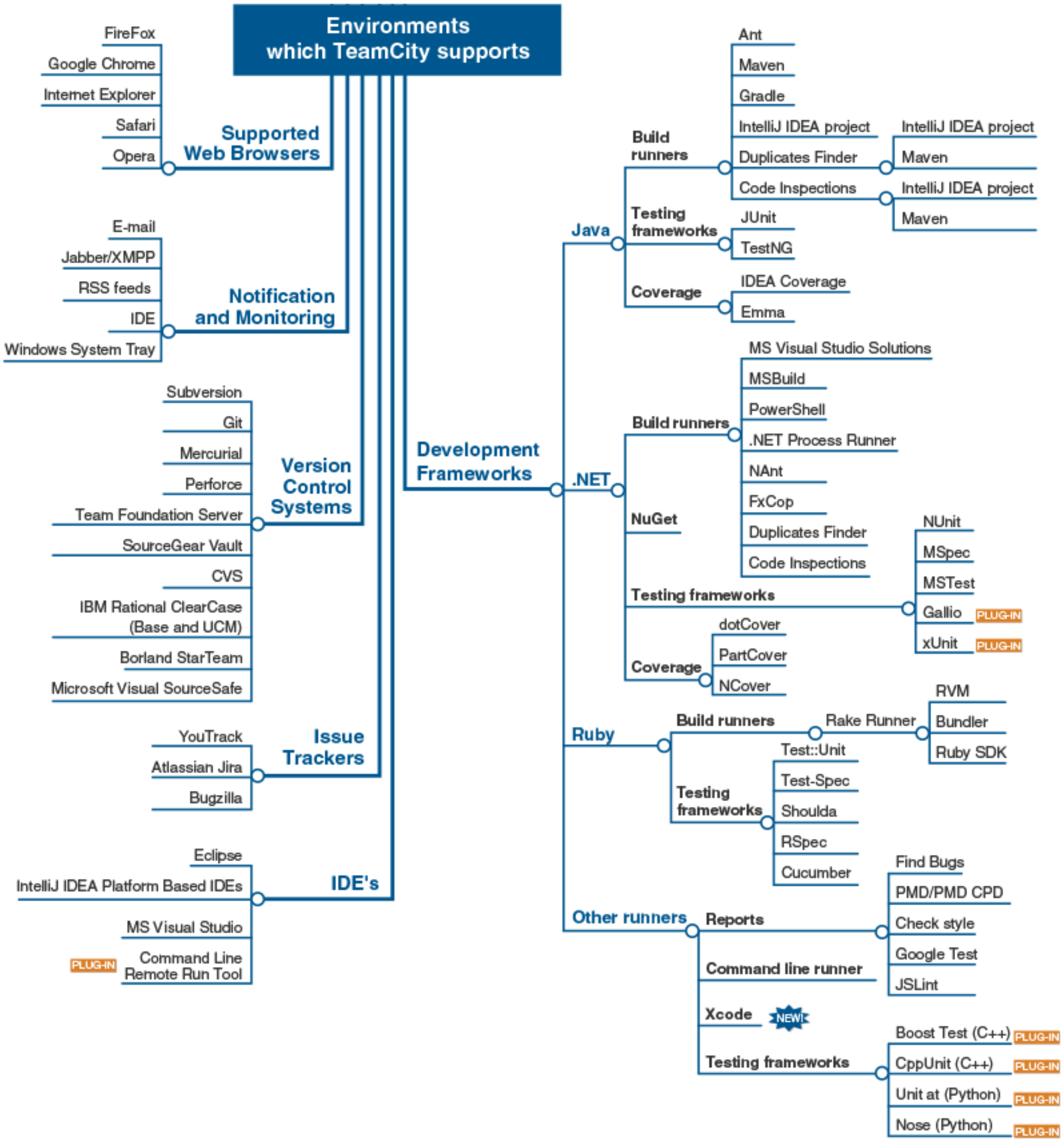
If you're using TeamCity earlier than 7.0, please refer to the corresponding documentation.

TeamCity Plugins

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

TeamCity Supported Platforms and Environments

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



See details at Supported Platforms and Environments.

What's New in TeamCity 7.1

- Feature branches for Git & Mercurial
  - Branch specification & default branch
  - VCS trigger
  - Branch label on a build
  - Branches filtering

Feature branches for Git & Mercurial

Distributed version control systems like Git & Mercurial provide a convenient way to develop in feature branches. Previous versions of TeamCity already supported development in feature branches to some extent.
Prior to TeamCity 7.1 you could:

- Create a copy of a build configuration for each feature branch with a new copy of a VCS root.
- Use Branch Remote Run Trigger.

Both ways have disadvantages. Creating a new build configuration for each branch is tedious and requires permissions, and it does not provide the same feeling as creating a branch in Git.
On the other hand Branch Remote Run Trigger can trigger personal builds only (by default, such builds are only visible to the build owner), so this approach does not work well if more than one developer works in a feature branch. Branch Remote Run Trigger also lacks some configuration parameters often needed in real-world scenarios, like an ability to wait some time before a build is triggered, or ignore some of the commits, etc.

These ways look like workarounds, so we felt a new approach was required to make TeamCity fully compatible with this development model.

Our main idea was to make development and building in branches as simple as possible. Ideally, all you have to do is to push your branch to a Git or Mercurial repository and TeamCity would detect it and start a build on your changes. And this is how it actually works in TeamCity 7.1! But to enable this you need to make some tiny changes in your build configuration.

Branch specification & default branch

Your repository may have a lot of branches and naturally not all of them are feature branches. In TeamCity you can tell the server which branches to watch for changes using branch specification. It has the following syntax:

```
+:<branch rule>
-:<branch rule>
```

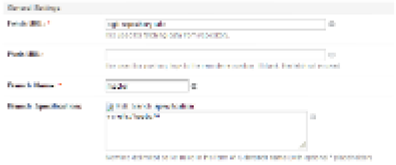
Where `<branch rule>` is a branch name in a VCS repository, `+:` and `-:` instruct TeamCity to include or exclude changes from branches with specified names (as you can see, the syntax is similar to checkout rules). There is no need to specify exact branch names, you can also use `*`. For example:

```
+:refs/heads/feature-*
```

Branch specification can be set right on the VCS root page, and as with almost any VCS root setting you can use parameter references here too.

Refer to our documentation to learn more about branch specification syntax.

Git & Mercurial VCS roots also have a dedicated field for branch name. This field specifies the default branch to be used when a branch is not defined. For example, if a build is added to the queue and no branch is provided, TeamCity will start a build in the default branch. There is a number of places where default branch is required, we'll cover them later.

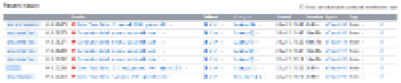


VCS trigger

Once you've configured branch specification in a VCS root, TeamCity starts monitoring branches for new changes. To enable automated build triggering on a commit to a VCS repository you need to add a VCS trigger to your build configuration.
Once a change is detected in a branch, TeamCity will trigger a build on this branch. All VCS trigger parameters like quiet period, per-checkin triggering and so on also work with branches. You can even configure TeamCity to trigger on a change in a branch in snapshot dependency!

Branch label on a build

When the VCS trigger triggers a build on a branch, a label with a branch name is assigned to the build:



Labels for builds from default branches are shown differently, see builds with `default` label on screenshot.

It is important to understand that branch label may not be the same as the actual branch name in the repository. For example, Git branch names typically look like this:

```
refs/heads/master
refs/heads/bugfix
refs/heads/release-1.0
```

It would be tedious to repeat `refs/heads/` in each build. Most likely, you would prefer to see `master`, `bugfix` or `release-1.0` instead. There are two solutions to this problem. First of all you can use `*` in branch specification. In this case, TeamCity will only take a part of branch name matched by `*` and use it as branch label in builds. So in our example, branch specification `+:refs/heads/*` would solve this problem. You can also use parentheses in branch specification to specify exactly which part of actual branch name should be used as branch label in build:

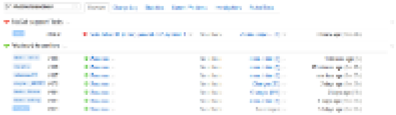
```
+:refs/heads/release-(1.0)
```

In this case, builds triggered on a `refs/heads/release-1.0` branch will be labeled `1.0`.

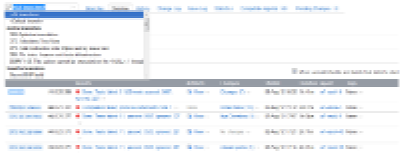
Branches filtering

Builds can be filtered by branch label in various places of the TeamCity web interface.

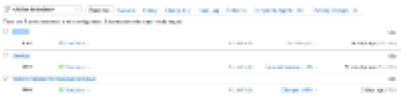
On the overview page you can filter all build configurations of a single project by selected branch name:



On the build configuration home page you can filter history, change log, issue log and pending changes by branch label:



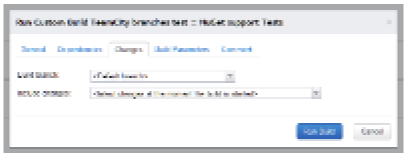
There is also a **Branches** tab in the build configuration which will give you a bird's-eye view on the status of active branches in a build configuration (i.e. branches with builds or changes within 1 week):



On this tab you can also easily trigger builds in a specific branch.

Run custom build dialog

If your build configuration has branches (on its own or via snapshot dependencies), custom build dialog will let you choose a branch to be used for a build:



Run Custom Build TeamCity branches test :: NuGet support Tests

Multiple VCS roots

If your build configuration uses multiple VCS roots and branches are set up in several of them, the way builds are triggered is more complicated.

VCS Trigger groups branches from several VCS roots by branch label (the part matched by `*` in branch specification). When some root doesn't have a branch from the other root, its default branch is used. Say, you have two VCS roots and both have a default branch `refs/heads/master`, first root has branch specification `refs/heads/7.1/*` and has changes in branches `refs/heads/7.1/feature1` and `refs/heads/7.1/feature2`, second root has specification `refs/heads/develop/*` and has changes in branch `refs/heads/develop/feature1`. In this case, VCS trigger runs 3 builds with revisions from the following combinations of branches:

root1	root2
refs/heads/master	refs/heads/master
refs/heads/7.1/feature1	refs/heads/devel/feature1
refs/heads/7.1/feature2	refs/heads/master

VCS branch parameter

For Git and Mercurial, TeamCity provides additional build parameters with names of VCS branches known at the moment of build starting. If a build took a revision from the `refs/heads/bugfix` branch, TeamCity will add a configuration parameter with the following name:
`teamcity.build.vcs.branch.<simplified VCS root name>=refs/heads/bugfix`

Where `<simplified VCS root name>` is the name of the VCS root where all non-alpha numeric characters are replaced with `_`.

In addition to this parameter, VCS branch is now shown on build changes page:

Other TeamCity features and their relationship to branches

A number of existing features in TeamCity were affected by branches support.

Build changes

For each build TeamCity shows the changes included in it. For builds from branches, changes calculation process takes the branch into account and presents you with changes relevant to the build branch.

First failed in / fixed in

TeamCity tries to detect new failing tests in a build - you can see in which build the test started to fail for those tests that are not newly created. Now this functionality is aware of branches too, i.e. when the first build is calculated, TeamCity traverses builds from the same branch.

Additionally, branch filter is available on the test details page, so you can see how the test passed / failed in a specific branch.

Dependencies

If a build configuration with branches has snapshot dependencies on other build configurations, all builds from the chain will be marked with this branch when the first build is triggered.

Artifact dependencies work mostly with builds from the default branch. Currently there is no way to have an artifact dependency to last finished build from a non-default branch. The same applies to finish build trigger - it will only watch for finished builds in the default branch.

Notifications

All notification rules except "My changes" will only notify about builds from the default branch. At the same time "My changes" rule will work for builds from all available branches.

VCS labeling

If you configured VCS labeling for your VCS roots, TeamCity will only label builds from the default branch.

REST

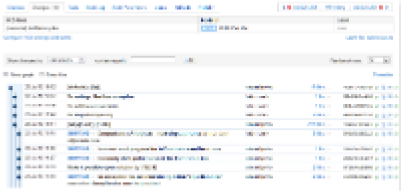
REST is fully aware of branches. The following requests are supported:
All available branches of a build configuration: GET `http://teamcity:8111/app/rest/buildTypes/<buildType_locator>/branches`
Build locator now supports the "branch" dimension. Use "<any>" for <branch_name> to return builds from all branches. If branch is not specified, only builds from the default branch are returned.
e.g. all builds for the specified branch: GET{{http://teamcity:8111/app/rest/builds/?locator=branch:<branch_name>}}
get build's branch: GET `http://teamcity:8111/app/rest/build/branch` - returns a text/plain response with the branch name (empty string for the default branch)

Branch Remote Run Trigger

Branch Remote Run Trigger works as usual, but with one important exception. If branch specification is provided in build configuration VCS roots, this trigger will not trigger builds on branches specified in branch specification. It will continue to trigger builds on all other branches matched by its own pattern.

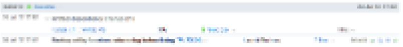
Change log and changes pages

Build Changes tab and Pending Changes tab have got a new look, very similar to Change Log. Apart from unified look and feel these pages also inherited some change log goodies, like graph of commits (especially useful for Git or Mercurial), ability to filter changes by user name and file path, and paging.

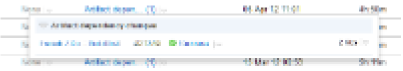


Artifact dependency changes

If TeamCity detects a change in the build artifact dependency (i.e. artifacts were downloaded from a different build compared to the previous build), this fact will be presented as a new change and will be shown on changes pages:

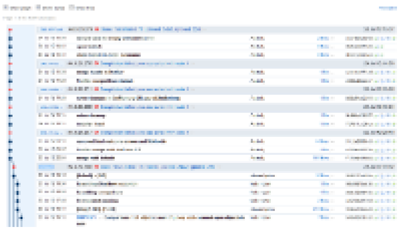


and in changes popup:



Builds on the change log graph

If the "Show builds" and "Show graph" options are enabled in the change log, TeamCity will display build markers on the graph. We find that this enhancement makes it much more clear which commit ends up in which build:

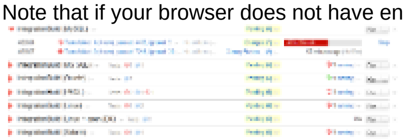


Current problems & investigations

Current problems are shown for collapsed configurations

In order to make the overview page more usable for those who have to monitor problems in many build configurations we started showing current problems for build configurations right on the overview page. To avoid clutter, problems are only shown if the build configuration is collapsed:

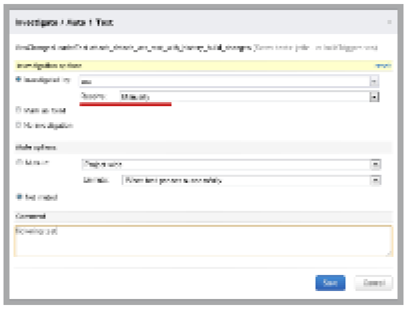
Note that if your browser does not have enough horizontal space TeamCity will tune the problems presentation accordingly:



Also note that similarly to queued builds, information about currently running builds can be seen in the popup too - there is no need to expand the build configuration.

Sticky investigation & my investigations highlighting

Usually if an investigation is assigned for a build configuration or a test, it is automatically removed once the build configuration becomes green or the test passes.
Sometimes it is not convenient though. For example, if a test fails from time to time (so-called flickering test), investigation will be removed - however, such test can't be considered fixed.
To allow better management of such problems we introduced the manual mode for investigation resolving. When you assign an investigation you can select how you want it to be resolved: automatically or manually:



The same way highlighting of user changes works across the TeamCity web interface, when an investigation is assigned to you and is shown somewhere in the web UI, such investigation will be highlighted:

Investigations highlighting is controlled by the same user profile setting that is used for changes' highlighting - the setting is now called "**Highlight my changes and investigations"**.

Version Control Repository browser

In several places in build step settings where a path to a build script can be specified you may now notice a small new icon: . By clicking on it you'll be presented with a VCS repository browser which allows you to choose a file in the repository.

This functionality is available for build configurations with Git, Mercurial, Subversion, Perforce, TFS and ClearCase VCS roots.

Support for NTLM HTTP authentication

The long-standing and popular feature request to add support for transparent/single-sign-on NTLM HTTP authentication has now been implemented: TW-6885.

The feature only works if the TeamCity server is installed on Windows and uses Windows domain authentication and only in the web browser.

Read more in our documentation

Manually mark build as successful or failed

Sometimes you need to mark a failed build as successful - for example, if a failure happened because of a minor problem, and you want other builds to take artifacts of this build, or you want to promote this build further through the pipeline.
On the other hand, there are cases when TeamCity is not able to determine build failure. For example, a build finished successfully but because of an error in the build script it did not produce any artifacts. So the build clearly cannot be treated as successful.

To handle both of these cases we added the ability to mark build as successful or failed. In both cases you need to provide some description of the reason of this operation. Also to be able to change build status one must have **Project administrator** role. These operations are logged into audit log and can be examined later.

Read more in our documentation.

New Windows service implementation for TeamCity server

Service which is used to start TeamCity on Windows was re-implemented in order to fix several annoying issues.

Starting from this release, all TeamCity server JVM options are stored in the exact same way as for command line execution mode - in environment variables.

Issues covered: TW-6450, TW-7467 (for server only), TW-12704, TW-14725, TW-15071, TW-15426, TW-17130, TW-18734, TW-20421.

Build steps execution

In previous TeamCity versions, if your build consisted of several steps, TeamCity made a decision whether to continue build steps execution basing only on an exit code returned by the previous step. If a previous build step returned a non-zero exit code, TeamCity would not execute the subsequent steps at all. As it turned out, this simple logic was too simple for many of our customers: TW-13682.

In TeamCity 7.1 we improved it and now you have a choice of the following options:
Execute step if:

- Only if all previous steps were successful (default choice, this is how previous versions of TeamCity behaved).
- Even if some previous steps are failed.
- Always, even if build stop command was issued.

The last policy can be useful for some cleanup tasks. Note that as in previous TeamCity version whether the step has failed or not is mostly determined by the process exit code.

Read more in our documentation.

Checkout on label in Perforce integration

In Perforce VCS root settings you can now specify the P4 label to use for sources checkout. Changes between the builds using labels are shown simply for informational purposes and are only accurate if the label was moved from one revision to another without labelling specific files.

If a build always checks out a certain fixed revision, a note is shown in the revisions table.

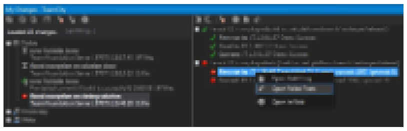
Related issues in the tracker: TW-11070 and TW-5061.

Read more in our documentation.

Integrations

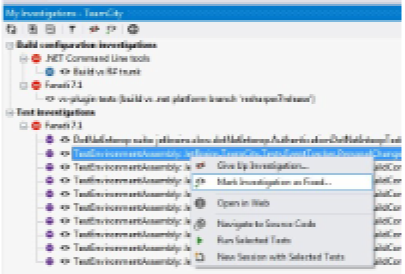
TeamCity VS-addin and Microsoft Visual Studio 2012

TeamCity VS-addin is now fully compatible with Visual Studio 2012, including dark theme support.



Other updates:

- "My Investigations" tool window added.



- VS-addin also supports TFS TeamExplorer 2012, ReSharper 7.0, dotCover 2.0.

NuGet support

TeamCity 7.1 fully supports NuGet 1.8 and 2.0.

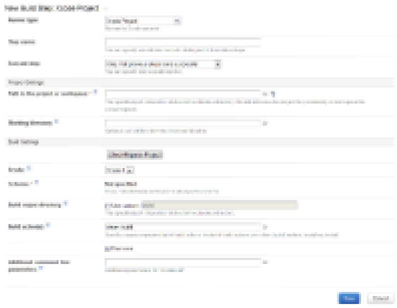
Other improvements are:

- NuGet Commandline tab now allows to upload a custom NuGet.CommandLine package on the server instead of downloading it from the public feed.
- NuGet Trigger supports Pre-release packages.
- NuGet Installer runner no longer requires you to have `packages/repository.config` file under solution. Now NuGet Installer runner uses Visual Studio solution file (.sln) to create the full list of NuGet packages to install. It also supports Solution-Wide packages from `.NuGet/packages.config` file. Packages upgrade can be done for entire solution or via packages.config files.

Xcode runner

Xcode runner has been made available as a separate plugin for both TeamCity 6.5 and 7.0. Starting with TeamCity 7.1 it is bundled.

Xcode runner needs to load information from Xcode project files, so once you specify path to the project or workspace, you need to click "Check / Reparse Project" button to load project settings.



A brief list of Xcode runner features:

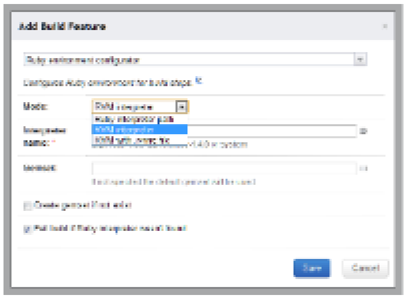
- Both Xcode 3 (target-based) and Xcode 4 (scheme-based) projects are supported.

- Structured build log based on Xcode build stages.
- Compilation errors are detected and reported to TeamCity.
- Tests started by `xcodebuild` are reported on the fly.
- Various Xcode-related tools installed on the agent are reported via agent parameters to simplify agent requirements configuration.

Read more in our documentation.

Ruby environment configurator

Ruby environment configuration now supports `.rvmrc` file. So if you have `.rvmrc` file under version control it is easy to instruct TeamCity to configure Ruby environment from this file.



Add Build Feature

Ruby environment configurator

Configures Ruby environment for build steps.

Mode: RVM interpreter

Ruby interpreter path
RVM interpreter
RVM with .rvmrc file

Interpreter name: *

Gemset:

If not specified the default gemset will be used

□ Create gemset if not exist

☑ Fail build if Ruby interpreter wasn't found

Read more in our documentation.

Amazon EC2

It is now possible to specify the Amazon Instance ID instead of Image ID in TeamCity cloud profile settings:

- If image ID is specified, it is run as a usual image and is **terminated** at the end.
- If instance ID of a not running EBS-based instance is specified, it is resumed when there is a need for an agent and **stopped** at the end.

A number of usability improvements has been done, for example, TeamCity now shows cloud agents with starting instances in agent compatibility lists.

Other integrations

- Eclipse plugin: Eclipse 4.2 is supported
- Databases: Microsoft SQL Server 2012 is supported
- Version control: TFS 2012 is supported
- NAnt 0.92 support added

Other improvements

- Several cosmetic changes applied all over the web interface
- Filter to show unused VCS roots added on global VCS roots page in Administration area
- Number of retry attempts can now be specified in Retry build trigger
- AssemblyFileVersion field added to AssemblyInfo patcher build feature
- Ability to navigate to corresponding place in the build log from the failed test popup
- Agent requirements imposed by parameter references now provide details where in configuration the reference is defined
- Server usage statistics show statistics about the browsers being used to access TeamCity
- Counters are shown for a number of configured build parameters, triggers, dependencies and requirements in the right sidebar on build configuration editing pages
- Server startup logic behind startup screens has been reworked to be more robust
- Windows user name and password are no longer exposed when you install the build agent service from Windows installer (TW-6425)
- CORS requests are supported in REST API
- You can embed TeamCity build status as a simple image into any HTML page via Build Status Icon
- Tests can now be easily muted and unmuted right from IntelliJ IDEA plugin
- Diff viewer is now able to highlight many more languages
- fixed issues

Previous Releases

What's New in TeamCity 7.0

What's New in TeamCity 7.0

Agent Pools

If you have tens of agents shared between several projects you probably suffer from the following issues:

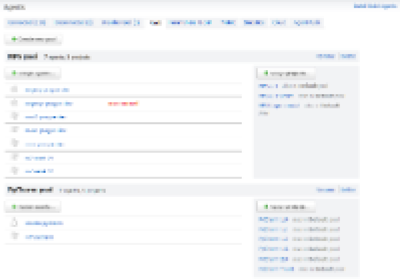
- there are no agents available for your project because all of them are used by builds from other projects;
- you cannot easily change software on agents because it can affect not only your builds but builds from other projects too;
- it is not easy to predict whether builds from your project will gain some benefits if more agents are added to the system.

To solve some of these issues you could assign agents and build configurations one to one, but this would be a tedious and absolutely not scalable approach due to low granularity level.

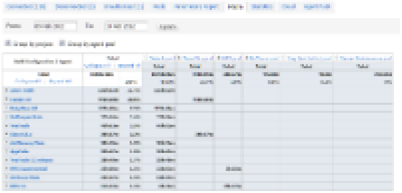
TeamCity 7.0 addresses these problems in a more convenient way. You can distribute your agents among several agent pools and assign each project to a pool. A project can be assigned to more than one agent pool, but agent can reside in a single pool only. If a project is assigned to a pool, its builds can run on agents of this pool only. They won't affect other pools and other projects. There is also a so-called **Default** pool, that cannot be removed and serves as a place for all agents and projects not assigned directly to other pools.

Ability to perform one-to-one assignment between agents and build configurations is still there and provides a fine grained control within a pool.

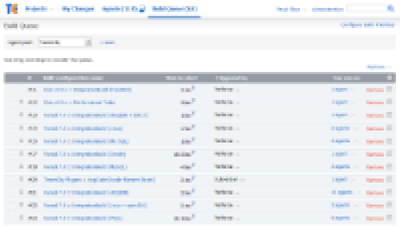
Agent pools can be configured on the "Pools" tab under "Agents":



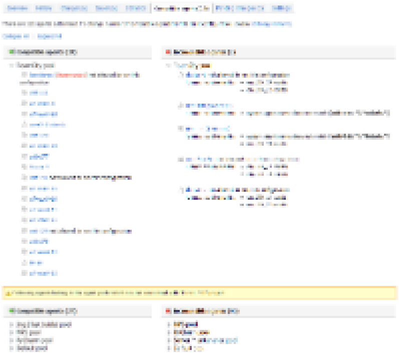
When you have agent pools configured you can easily monitor pools load on the agents matrix page:



If your project belongs to a single pool and there are no other projects in the pool, it is convenient to watch build queue for this pool only. This feature is also available:



Agent pools also affect compatible agents pages:



Learn more about agent pools

Typed build parameters

When defining a build parameter either on build configuration or project level, you can specify a type of a control for this parameter. For example, if a parameter has two values - true or false, it is natural to show it as checkbox. Currently this only affects custom build dialog.

Besides control type, you can also specify some standard properties for each parameter, such as:

- label - custom label to display for this parameter in custom build dialog
- description - text to be shown under the control in custom build dialog
- display type: **normal**, **hidden** or **prompt**. If **hidden** is specified, parameter will not be shown, but will be sent to a build; if **prompt** is specified, TeamCity will always require a review of parameter value, even when you trigger a build by simply clicking the "Run" button; if **normal** is selected, parameter will be shown as usual

Supported control types are:

- text field
- checkbox
- select
- password

If **password** parameter is used, TeamCity will do its best to hide the actual value of password parameter. You won't see the password in build logs, TeamCity web interface, or IDE plugins. It will be stored in configuration files in scrambled form. Note that as TeamCity performs global replacement of the password in build log, you have to choose relatively strong passwords - those that are unlikely to appear in the build log as a sequence of characters.

Parameter specification (a part of parameter which defines how to show it) cannot be overwritten if a parameter is inherited from a template or project.

Learn more about typed parameters

Build Chains

Build chains (a collection of builds combined by snapshot dependency) is a TeamCity approach to setting up something similar to a pipeline. They first appeared in TeamCity 4.0 (released in November 2008) and have seen many improvements since then.

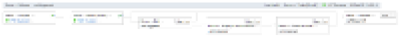
Some important notes:

- each build chain is a directed acyclic graph, i.e. build chain cannot have cycles
- all builds in a chain will have either the same revision, or, in case of several VCS roots, revisions corresponding to changes made at the same moment of time
- if a build depends on another one by a snapshot dependency it means the build cannot start until the dependency finishes
- several build chains can have common builds. For example, in order to reduce build chain execution time, TeamCity can decide to merge some build chains, if the results of a previous chain can be reused in a subsequent chain

Given the complexity of build chains, it is not easy to represent them visually. But we decided to make a first step in this direction in TeamCity 7.0. We added a new "Build Chains" tab on project and build configuration pages. The tab shows build chains containing builds from the project or build configuration correspondingly. The tab looks like this:

For each build chain you can see all builds that constitute a build chain as well as their status: not triggered, in queue, running or finished. Clicking a build in a chain will highlight this build and all its direct dependencies (both upstream and downstream). Since there can be several build chain in a single project, there is an ability to filter them.

But this page not just visualizes the chain, but also provides some abilities you can't easily reproduce using earlier versions of TeamCity. For example, the following chain is in **not triggered** state:

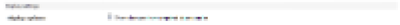


This means some of the builds were not started yet. By pressing the "Run" button you can continue the chain, i.e a new build will be started on the chain revisions and associated with builds from this chain.
An icon is shown for started builds only and opens the custom build dialog with build chain revisions preselected. This action can be used if you want to re-run some build in the chain.

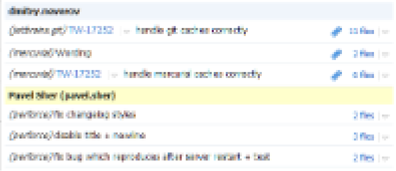
Besides build chains presentation we've made some other important improvements.

Changes from dependencies

For a build configuration with snapshot dependencies you can enable showing of changes from these dependencies transitively. The setting is available on the "Version Control Settings" step of the build configuration administration pages:



Enabling of this setting affects pending changes of build configuration, builds changes in builds history, change log and issue log. Changes from dependencies are marked with 🔗. For example:



dmitry.novakov

(jetbrains.git)/TW-17252 handle git caches correctly 10 files

(mercurial) Wording 2 files

(mercurial)/TW-17252 handle mercurial caches correctly 4 files

Pavel Sher (pavel.sher)

(perforce)/fix changelog styles 2 files

(perforce) disable title + newline 3 files

(perforce)/fix bug which reproduces after server restart + test 2 files

With this setting enabled, "Schedule Trigger" with a "Trigger build only if there are pending changes" option will consider changes from dependencies too.

Finish build trigger

Finish build trigger has become a little bit smarter too. If a build configuration is set up with a "Finish build trigger" and has snapshot dependencies to selected build configuration, the trigger will run the build on the same revisions and will attach the build to the chain. Thus you can have automated promotion of builds in a chain.

Build failure conditions

Newly added build configuration editing step called "Build Failure Conditions" accumulates all settings that define when the build should be marked as failed. Standard options like "fail build if at least one test failed" were moved from the "General Settings" step to this page.

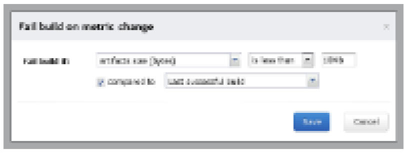
Besides the standard options, you can find two new build failure conditions:

- Fail build on metric change
- Fail build on specific text in build log

Fail build on metric change

With the help of this build failure condition you can track whether a specific statistical value changes within certain limits, and if it exceeds these limits, a build will be marked as failed. For example, you can require that the size of artifacts in your build is never less than 1Mb. Or that line coverage is always more than 80%.

There are cases when a build must be marked as failed if some statistical value differs significantly from a previous build (or from some reference build). For example, you may want to fail a build if the number of covered lines drops significantly compared to the previous build. Or, if compared to a build marked with some tag (reference build), the number of duplicate code fragments increased by more than a specified threshold. This is all possible with "fail build on metric change" condition.



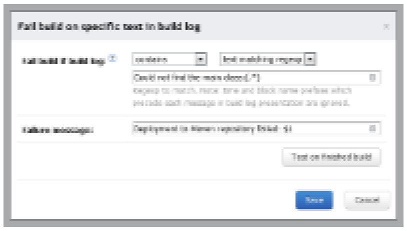
Fail build on metric change

Fail build if: artifacts size (bytes) is less than 10Mb

☑ compared to: Last successful build

Fail build on specific text in build log

The name of this failure condition speaks for itself. If text in the build log matches some regular expression, a build will be marked as failed. You can also define a failure message that will be shown in build status text and in the build log.



Fail build on specific text in build log

Fail build if build log: contains | text matching regexp

Could not find the main class(.*)

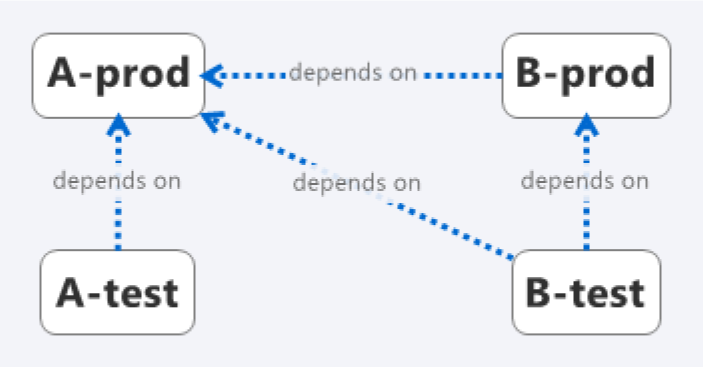
Regexp to match. Error, time and block name prefixes which precede each message in build log presentation are ignored.

Failure message: Deployment to Maven repository failed: $1

Learn more about build failure conditions

Incremental builds

Maven, Gradle and IntelliJ IDEA Project build runners now support incremental building (for IDEA it's limited to running tests). Say, your project has these modules: A-prod, A-test, B-prod, B-test.
Modules with -prod suffix contain production code, while modules with -test suffix contain tests for the corresponding production modules.



Now if a build starts with a change in module A-prod, TeamCity agent will run the tests in both modules (because B-test depends on A-prod and can be affected by the change).
However, if a change was made in B-prod only, TeamCity will only run the tests from B-test module.

In general, the more independent your modules are, the better. Modularity is a common good practice of software design, and now you can get another benefit from such approach: faster builds.

To enable this functionality for Gradle or IntelliJ IDEA project runners, simply turn on "Run affected tests only (dependency based)" checkbox. For Maven, you should enable "Build only modules affected by changes" checkbox.

Note that since IntelliJ IDEA project runner operates with run configurations instead of individual tests, when the "Run affected tests only (dependency based)" checkbox is enabled the runner will execute run configurations depending on affected modules only.

Read more about incremental building feature
Incremental building in Maven.

REST API

A long-standing feature request to allow build configuration editing via REST has finally been implemented.
Now you can get complete settings of a build configuration or template via REST and can also change them.
You can also create and delete build configurations, projects and VCS roots.

Here is an example of how to get build configuration steps:

```
curl -v --request GET http://user:pass@localhost:8111/app/rest/buildTypes/id:bt2/steps
```

Save individual step details to a file:

```
curl -v --request GET http://user:pass@localhost:8111/app/rest/buildTypes/id:bt2/steps/RUNNER_1 -o
response.xml
```

And create a new step (a copy of the saved one):

```
curl -v --request POST http://user:pass@localhost:8111/app/rest/buildTypes/id:bt2/steps --data-binary
@response.xml --header "Content-Type: application/xml"
```

REST API has had "/app/rest/application.wadl" since the very beginning, but it has now got an XML schema attached (see *<ns2:grammars><ns2:include href="application.wadl/xsd1.xsd">* element and try *application.wadl/xsd1.xsd* request) and will hopefully get some more inline comments soon.

If you are already using the REST API please note that the protocol version has changed.

Additionally with TeamCity 7.0 REST API you can:

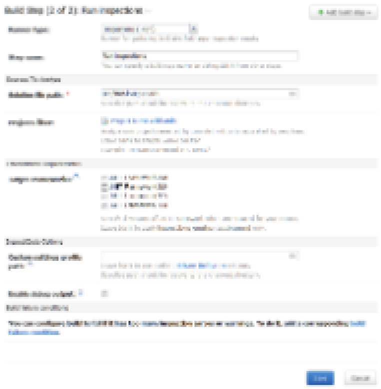
- change project level parameters
- authorize and disable agents and get parameters reported by an agent
- change user properties (e.g. VCS usernames)
- pause build configurations
- get content of a file from VCS using revisions of a build

Learn more about TeamCity REST API

Integrations

ReSharper Inspections

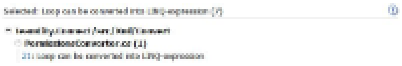
New Inspections runner for .NET is now available. Internally this runner uses parts of JetBrains ReSharper, so basically it means that ReSharper Code Analysis is now available on the server side too.
This runner helps you detect errors and problems in C#, XAML, XML, and ASP.NET code.



Key benefit of .NET Inspections runner is that it can use ReSharper 6.1 settings profiles. So you can easily tune the runner to match your own code style.

Inspection results are shown in the standard TeamCity interface.

Also, links to on-line inspections wiki are available for some inspection types.



Selected: Loop can be converted into LINQ-expression (?)

21: Loop can be converted into LINQ-expression

Which should help you learn how the code can be improved.

Loop can be converted into a LINQ expression

Known issues

We still have a few unresolved technical problems which could lead to incorrect inspection results. All these issues will be targeted in the future releases. Among them:

- in order to have adequate inspections execution results it might be necessary to **build your solution before running analysis**. This pre-step is especially relevant when you use (implicitly or explicitly) **code generation** in your project, see TW-19220.
- currently you could face with problems if you use MSBuild properties in your project files, see TW-19678.

Learn more about ReSharper inspections

NuGet

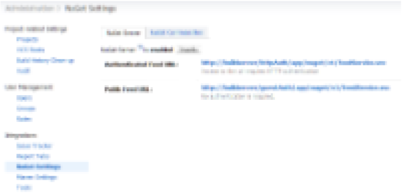
NuGet is a relatively new tool in .NET world. In brief, it aims at solving dependency management issues experienced by many .NET projects. You can read more about NuGet on NuGet.org.

TeamCity offers several NuGet-related features:

- keep an up-to-date NuGet version on all agents and on TeamCity server
- new build runners
    - NuGet Installer - installs and updates NuGet packages on agents for subsequent build steps
    - NuGet Pack - creates NuGet package from a spec file
    - NuGet Publish - publishes NuGet package to a given NuGet feed
- TeamCity can act as a NuGet feed server
- with the help of NuGet Dependency trigger TeamCity can watch for changes in a specified NuGet package

TeamCity as NuGet feed server

If you publish NuGet packages as TeamCity artifacts you can enable NuGet Server feature on Administration -> NuGet Settings page.



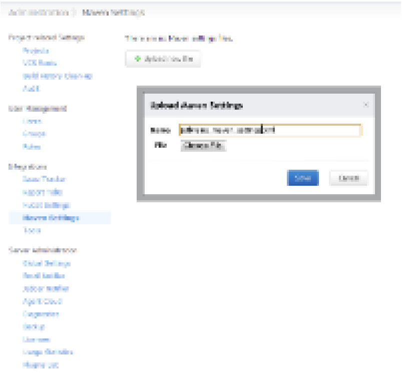
In this case TeamCity will provide its own NuGet feed with all published packages. You can use this feed in Visual Studio to get notifications when packages are updated on the server.

Learn more about TeamCity NuGet integration

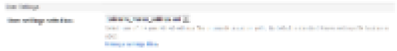
Maven

Maven settings

Maven settings files often contain shared settings for all Maven-based projects within a company. Until TeamCity 7.0 it was hard to define them in TeamCity as you had to store these settings in version control (or on a network share), and remember to use command line options to specify a path to them. Now you can upload your settings files to TeamCity:

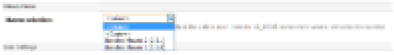


And then use them in Maven-specific build steps:



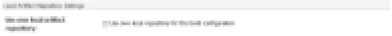
Bundled Maven 3

Maven 3 is now bundled with TeamCity in addition to Maven 2. You can select which Maven version to use in Maven build step.



Own local artifacts repository

In TeamCity 7.0 you can also isolate a specific build configuration artifacts repository from other local repositories. This can be done in one click on build step page.



Learn more about TeamCity Maven integration

Subversion 1.7 support

TeamCity 7.0 is compatible with Subversion 1.7:

- TeamCity can communicate with Subversion 1.7 servers
- TeamCity can create and update a 1.7 working copy on the agent in case of agent side checkout
- Visual Studio Add-in supports Subversion 1.7 for Remote Run and Pre-tested commit
- TeamCity Eclipse plugin supports Subversion 1.7 for Remote Run and Pre-tested commit

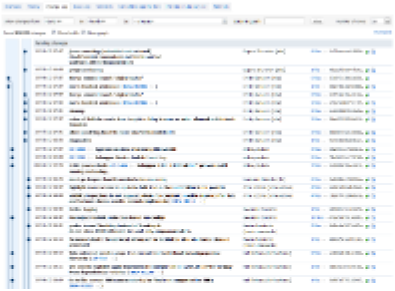
Since IntelliJ IDEA does not yet support SVN 1.7, TeamCity plugin for IntelliJ IDEA also won't work with 1.7.

User interface & usability improvements

Several user interface improvements have been made in TeamCity 7.0.

Branch graphs in change log

Project and build configuration change log now shows branch graphs for distributed version control systems like Git or Mercurial.

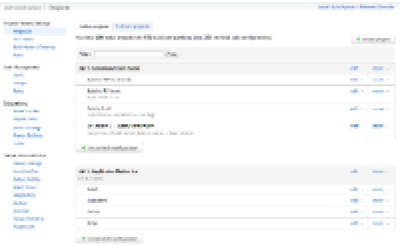


There will be as many graphs as the number of VCS roots you have in your project or build configuration. When clicking a node in the graph, corresponding part of the change log is highlighted.

Also in places where information about committer is shown, TeamCity will try to show the TeamCity user name instead of the version control user name.

New administration area

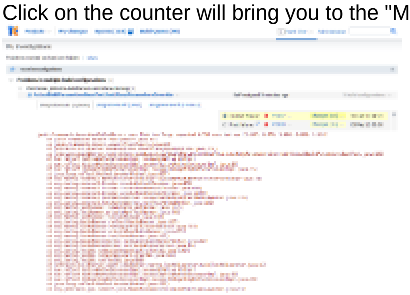
We restructured the TeamCity administration area to make it more logical, scalable and to bring some often used pages up front.



Current problems & investigations

If you are assigned as investigator for some problem, a counter near your name will be updated:

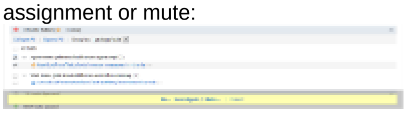
Click on the counter will bring you to the "My Investigations" page:



Note that for each failed test on this page you can get a bunch of information instantly, without a need to leave the page. For example, you can see all build configurations in which this test is currently failing. You can also see a current stacktrace and information about a build where the test is currently failing. You can also see information about the first failure of this test, again with stacktrace and build.

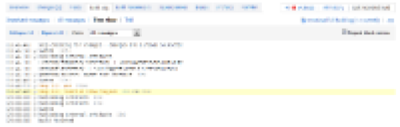
Similar improvements were also applied to project's "Current Problems", "Investigations", and "My Changes" pages.

We also decided to get rid of Bulk investigation mode introduced in TeamCity 6.5. Instead, we always show checkboxes for investigation assignment or mute:



Build log tree view

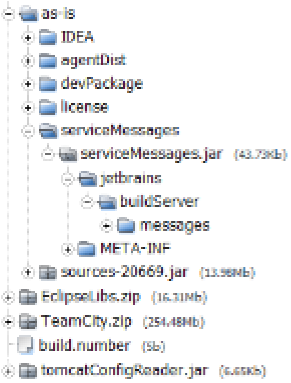
Build log tree view has got several improvements:



You can filter build log by message priority, expand all nodes in the whole build log, or within a single node only. Tree view is now available for running builds as well (it was not so before TeamCity 7.0).

Artifacts browsing

Browsing inside artifact archives from within the TeamCity UI has become extremely easy. Supported archive types are: .zip, .jar, .war, .ear, .tar, .tar.gz, .apk, .sit. Browsing archives inside archives is supported too.



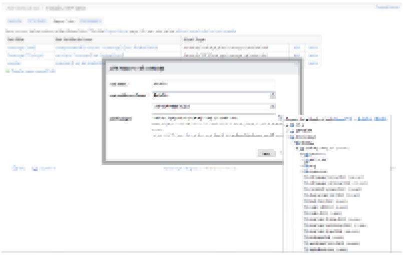
```
as-is
  IDEA
  agentDist
  devPackage
  license
  serviceMessages
    serviceMessages.jar (43.73Kb)
      jetbrains
        buildServer
          messages
      META-INF
  sources-20669.jar (13.96Mb)
EclipseLibs.zip (16.31Mb)
TeamCity.zip (254.48Mb)
build.number (5b)
tomcatConfigReader.jar (6.65Kb)
```

If you want to download a file from an archive in a build script, use this URL syntax:

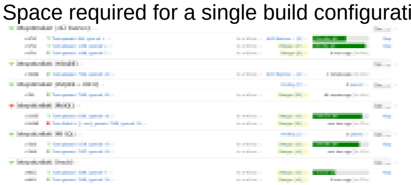
```
http://<server url>/repository/download/<build conf id>/<build>/<archive>!<path in archive>
```

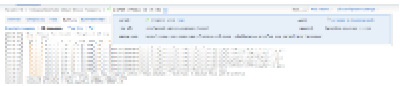
See more on this in our documentation

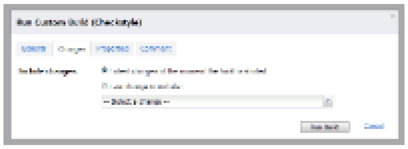
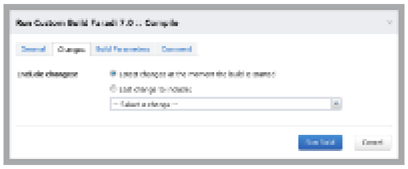
We also removed the "Base Path" field in project report tabs, as its only purpose was to handle paths inside archives.

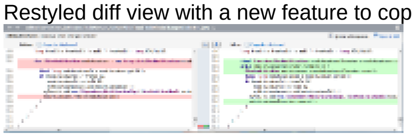


Small, but noteworthy changes

- Space required for a single build configuration on overview page has been reduced to fit more data:

- Quick navigation between build configurations of a current project is accessible from breadcrumbs in user and administration area:

- While staying on a build log, it is sometimes necessary to see the build summary quickly - now you can easily access it from each build tab:

- Layout and styles in most of the dialogs have been improved:

Before	After
	

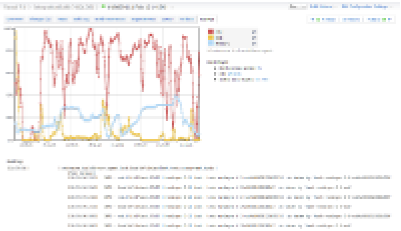
- Restyled diff view with a new feature to copy file content to clipboard:

- Run button has become much more usable and good looking:

- My Changes page shows estimates for queued builds on Builds tab.
- Newly created projects are not added to the overview automatically (yellow prompt is shown instead).
- Text fields that accept command line arguments, as well as text fields used for parameter values now auto-expand as you type and support multiline input.
- All projects popup now supports filtering by build configuration name, in addition to filtering by project.
- Changes by current user are highlighted in the changes popup, also too long comments are clipped.
- When TeamCity is able to determine agent operating system, a corresponding icon is shown near the agent name.
- Java coverage report presentation improved.
- A more detailed presentation of the test details (test history) page.

Other improvements

Performance monitor

New "Performance Monitor" build feature can help you to track hardware resources used by your build. Upon build finish you'll see an additional tab called "PerfMon" with results like:



Here you can see that build produced zero CPU and Disk usage during a period of several minutes, starting at 15:00. Most likely there was some code spending a lot of time in wait state. By clicking a point in the chart, a part of build log for this period of time will be shown.

Performance monitor spawns a separate process which gathers hardware statistics. Statistics is gathered for the whole agent machine, so it won't work properly if you have more than one agent installed on the same machine and these agents run builds in parallel. CPU, Disk and Memory usage are all percentages of what is available on the agent. Sometimes you can see that only 50% of CPU is being used by a build. This usually means that a build does not utilize all CPU cores of the agent machine. Platforms supported by PerfMon are Windows, Solaris, Linux and MacOS X.

Per-check-in builds

VCS trigger now supports two new options:

- Trigger one build per each checkin
- Trigger one build per group of checkins from the same user

With the first option enabled, each new check-in will result in a build added to the queue and associated with this check-in (i.e. build won't include other changes).

If the second option is enabled, and there are several pending changes, TeamCity will group them by user and will start builds having single user changes only.

Clean sources

Option to run a build with clean sources was added to custom build dialog and to "Schedule Trigger".

Build execution timeout

Global, per-server build execution timeout can be defined for all of the builds. If build configuration level timeout is specified, it will override the global setting.

Diagnostics page

Diagnostics page now shows current memory usage and JVM arguments. In case of memory issues (high memory usage for a prolonged period of time), a warning will be shown in TeamCity web interface. If you customized some TeamCity internal properties, you can see them on Diagnostics page too.

Artifact dependencies

Artifact dependencies now support syntax similar to checkout rules, i.e. you can define a set of new line delimited rules:
`[+:|-:]SourcePath[!ArchivePath][=>DestinationPath]`

Enable / disable parts of build configuration

Settings like build triggers, build steps, build features and build failure conditions can be disabled in build configuration or template. Disabled settings will stay inactive until they are enabled again. If you disable a build step, this build step will not be executed on the agent. Inherited settings can be disabled too, for example, if a build step is inherited from a template, you can disable it in the build configuration and define some other step instead of inherited one.

Agent hardware information

Plugin providing information about agent hardware was available for quite some time (see http://youtrack.jetbrains.net/issue/TW-4857). We decided that it is useful enough to bundle it with TeamCity distribution.
The plugin provides the following parameters:

- `teamcity.agent.hardware.memorySizeMb` - agent physical memory
- `teamcity.agent.hardware.cpuCount` - number of CPUs (cores) on agent
- `teamcity.agent.work.dir.freeSpaceMb` - available free space in Mb on a disk where agent work directory resides, updated on

every build finish

Artifacts cleanup

Ant-like file patterns can be specified for artifacts cleanup rules, in the form:
`[+:|-:]Ant_like_pattern`

VCS username

VCS username, default for all of the types of version controls, can be specified on user profile.

Viewing personal builds of other users

To see personal builds of other users enable "Show all personal builds" setting on your profile.

Ruby environment configurator

First of all, the long standing bug with non-functional Ruby environment configurator (http://youtrack.jetbrains.net/issue/TW-14803) has now been fixed.

Also, agents now detect the presence of RVM and report rvm_path environment variable with the path to RVM. This way you can easily set up your build configurations to run on agents where RVM is installed. Enabling of Ruby environment configurator will add this requirement automatically.

Previous Releases

What's New in TeamCity 6.5

Getting Started

In this section:

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

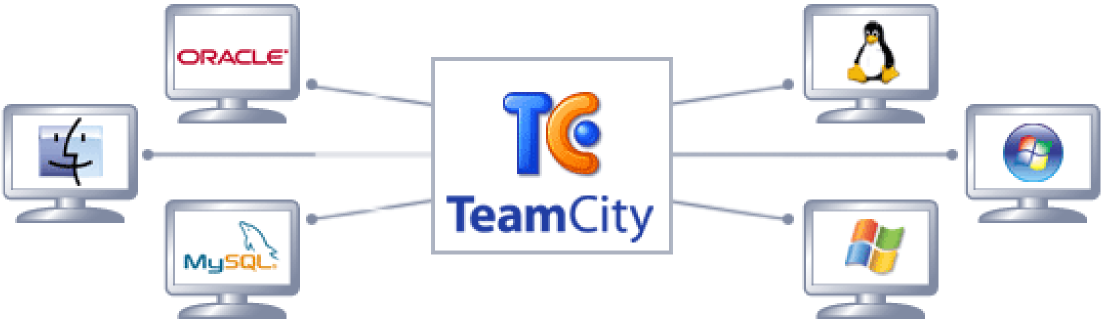
TeamCity is a user-friendly continuous integration (CI) server for developers and build engineers that is easy to set up and free of charge for small and medium teams. With TeamCity you can:

- 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 environment can be found here.

TeamCity Architecture

Unlike some build servers, TeamCity has distributed build grid architecture, which means that TeamCity build system comprises of 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 server performance.

TeamCity's Build Agents 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 server and agent, what is passed to agents, how and when TeamCity gets the results, and so on. Let's sort this out by considering a simple case of build lifecycle in TeamCity.

▼ Related Documentation Pages

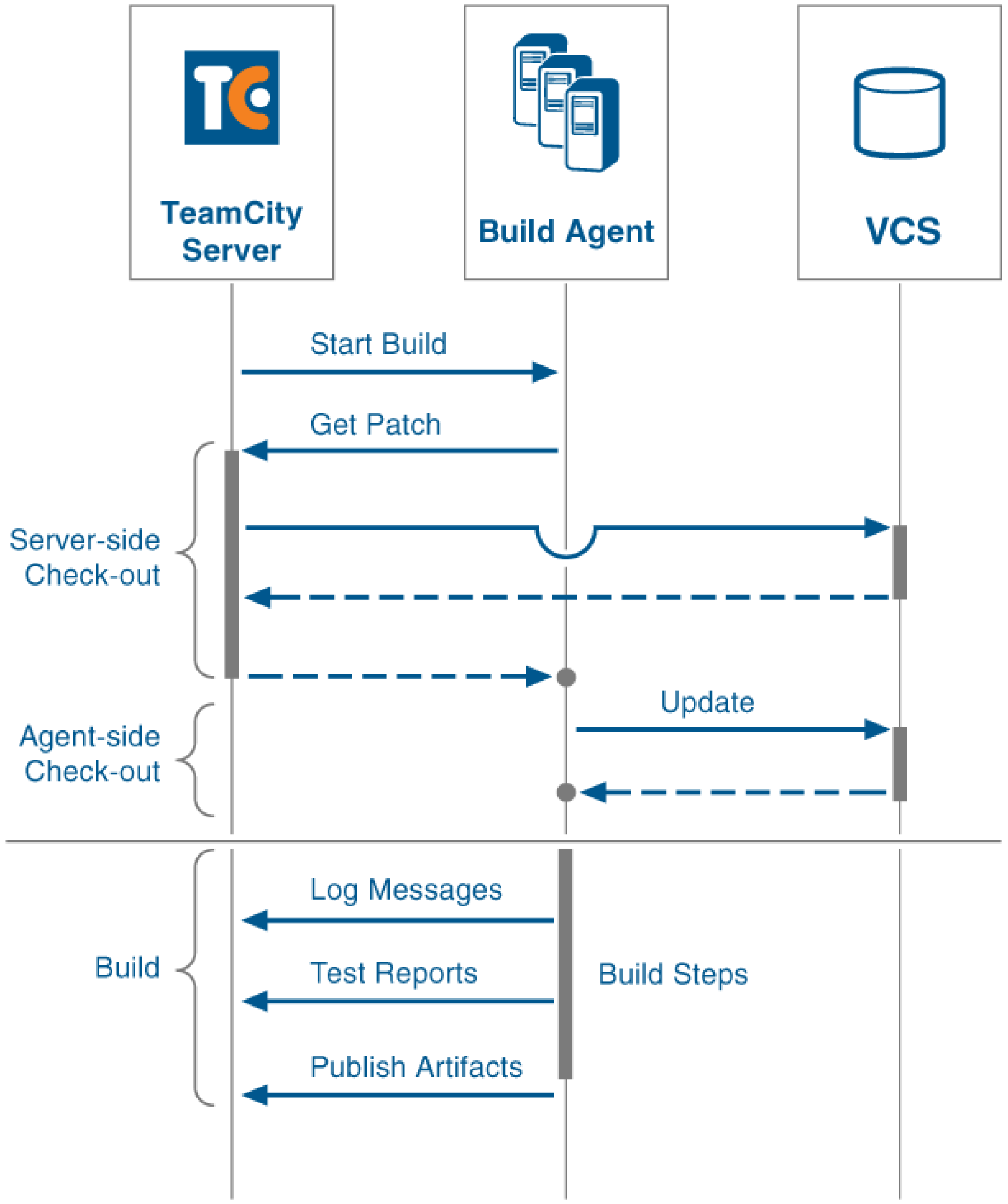
Installing and Configuring the TeamCity Server
Build Agent
Build Grid
Setting up and Running Additional Build Agents

Build Lifecycle in TeamCity

To demonstrate 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, VCS, TeamCity server and build agent are 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 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 TeamCity server tries to select the fastest agent based on the history of similar builds, and of course it selects an agent with appropriate environment. If there's no idle build agents among the compatible agents, the build is placed to the build queue, where it waits to be assigned to a particular agent. Once the build is assigned to a build agent, the build agent has to get the sources to run on.

At this point, TeamCity provides two possible ways how the build agent can get sources needed for the build:

- Server-side Checkout
- Agent-side Checkout

Server-side checkout
If server-side checkout is used, TeamCity server exports the required sources and passes them to the build agent. Since the build agent itself doesn't interact with your version control system, you don't need to install VCS client on agents. However, since sources are exported rather than checked out, no administrative data is stored in the file system and build agent cannot perform version control operations (like check in 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: TeamCity server sends to the agent incremental patches to update on the agent only the files changed since the last build.

Agent-side checkout
If agent-side checkout is used, the build agent itself checks out the sources before the build. 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 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, 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, build agent sends to the server **Build Artifacts**, 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 following simple steps:

1. Create a project (click to expand)

Start working with TeamCity by creating a project: a project is a collection of your build configurations. It allows you to organize your own projects and adjust security settings: you can assign users different permissions for each project.

Just click the *Create Project* link, then specify project's name and add some optional description.

2. Create a build configuration (click to expand)

When you have created a project, TeamCity suggests to populate it with a build configuration:

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 build configuration, so click the *add a build configuration* link. Specify general settings for your build configuration, like:

- Build configuration name
- 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 build number format will be maintained by TeamCity and will be resolved into a next integer value on each new build start. The counter you can specify in the **Build counter** field.
- Atrifact 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 build, specify here paths to such artifacts. Learn more.

If needed, specify when build should be considered as failed and click the **VCS Settings** button to proceed.

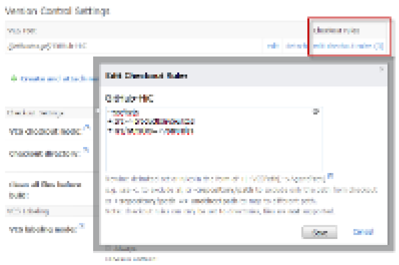
3. Specify sources to be build (click to expand)

To be able to create a build TeamCity has to know where the source code resides, thus setting up 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 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 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 build agent different from the default. 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 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 dependant on each other, using properties and so on.

Happy building!

Concepts

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

- Agent Home Directory
- Agent Requirements
- Agent Work Directory
- Already Fixed In
- Authentication Scheme
- Build Agent
- Build Artifact
- Build Chain
- Build Checkout Directory
- Build Configuration
- Build Configuration Template
- Build Grid
- Build History
- Build Log
- Build Number
- Build Queue
- Build Runner
- Build State
- Build Tag
- Build Working Directory
- Change
- Change State
- Clean Checkout
- Clean-Up
- Code Coverage
- Code Duplicates

- Code Inspection
- Continuous Integration
- Dependent Build
- Difference Viewer
- First Failure
- Guest User
- History Build
- Notifier
- Personal Build
- Pinned Build
- Pre-Tested (Delayed) Commit
- Project
- Remote Run
- Role and Permission
- Run Configuration Policy
- Security
- Supported Platforms and Environments
- TeamCity Data Directory
- TeamCity Editions
- TeamCity Specific Directories
- Testing Frameworks
- User Account
- User Group
- Wildcards

Agent Home Directory

Build Agent Home Directory is the directory where the agent has been installed.

The Build Agent can be installed into any directory.
If you use TeamCity .tar.gz distribution or .exe distribution opting for Build Agent installation, the agent will be placed into `<TeamCity Home>/buildAgent`.
The default directory suggested by .exe or Java Web Start agent installation is `C:\BuildAgent`.

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

Agent Directories

The agent consists of:

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

Agent Files Modification

Agent configuration is the only directory designed to have files that might be edited by the user.
All the other directories should not be edited by the user.

Content of agent work directory can be deleted (but only entirely). This will result in clean checkout for all the affected builds.

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

Important Agent Files and Directories

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

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

Agent Requirements

Agent requirements are used in TeamCity to specify whether a build configuration can run on a particular build agent.

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

The administrator can specify required environment variables and system properties for a build configuration on the build configuration's **Agent Requirements** page. For instance, if a particular build configuration must run on a build agent running Windows, the administrator specifies this by adding a requirement that the `os.name` system property on the build agent must contain the `Windows` string.

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

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

Implicit Requirements

Any reference (name in %-signs) to an unknown parameter is considered an "implicit requirement". That means that the build will only run on the agent which provides the parameters named.
Otherwise, the parameter should be made available for the build configuration by defining it on the build configuration or project levels.

For instance, if you define a build runner parameter as a reference to another property: `%env.JDK_16%/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 **Build Parameters** page (in the latter case, the same value of the property for all build agents will be used).

See also:

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

Agent Work Directory

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

To modify the default directory location, see 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 Scheme

TeamCity supports the following authentication schemes:

- **Default Authentication** (cross-platform): A database of users is maintained by TeamCity. New users are added by the TeamCity administrator (in the administration area section) or they can register themselves if the `<free-registration allowed="true" />` tag is specified.
- **NT Authentication** (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 Authentication** (cross-platform): Authentication is performed by directly logging into LDAP with credentials entered into the login form.

Please refer to Configuring Authentication Settings for specific authentication scheme configuration.

> ✓ TeamCity server supports basic HTTP authentication allowing to access certain web server pages and perform actions from various scripts. Please refer to the Accessing Server by HTTP page for details.

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 that actually executes a build process. It is installed and configured separately from the TeamCity server. An agent can be installed on the same computer as the server or on a different machine (the latter is a preferred setup for server performance reasons).

An Agent typically checks out the source code, downloads artifacts of other builds and runs the build process. Number of agents basically limits the number of parallel builds and environments in which your build processes are run.

TeamCity server monitors all the connected agents and assigns queued builds to the agents based on compatibility requirements.

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

In TeamCity, a build agent can have following statuses:

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

Authorized/ Unauthorized	Agents are manually authorized via the web UI on the **Agents** page. Only authorized build agents can run builds. The number of simultaneously authorized agents cannot exceed the number of agent licenses in your license pool. When an agent is unauthorized, a license is freed and a different build agent can be authorized. Purchase additional licenses to expand the number of agents that can concurrently run builds. When a new agent is registered on the server for the first time, it is **unauthorized** by default and requires manual authorization to run the builds. If a build agent is installed and running on the same computer as TeamCity build server, it is authorized automatically.
Enabled/ Disabled	Agents are manually enabled/disabled via the web UI. The TeamCity server only distributes builds to agents that are enabled. 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 name.

Only users with certain roles can manage agents. See Role and Permission for more information.

For a build agent configuration please refer to Build Agent Configuration section.

Agent Upgrade

TeamCity agent is upgraded automatically when necessary. The process involves downloading new agent files from the TeamCity server and restarting the agent on new files. In order to successfully accomplish this, the user under which agent runs should have enough permissions.

Typically, an agent upgrade happens on:

- server upgrade
- agent plugin added or updated on the server
- new tool installed (e.g. handle.exe for Swabra or new version of NuGet)

See also:

Concepts: Build Grid | Agent Work Directory | TeamCity Editions | Role and Permission
Installation and Upgrade: Installing and Running Build Agents
Administrator's Guide: Assigning Build Configurations to Specific Build Agents

Build Artifact

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

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

TeamCity stores artifacts on disk in a directory structure that can be accessed directly (for example, by configuring the Operating System to share the directory over the network). The artifacts are stored under `<TeamCity data directory>/system/artifacts` directory. Storage format is described in the TeamCity Data Directory#artifacts section. The artifacts are stored on the server "as is" without additional compression, etc.

All artifacts stay on the server and are available for download until they are cleaned up.

Build artifacts can also be uploaded to the server while the build is still running. To instruct TeamCity to upload the artifacts, 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 `maven-build-info.xml` file that contains Maven-specific data collected during the build. 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 `.teamcity` directory in the root of build's artifacts.
- Hidden artifacts are not listed at 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.

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's Properties tab.
- `.NETCoverage` - raw .Net coverage data (e.g. used to open dotCover data in VS addin)
- `coverage_idea` - raw IntelliJ IDEA coverage data (e.g. used to open coverage in IDEA)

See also:

> **Concepts**: Dependent Build
> **Administrator's Guide**: Configuring General Settings | Configuring Dependencies | Patterns For Accessing Build Artifacts

Build Chain

Build chain is a sequence of builds interconnected by snapshot dependencies. Thus, all the builds in a chain use the same sources snapshot.

The most common use case for specifying a build chain is running the same test suit 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 firstly run an integration build and after that a release build.

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

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

Configuring Build Chains

In TeamCity, you can configure dependent builds when creating/editing build configuration and specify two different types of dependencies, namely:

- Artifact dependencies
- Snapshot dependencies

To specify the dependencies in your build configuration:

1. Navigate to the **Administration** and create a new build configuration or for the already existing build configuration, navigate to the **edit** link, click the **arrow** icon to access the drop-down list and select **Dependencies**.
2. On the **Dependencies** page which opens click the **add new dependency** link under the **Snapshot Dependencies** or **Dependencies by Artifacts**.

Stopping/Removing From Queue a Build from Build Chain

If a build being stopped or removed from build queue is a part of Build Chain, there is a message below 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 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 listed build depends on the original build being stopped).

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

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

If user has no "View" access right for a build, which is a part of the build chain, this build is not listed at all. If there is at least one such build, we show a yellow stripe with warning: **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 the 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 is no running or queued builds for the build chain (i.e. all other parts of the build chain has finished), no additional information is shown.

Build Chains Visual Representation

Basically, each build chain is a directed acyclic graph, i.e. it cannot cannot have cycles.
You can review build chains on both project and build configuration pages: at each of those page you can find a 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 on 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.
- builds are shown in the following order: builds that start first are to 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 re-run some build in the chain.

See also:

Concepts: Dependent Build
Administrator's Guide: Configuring Dependencies

Build Checkout Directory

The *build checkout directory* is the directory on agent where all of the project's sources reside. If you use agent-side checkout mode, build agent checks out the sources into this directory before the build. In case you use server-side checkout mode, 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, default (empty) value is recommended. See Custom checkout directory.
If not specified, automatically created directory is used with a generated name: `<Agent Work Directory>/<VCS settings hash code>`.
The VCS settings hash code is calculated based on VCS roots and VCS settings used by the build configuration. Effectively, this means that the directory is shared between all the build configurations with the same VCS settings.

If you want to investigate some issue and need to know the directory used by a build configuration, you can get the directory from the build log, or you can refer to `<Agent Work Directory>/directory.map` generated file which lists build configurations with their last used directories.

In your build script you may refer to the effective value of build checkout directory via the `teamcity.build.checkoutDir` property provided by TeamCity.

Custom checkout directory

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

With this default checkout directory TeamCity ensures best performance and consistent incremental sources updates.

If for some reason you need to specify custom checkout directory (for example, the process of creating builds depends on some particular directory), please ensure 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 int he directory);
- the content of the directory is not modified by other processes than 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 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, another agent is appearing as an external process to it.

Please also 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 disk if not used (no builds were run on the agent using the directory as checkout directory) for a specified period of time (8 days by default).
(Please also 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 new value (in hours) by either of the following ways:

- `'system.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 finish.
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 background and can be paused by consequent builds.

See also:

> **Administrator's Guide**: Configuring VCS Settings

Build Configuration

Build Configuration describes a class of builds of a particular type, or a procedure used to create builds. Examples of build configurations are *integration builds*, *release builds*, *nightly builds*, and others.

You can explore details of a build configuration on its home page and modify its settings on the editing page. In this section:

- Build Configuration State
- Build Configuration Status
- Build Configuration ID

Build Configuration State

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

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

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

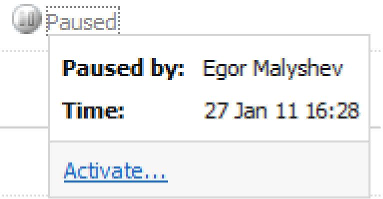
- On the Build Configuration settings page click the **Pause**/**Activate** button.
- Navigate to the Build Configuration Home Page, click **Actions** on the toolbar and select **Pause build configuration**/**Activate build configuration** from the drop-down.

Build Configuration Status

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

Personal builds do not affect the build configuration state.

You can view the build configuration status on the **Projects** page or Project Home Page, when the details are collapsed.

Icon	Description
🟩	The last build executed successfully.
🟥	The last build with this build configuration executed with errors or one of the currently running builds is failing. The build configuration status will change to "failed" when there's at least one currently running and failing build, even if the last finished build was successful.
🔍	Indicates that someone has started investigating the problem, or already fixed it. (see Investigating Build Problems).
no icon	There were no finished builds for this configuration, status is unknown.
 Paused **Paused by:** Egor Malyshev **Time:** 27 Jan 11 16:28 Activate...	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.

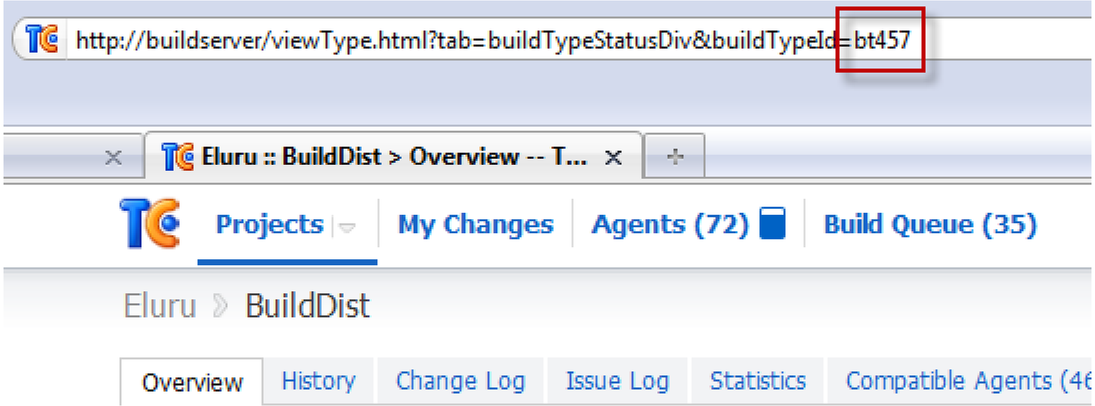
Build Configuration ID

Each build configuration has an internal unique identifier which you may need when constructing TeamCity URLs or editing TeamCity configuration manually.

Automatically assigned IDs use `bt<NUMBER>` format.

To determine the internal ID of a build configuration:

- Navigate to the build configuration home page and in page URL, find the `buildTypeID=btXXX` parameter (see picture below). The `btXXX` value is the value you search for.



http://buildserver/viewType.html?tab=buildTypeStatusDiv&buildTypeId=bt457

See also:

Administrator's Guide: Creating and Editing Build Configurations

Build Configuration Template

Build Configuration Template allows 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's settings affects **all** build configurations associated with this template.

How can I create build configuration template?

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

Associating build configurations with templates

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

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

Redefining settings inherited from template

Although a build configuration associated with a template inherits all it's settings, it is still possible to fine tune the some of the inherited settings: you can extend the set of inherited settings or redefine some of them.
Inherited settings cannot be deleted.

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 all text fields or your template settings except parameter names, build step names, agent requirements and passwords, you can use configuration parameters instead of actual values. Thus you can define the actual value of this parameter in each particular associated build configuration separately. Refer to the Configuring Build Parameters for an example of configuration parameters usage.

Other settings: drop-downs, lists, check boxes, etc.
These settings are inherited from template as is and cannot be redefined in an associated build configuration.

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

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

Refer to Configuring Build Parameters for more details on redefining settings.

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

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

See also:

> **Administrator's Guide**: Creating and Editing Build Configurations | Configuration Parameters

Build Grid

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

See also:

Concepts: Build Agents | Build Queue

Build History

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

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

Build Log

Build Log is a structured list of the events which took place during a 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.

Build log is available for browsing at the Build Results page.

You can download a full build log in a textual form from the Build Results page using the link

Download full build log

"Important messages" sub-tab displays the log messages filtered by "error" and "warning" status.

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 in a chain build
- Used in artifact dependencies
- Set with help of service messages

See also:

Administrator's Guide: Build number format

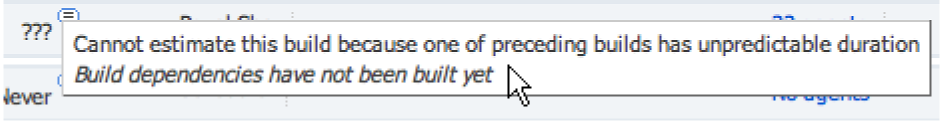
Build Queue

The build queue is a list of builds that were triggered and are waiting to be started. TeamCity will distribute them to compatible build agents as the agents become idle. The build is assigned to the agent only when it is started on the agent, no pre-assignment is made while the build still waits in the build queue.

When a build is triggered, firstly it is placed into the build queue, and, when a compatible agent becomes idle, TeamCity will run the build. The list of builds waiting to be run can be viewed on the **Build Queue** tab. This tab displays the following information:

- Sequence number of the build;
- Build configuration name in the following format: **<project name>::<build configuration name>**, where project and build configuration names are links to corresponding overview pages;
- Time to start: the estimated time that the build configuration will be queued before it starts to build and its estimated time of completion. Hovering the mouse cursor over the estimated time value shows a tooltip with expected start time and duration, and the links to the build results and Agents pages. If current build is a part of build chain and 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 correctly, so the relevant message will be displayed in the tooltip, for example:



Cannot estimate this build because one of preceding builds has unpredictable duration
Build dependencies have not been built yet

- Brief description of the event that triggered the build (**Triggered by** column). Learn more about Configuring Build Triggers.
- The number of agents compatible with this build configuration. You can click agent's name link to open Agents page, or use down arrow to quickly view the list of compatible agents in the popup window.

You can reorder the builds in queue manually, remove build configurations or personal builds from queue, and, if you have System Administrator permissions, you can assign different priorities to build configurations, which will affect their position in the queue. Learn more.

See also:

Concepts: Build Chain
Administrator's Guide: Ordering Build Queue

Build Runner

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

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

TeamCity comes bundled with the following runners:

- .NET Process Runner
- Ant
- Command Line — run arbitrary command line
- Duplicates Finder (.NET)
- Duplicates Finder (Java)
- FxCop
- Gradle
- Inspections — a set of IntelliJ IDEA inspections
- Inspections (.NET) — a set of ReSharper inspections
- IntelliJ IDEA Project, and an older version of it: Ipr (deprecated)
- Maven
- MSBuild
- MSpec
- MSTest
- NAnt
- NuGet-related runners
- NUnit
- PowerShell
- Rake
- Visual Studio (sln) — Microsoft Visual Studio 2005/2008/2010 solutions
- Visual Studio 2003 — Microsoft Visual Studio 2003 solutions

Technically, build runners are implemented as plugins.

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

See also:

Administrator's Guide: Configuring Build Steps

Build State

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

Build States

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

Personal Build States

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

Hanging and Outdated Builds

TeamCity considers a build as *hanging* when its run time significantly exceeds estimated average run time and the build did not send any messages since the estimation exceeded.
A running build can be marked as *Outdated* if there is a build which contains more changes but it is already finished.

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

Failed to Start Builds

Builds failed to start due to a configuration error are marked with the icon ⚠ .
For example, VCS repository can be down when build starts, or artifact dependencies can't be resolved, and so on. In case such error occurs, TeamCity:

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

See also:

Concepts: Build Configuration Status | Change | Change State
User's Guide: Viewing Your Changes

Build Tag

Build tags are labels that can help you to:

- organize history of your builds
- quickly navigate to the builds marked with the specific tag

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

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

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

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

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

Additionally you can search for builds with particular tags using the search field on the **Projects** page.

Build Working Directory

The *build working directory* is the directory set as current for the build process. By default, this is the same directory as the Build Checkout Directory.
If the build script needs to run from a location other than the checkout directory, then you can specify it explicitly using the **Working Directory** field on the settings page of the **Build Runner** settings.

> Not all build runners provide the working directory setting.

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

See also:

> **Concepts**: Build Checkout Directory

Change

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

TeamCity suggests several ways to view changes:

- My Changes page shows the list ofyour changes and how they have affected different builds.
- Pending changes are accessible from the **Projects** page, build configuration page, or build results page.

Viewing and analyzing changes involves the following possibilities:

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

See also:

> **Concepts**: Build Configuration
> **User's Guide**: Investigating Build Problems

Change State

Icon	State	Description
⌛	pending	The change is scheduled to be integrated into a build that is currently in the build queue. 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.
⚙	running successfully	The change is being integrated into a build that is running successfully.
✓	successful	The change was integrated into build that finished successfully.
⚙	running and failing	The change is being integrated into a build that is failing.
●	failed	The change was integrated into a build that failed.

Personal Change States

Icon	State	Description
👤⌛	pending	The change is scheduled to be integrated into a personal build that is currently in the build queue.
👤⚙	running successfully	The change is being integrated into a personal build that is running successfully.
👤✓	successful	The change was integrated into a personal build that finished successfully.
👤⚙	running and failing	The change is being integrated into a personal build that is failing.
👤●	failed	The change was integrated into a personal build that failed.

See also:

Concepts: Build Configuration Status | Build State | Change
User's Guide: Viewing Your Changes

Clean Checkout

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

Enforcing Clean Checkout

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

You can enforce clean sources action for a build configuration from the Build Configuration home page (**Actions** drop-down in the top right corner), or for an agent from the Agent Details page. The action opens a list of agents/build configurations to clean sources for.

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

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

TeamCity maintains an internal cache for the sources to optimize communications with the VCS server. The caches are reset during the cleanup time. To resolve problems with sources update, the caches may need to be reset manually. To do this, just delete `<TeamCity Data Directory>/system/caches` directory.

Automatic Clean Checkout

If clean checkout is not enabled, TeamCity updates the sources in the checkout directory incrementally to the required state.
TeamCity tries to detect if the sources in the checkout directory are not corresponding to the expected state and triggers clean checkout in such cases to ensure sources are appropriate.

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

TeamCity performs automatic clean checkout in the following cases:

- Build checkout directory was not found or is empty (either the build configuration is started on the agent for the first time or the directory has disappeared since the last build). This also covers
  - particularly when no builds were run in a specific checkout directory for a configured (or default) time and the directory became empty. See more at automatic checkout directory cleaning.
  - particularly when there were not enough free space on disk in one of the earlier builds and the directory was deleted
- a user invoked "Enforce clean checkout" action from the web UI for a build configuration or agent
- the build was triggered via Custom Run Build dialog with "Clean all files in checkout directory before build" option selected or by a trigger with corresponding option
- VCS settings of the build configuration were changed
- the previous build in this directory was of a build configuration with different VCS settings (can only occur if the same checkout directory is specified for several build configurations with individual VCS settings and VCS Roots)
- the previous build in this directory was built on more recent revisions then 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.

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

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

Clean-Up

TeamCity has a feature of automatic deletion of the data belonging to old builds. It is called Clean-up and can be run manually or each day by schedule (which is recommended).
Clean-up settings are configured under **Administration | Project-related Settings | Build History Clean-up**.
It is recommended to configure clean-up rules to remove builds that are no longer necessary to free disk space, remove builds from TeamCity UI and reduce TeamCity load processing no longer necessary data.
The cleanup deletes the data stored under `TeamCity Data Directory/system` and in the database. Also, during cleanup time the server performs various maintenance tasks (e.g. resets VCS full patch caches).

Clean-up Rules

A *clean-up* rule defines when and what data to clean. The default clean-up rule applies to all build configurations and can be configured. However, you can define a clean-up rule for a specific build configuration to override default. In each rule, you can define a number of successful builds to preserve, and/or a length of time that builds should be kept in history (e.g. keep builds for 7 days).

The following cleanup levels are available:

- **Clean artifacts** (all other data is preserved. Hidden Artifacts are also preserved);
- **Clean history** (all the build data is deleted except for builds statistics values that are visible in the statistics charts);
- **Clean everything** (no build data remains in TeamCity).
  Each level includes all listed before it.

For each of the above items you can specify:

- the number of days. Builds older then the number of days specified will be cleaned with the specified level. A day is equivalent to a 24-hour period, not a calendar day;
- the number of successful builds. Only builds older then the last matching successful build will be cleaned with the level specified (all the failed builds between preserved successful ones are kept).

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

- pinned builds;
- builds used as a source for artifact dependency in other builds when "Prevent dependency artifacts clean-up" option is enabled. See Clean-Up for Dependent Builds below;
- builds of build configurations that were deleted less then one day ago.

Clean-up for Dependent Builds

TeamCity preserves builds that are used in other builds by either snapshot dependencies or artifact dependencies.

This behavior is regulated by the option "Prevent dependency artifacts clean-up" in **Dependencies** section of the **Edit Clean Up Rules**

When the option is ON, it protects artifacts of the builds that were used as a source of artifact dependencies for the builds of the current build configuration.
This option is OFF by default. Also you can set default value for this option in the default cleanup rule.

Example:
Say, a build configuration A has an artifact dependency on B. If **Prevent dependency artifacts clean-up** option is enabled for A, the builds of B that provided artifacts for the builds of A will not be processed while cleaning artifacts, so that artifacts will be preserved.

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:

- **Java**: IntelliJ IDEA coverage engine, EMMA open-source toolkit.
- **.NET**: NCover 1.x, 3.x, PartCover and dotCover.

To get the code coverage information displayed in TeamCity, you need to configure it in the dedicated section of a build runner settings page. The following build runners include code coverage support:

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

Note, that currently Maven2 runner supports only IntelliJ IDEA coverage engine.
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:

Moreover, the chart is available for code coverage on the Statistics tab of the build configuration.

For the details on configuring code coverage, please, 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

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

A code inspection is an automated code review that verifies and reports the quality of your code by looking for common problems and anti-patterns. More than 600 Java (as well as HTML, CSS, JavaScript) inspections are preformed by the **Inspection** Build Runner.

Inspection results are reported in the Code Inspection tab of the build results page.

See also:

> **Concepts**: Build Runner
> **Administrator's Guide**: Inspections

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

Artifact dependency is just a way to get artifacts produces by one build in another. Without corresponding Snapshot dependency they are mainly used when the build configurations are not related in terms of sources. For example one build provides a reusable component for others. Snapshot dependency influences the way builds are processed and implies that the builds are deeply related with one build being a logic part of another one.

Snapshot Dependency

Snapshot Dependency is a powerful concept that allows to express dependencies between the build configurations on the source level in TeamCity.
See also Build Dependencies Setup.

A snapshot dependency from build configuration A to build configuration B enforces that each build of A has a "suitable" build of B so that both builds use the same sources snapshot (used sources revisions correspond to the same moment).

Snapshot dependency alters the builds behavior in the following way:

- when a build is queued, also queued are the builds from all the Build Configurations it snapshot-depends on;
- build does not start until all it's snapshot dependencies are ready;
- when a first build in the the build chain is ready to start, changes are checked for the entire build chain. If some of the build configurations already have a finished build with matching changes and the snapshot dependency has option "Do not run new build if there is a suitable one" ON, the queued matching builds are not run and are dropped.
- 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 snapshot dependency), the revision to use is determined upon the start of the first build in the chain. 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 that artifacts are downloaded from the same-sources build.

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 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 the build B starts to run on the agent, TeamCity adjusts the sources to include in the build A at this exact moment. All builds will be run with sources taken on the moment the build B started to run on a build agent.

> ℹ If the build configurations connected with 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, then TeamCity will start to run build A.

> ℹ Please note, that the changes to be included in the build A could become not the latest ones to the moment of build start to run. In this case build A becomes a history build.

The above example shows the core basics of the snapshot dependencies - straight forward process without any additional options. For snapshot dependency options refer to the Snapshot Dependencies page.

If a build has snapshot dependencies on several builds, the snapshot is taken at the moment the first build of the whole set (chain) starts to run on a build agent. Depending on the dependencies topology builds could be subsequent or can be started in parallel.

Two or more builds connected by snapshot dependencies form the Build Chain.
By default TeamCity preserves builds that are a part of a chain from clean-up, but a user can switch off the option. Refer to the Clean-Up description for more details.

See also related "Trigger on changes in snapshot dependencies" setting of a VCS trigger and "Show changes from snapshot dependencies" check box on "Version Control Settings" configuration section.

Artifact Dependency

Artifact Dependencies provide you with a convenient means to use 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 a build or what build used artifacts of the current build on a **Dependencies** tab of build results.

To create and configure an artifact dependency use the **Dependencies** 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 snapshot dependency and artifact dependency are configured for the same build configuration, in order to take artifacts from the build with the same sources **Build from the same chain** option must be selected in artifact dependency.

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 artifacts downloaded by this configuration from other builds can be cleaned or not. This setting is available on the cleanup policies page.

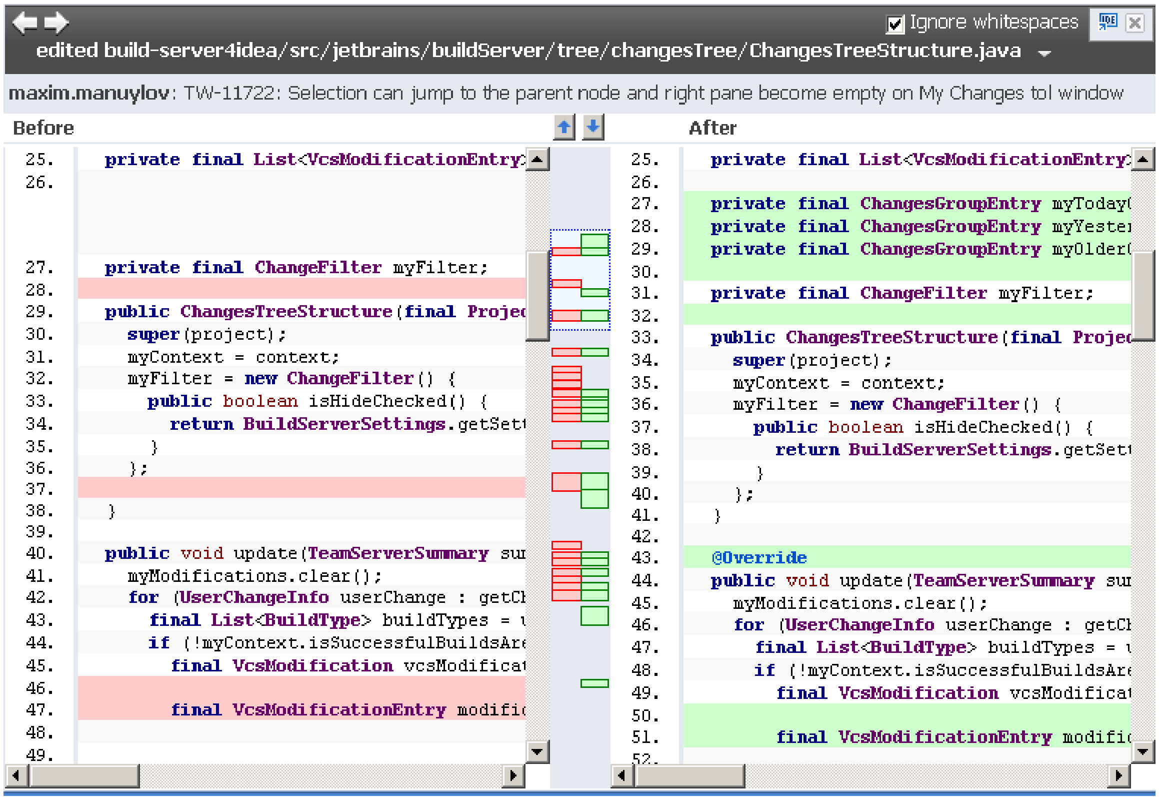
See also:

Concepts: Build Artifact | Build Dependencies Setup
Administrator's Guide: Configuring Dependencies

Difference Viewer

TeamCity Difference Viewer allows to review the differences between two versions of a modified file and navigate between these differences. You can access it 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 and so on.

After you click the modified file name, a new window opens:



edited build-server4idea/src/jetbrains/buildServer/tree/changesTree/ChangesTreeStructure.java

maxim.manuylov: TW-11722: Selection can jump to the parent node and right pane become empty on My Changes tol window

Before:

```
  private final List<VcsModificationEntry>


  private final ChangeFilter myFilter;

  public ChangesTreeStructure(final Projec
    super(project);
    myContext = context;
    myFilter = new ChangeFilter() {
      public boolean isHideChecked() {
        return BuildServerSettings.getSett
      }
    };

  }

  public void update(TeamServerSummary sur
    myModifications.clear();
    for (UserChangeInfo userChange : getCh
      final List<BuildType> buildTypes = u
      if (!myContext.isSuccessfulBuildsAre
        final VcsModification vcsModificat

        final VcsModificationEntry modifi
```

After:

```
  private final List<VcsModificationEntry>

  private final ChangesGroupEntry myToday(
  private final ChangesGroupEntry myYester
  private final ChangesGroupEntry myOlder(

  private final ChangeFilter myFilter;

  public ChangesTreeStructure(final Projec
    super(project);
    myContext = context;
    myFilter = new ChangeFilter() {
      public boolean isHideChecked() {
        return BuildServerSettings.getSett
      }
    };
  }

  @Override
  public void update(TeamServerSummary sur
    myModifications.clear();
    for (UserChangeInfo userChange : getCh
      final List<BuildType> buildTypes = u
      if (!myContext.isSuccessfulBuildsAre
        final VcsModification vcsModificat

        final VcsModificationEntry modifi
```

The window heading displays the file modifications summary:

- file name alone with its status,
- changes author,
- comment for the changes list.

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

If you want to switch to your IDE and digg into the issue deeper, click the **Open in the IDE** button in the upper-right corner of the window. The file in question opens, and you will navigate to this particular change.

See also:

Concepts: Project | Build Configuration

Guest User

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

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

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

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

You can use guest login to download artifacts:

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

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

An administrator can enable guest login on the **Administration | Global Settings** page.

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

See also:

Concepts: Role and Permission
Administrator's Guide: Enabling Guest Login

History Build

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

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

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

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

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

See also:

Concepts: Build History | Build Queue
Administrator's Guide: Triggering a Custom Build

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 initiated by the Remote Run procedure. Only users with "Project Developer" role can initiate a personal build.

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

See more at Pre-Tested (Delayed) Commit.

See also:

Concepts: Pre-tested Commit | Remote Run
Installing Tools: IntelliJ Platform Plugin | Eclipse Plugin | Visual Studio Addin

Pinned Build

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

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

See also:

Concepts: Clean-up policy

Pre-Tested (Delayed) Commit

An approach which prevents committing defective code into a build, so the entire team's process is not affected.

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

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

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

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

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

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

If TeamCity cannot match the changes with the builds, a message "Submitted changes cannot be applied to the build configuration" is displayed.

The VCS integration should be correctly configured in the IDE and TeamCity should be able to match the files on the developer workstation to the build configurations present on TeamCity server.
In order to be able to do that, VCS should be configured in the same way on the developer's workstation and on the server.
This includes:

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

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

General Flow of a pre-tested commit

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

See also:

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

Project

Project is a collection of build configurations. The project has a name (usually corresponds to the name of a real project) and can have some meaningful description.

TeamCity uses project level for managing security. It is possible to assign per-project roles to users. This makes it possible to assign users different permissions for each project.

You can create and change projects in the **Administration** area. Use the **Projects** page to configure the visible projects that display in the overview.

See also:

Concepts: Build Configuration

Remote Run

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

For a list of version control systems supported by each IDE please see supported platforms and environments.

See more at Pre-Tested (Delayed) Commit.

See also:

> **Concepts**: Pre-tested Commit | Personal Build
> **Remote Run on Branch:** Remote Run on Branch Build Trigger for Git and Mercurial
> **Installing Tools**: IntelliJ Platform Plugin | Eclipse Plugin | Visual Studio Addin

Role and Permission

Role is a set of *permissions* that can be granted to a user in one or all projects.
Permission is an *authorization* granted to TeamCity user to perform particular operations, for example run build, or modify build configuration settings.

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

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

Changing Authorization Mode

Unless explicitly configured, simple authorization mode is used when TeamCity is working in Professional mode and per-project is used when working in Enterprise mode.
To change the authorization mode, use the **Enable per-project permissions** check box at the **Administration**| **Global Settings** page.

Simple Authorization Mode

Administrator	Users with no restrictions (corresponds to **System Administrator** role in per-project authorization mode)
Logged-in user	Corresponds to default **Project Developer** role granted for all projects in per-project authorization mode
Guest user	Corresponds to default **Project Viewer** role granted for all projects in 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 cleanup rules, change shared VCS roots, and etc.
Project Administrator	Project Administrator is a person who can customize general settings of a project, build configuration settings and assign roles to the project users and 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 only read-only access to projects and can only *view* the project. Project Viewer role does not have permissions to view agent details.

When per-project permissions are enabled, server administrators can modify these roles, delete them, or add new roles with any combination of permissions right in 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 a 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

Security

User access levels are handled by assigning different roles to users. Each role provides a set of permissions and is assigned (or not assigned) for each project, thus controlling access to the projects and the various features in the Web UI.

See also:

Concepts: Role and Permission

Supported Platforms and Environments

This page covers software-related environments TeamCity works with. For hardware-related notes, please the notes.

In this section:

Platforms (Operating Systems)

The TeamCity Server

The TeamCity server is a web application that should be deployed into a capable J2EE servlet container. Core functions of TeamCity server are platform-independent.

Requirements:

- Java (JRE) 1.6+ (Java 1.6 is already included in the Windows .exe distribution). TeamCity is tested with Sun Java. Both 32bit and 64bit Java versions can be used. Note that OpenJDK is NOT supported.
- J2EE Servlet (2.5+) container, JSP 2.0+ container based on Apache Jasper. TeamCity is tested under Tomcat 7 which is a 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 recommended to use Tomcat 6.0.27+, earlier versions of Tomcat have some issues which can cause a deadlock in TeamCity on start-up.

> ⚠ TeamCity with native MSSQL driver is not compatible with Sun Java 1.6.0_29, due to bug in Java itself. You can use earlier or later versions of Sun Java.

TeamCity server is tested under the following operating systems:

- Linux
- MacOS X
- Windows XP
- Windows Vista/Windows Vista 64
- Windows 7/7x64
- Windows Server 2008
  under Tomcat 7 web application server.

Reportedly works on:

- Solaris
- FreeBSD
- IBM z/OS
- HP-UX

Build Agents

TeamCity Agent is a standalone Java application.

Requirements:

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

TeamCity agent is tested under the following operating systems:

- Linux
- MacOS X
- Windows 2000/XP/XP x64/Vista/Vista x64
- Windows 7/7x64

Reportedly works on:

- 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
- 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

Web Browsers

The TeamCity Web Interface is W3C-compliant, so just about any modern browser should work well with TeamCity. The following browsers have been specifically tested and reported to work correctly:

- Microsoft Internet Explorer 7+ (without 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 1.6-1.8 (TeamCity comes bundled with Ant 1.8.2)
- Maven versions 2.0.x, 2.x, 3.x (known at the moment of the TeamCity release). Java 1.5 and higher supported. (TeamCity comes bundled with Maven 2.2.1 and Maven 3.0.4)
- IntelliJ IDEA project runner
- Gradle (requires Gradle 0.9-rc-1 or higher)
- Java Inspections and Java Duplicates based on IntelliJ IDEA (requires Java 1.6)

Supported .Net platform build runners:

- MSBuild (requires .Net Framework or Mono installed on the build agent)
- 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) (requires a corresponding version of MS Visual Studio installed on the build agent)
- FxCop (requires FxCop installed on the build agent)
- Duplicates Finder for Visual Studio 2003, 2005, 2008 and 2010/2012 projects. Supported languages are C# up to version 4.0 and Visual Basic .Net version 8.0 - 10.0 (requires .Net Framework 2.0+ installed on the build agent)
- Inspections for .NET based on ReSharper code
- NuGet runners, supported NuGet versions 1.4+

Other runners:

- Rake
- Command Line Runner for running any build process using a shell script
- PowerShell
- .NET Process runner for running any .NET application

Testing Frameworks

- JUnit 3.8.1+, 4.x
- NUnit 2.2.10, 2.4.x, 2.5.x, 2.6.0 (dedicated build runner)
- TestNG 5.3+
- MSTest (dedicated build runner; requires appropriate Microsoft Visual Studio edition installed on build agent)
- MSpec

Version Control Systems

- Subversion (server versions 1.4-1.7 and any further as long as the protocol is backward compatible)
- CVS
- Git
- Mercurial (Mercurial "hg" client v1.5.2+ should be installed on the server)
- Team Foundation Server 2005, 2008, 2010, **since TeamCity 7.1** 2012 (Team Explorer should be installed on TeamCity server, available only on Windows platforms)
- Perforce (Perforce client should be installed on TeamCity server)
- Borland StarTeam 6 and up (StarTeam client application should be installed on TeamCity server)
- IBM Rational ClearCase, Base and UCM modes (ClearCase client should be installed on TeamCity server)
- Microsoft Visual SourceSafe 6 and 2005 (SourceSafe client should be installed on TeamCity server, available only on Windows platforms)
- SourceGear Vault (Vault command line client libraries should be installed on the server)

Checkout on agent

The requirements noted are additional to those listed above.

- Subversion (working copies in Subversion 1.4-1.7 format are supported)
- CVS

- Git (git v.1.6.4+ should be installed on the agent)
- Mercurial (Mercurial "hg" client v1.5.2+ should be installed on TeamCity agent machine)
- Team Foundation Server 2005, 2008 and 2010 (requires Team Explorer to be installed on the build agent, available only on Windows platforms)
- Perforce (Perforce client should be installed on TeamCity agent machine)
- IBM Rational ClearCase (ClearCase client should be installed on TeamCity agent machine)

Labeling Build Sources

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

Remote run on a branch

- Git
- Mercurial

Feature branches

since TeamCity 7.1

- Git
- Mercurial

VCS systems supported via third party plugins

- AccuRev
- Bazaar
- PlasticSCM

Issue Tracker Integration

- JetBrains YouTrack 1.0 and later (tested with latest version).
- Atlassian Jira 3.10 and later.
- 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: TeamCity 7.0 plugin supports Eclipse versions 3.3.2-3.7, **since TeamCity 7.1** Eclipse versions 3.4.2-3.8 and 4.2 are supported. Eclipse should be run under JDK 1.5+
- IntelliJ Platform plugin: compatible with IntelliJ IDEA 9.x - 11.x (Ultimate and Community editions); JetBrains RubyMine 2.0 - 4.5, JetBrains PyCharm 1.0-2.x, JetBrains PhpStorm/WebStorm 1.0-4.x
- Microsoft Visual Studio 2005, 2008, 2010 and **since TeamCity 7.1** 2012.

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	• Subversion 1.4-1.7 (with Subclipse and Subversive Eclipse integration plugins). • Perforce (P4WSAD 2008.1 - 2010.1, P4Eclipse 2010.1 - 2012.1) • ClearCase (client software is required) • CVS • Git (EGit 0.6 - 0.9 Eclipse integration plugin) see also
IntelliJ IDEA Platform	• ClearCase • Git (remote run only) • Perforce • StarTeam • Subversion • Visual SourceSafe
MS Visual Studio	• Subversion 1.4-1.7 (command-line client is required) • Team Foundation Server 2005 and later (installed Team Explorer is required) • Perforce 2006 and later (command-line client is required)

Code Coverage

IDE	Supported Coverage Tool
Eclipse	IDEA and EMMA code coverage
IntelliJ IDEA Platform	IDEA and EMMA code coverage
MS Visual Studio	dotCover coverage. Requires JetBrains dotCover 1.1 or 1.2 installed in Visual Studio, **since TeamCity 7.1** works with dotCover 2.0

External Databases

See more at Setting up an External Database

- HSQLDB 1.8
- MySQL 5.0.33+, 5.1.49+, 5.5+ (Please note that due to bugs in MySQL, versions 5.0.20, 5.0.22 and 5.1 up to 5.1.48 are not compatible with TeamCity)
- Microsoft SQL Server 2005, 2008 (including Express editions), 2012 (since TeamCity 7.1)
- PostgreSQL 8+
- Oracle 10g+ (TeamCity is tested with driver version 10.2.0.1.0)

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. (See also notes on backup for the description of the data stored in the directory and the database).

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

The currently used data directory location can be seen on **Administration** | **Global Settings** page for a running TeamCity server instance, or in `logs/teamcity-server.log` file (look for "TeamCity data directory:" line on the server startup).

Please note that in this documentation and other TeamCity materials the directory is often referred to as **`.BuildServer`**. If you have another name for it, please replace ".BuildServer" with the actual name.

TeamCity Windows installer configures the TeamCity data directory during the installation steps. The default path suggested for the directory is:
Since TeamCity 7.1: %ALLUSERSPROFILE%\JetBrains\TeamCity
TeamCity 7.0 and previous versions: %USERPROFILE%\.BuildServer

To change the location of the TeamCity Data Directory:

- under all OS if TeamCity is run via startup script/from command line: set `TEAMCITY_DATA_PATH` environment variable (either system-wide or for the user under whom TeamCity will be started)

- under Windows if TeamCity is run as a service
    - since TeamCity 7.1: the same as above: set `TEAMCITY_DATA_PATH` environment variable as global environment variable for all users
    - for TeamCity 7.0 and previous versions: set `teamcity.data.path` Tomcat web server service JVM property

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

Recommendations as to choosing Data Directory Location

Note that the `system` directory stores all the artifacts and build logs of the builds in the history and can be quite large, so it is recommended to place TeamCity Data Directory on a non-system disk. Please also refer to Clean-Up section to configure automatic cleaning of older builds.

Please note that TeamCity assumes reliable and persistent read/write access to TeamCity Data Directory and can malfunction if data directory becomes inaccessible. This malfunctions can affect TeamCity functioning while the directory is unavailable and may also corrupt data of the currently running builds. Still under rare circumstances the data stored it 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 single local disk cannot store all the data, consider placing the data directory on a local disk and mapping `.BuildServer/system/artifacts` to a larger disk with the help of OS-provided filesystem links.
Related feature request: TW-15251.

Structure of TeamCity Data Directory

`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 get more understanding on what data is stored in the database and what 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's OK to delete them manually.
    - *_notifications* — notification templates and notification configuration settings, including syndication feeds template.
    - *_logging* — internal server logging configuration files, new files can be added to the directory.
    - *<project name>* — configuration settings specific to a particular project, new directories can be created provided they have mandatory nested files
        - *project-config.xml* — main project configuration, contains configurations of a project's Build Configurations
        - *plugin-settings.xml* — auxiliary project settings, like custom project tabs content
        - *project-config.xml.N* and *plugin-settings.xml.N*--- backup copies of corresponding files created when a project's configuration is changed via web UI
        - *<buildConfigurationID>.buildNumbers.properties* — build number settings for a build configuration with internal id "buildConfigurationID"
    - *main-config.xml* — server-wide configuration settings
    - *database.properties* — database connection settings, see more at Setting up an External Database
    - *vcs-roots.xml* — VCS roots configurations file (both shared and not shared)
    - *roles-config.xml* — roles-permissions assignment file
    - *license.keys* — a file which stores the license keys entered into TeamCity.
    - *change-viewers.properties* — External Changes Viewer configuration properties
    - *internal.properties* — file for specifying various internal TeamCity properties. Is not present by default and should be created if necessary.
    - *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
    - *database.*.properties* — default template connection settings files for different external databases
    - **.dtd* — DTD files for the XML configuration files
    - **.dist* — default template configuration files for the corresponding files without `.dist`. See below.

- ***.BuildServer/plugins*** — a directory where TeamCity plugins can be stored to be loaded automatically on TeamCity start. New plugins can be added to the directory. Existing ones can be removed while the server is not running. The structure of a plugin is described in Plugins Packaging.
    - *.unpacked* — directory that is created automatically to store unpacked plugins. Should not be modified while the server is running. Can be safely deleted if server is not running.

- ***.BuildServer/system*** — a directory where build results data is stored. The content of the directory is generated by TeamCity and is not meant for manual editing.
    - *artifacts* — a directory where the builds' artifacts are stored. The format of artifact storage is *<project name>/<build configuration name>/<internal_build_id>*. If necessary, the files in each build's directory can be added/removed manually - this will be reflected in corresponding build's artifacts.
        - *.teamcity* directory in each build's directory stores build's hidden artifacts. The files can be deleted manually, if necessary, but build will lack corresponding feature backed by the files (like displaying/using finished build parameters, coverage reports, etc.)
    - *messages* — a directory where build logs are stored in internal format. Build logs store build output, test output and test failure details. Build with internal id "xxxx" stores it's log in CHyy/xxxx.* file, where "yy" are the last two digits of xxxx. The files can be

removed manually, if necessary, but corresponding builds will drop build log and test failure details.

    - *changes* — a directory where the remote run changes are stored in internal format. Name of the files inside the directory contains internal personal change id.
    - *pluginData* — a directory where various plugins can store their data. It is not advised to delete or modify the directory. (e.g. state of build triggers is stored in the directory)
        - *audit* — directory holding history of the build configuration changes and used to display diff of the changes. Also stores related data in the database.
    - *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. However, it's more safe to delete the directory while server is not running.
    - *buildserver.** — a set of files pertaining to the embedded HSQLDB.

- ***.BuildServer/backup*** — default directory to store backup archives created via web UI. The files in this directory are not used by TeamCity and can be safely removed if they were already copied for safekeeping.

- ***.BuildServer/lib/jdbc*** — directory that TeamCity uses to search for database drivers. Create the directory if necessary. TeamCity does not manage the files in the directory, it only scans it for .jar files that store the necessary driver.

Direct Modifications of Configuration Files

The files in the `config` directory can be edited manually (unless explicitly noted). They can even be edited without stopping the server. 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.

.dist Template Configuration Files

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

- Together with the file (suppose named `fileName`) 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 they can be updated.

XML Structure and References

If you plan to modify configuration manually please note that there are interlinking entries that link by *id*. Examples of such entries are **build configuration -> VCS roots** links and **VCS root -> project** links. All the entries of the same type must have unique ids. 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 Editions

TeamCity comes in two editions:

- **Professional edition**: does not require any server license key and has a limitation of maximum 20 build configurations configured.
- **Enterprise edition**: edition with unlimited number of build configurations.

The editions are equal in all the features besides the maximum build configuration number allowed.
The same TeamCity distribution and installation is used for both editions. You can switch to Enterprise edition by entering a license key on the **Licenses** page in the **Administration** area. All the data is preserved when edition is switched.

Each TeamCity edition comes bundled with 3 agents. More Build Agents can be added with separate licenses.

Please see Licensing Policy for licensing description.

When an Enterprise license key is removed from the server, temporary license expires or TeamCity server is upgraded to a version released out of maintenance window of the available Enterprise license, TeamCity automatically switches to the Professional mode. If build configurations number exceed 20, the server stops to start any builds and displays a warning message to all users in the web browser.

Ways to switch your server into Enterprise mode:

- buy Enterprise Server license;
- request 60 days evaluation license from TeamCity download page (If you need to extend/repeat the evaluation, please contact our sales department);
- use TeamCity EAP release (not stable, but comes bundled with 60 day nonrestrictive license);
- if you are planning to use TeamCity for an open-source projects only, we have an open-source license for you. Please refer to the details on the page.

See also:

Concepts: Role and Permission

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.

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

There are also testing frameworks that have embedded support for TeamCity. e.g. Gallio and xUnit.
See also external plugins.

Also, you can import test run XML reports of supported formats with XML Report Processing.

Custom Testing Frameworks

If there is no TeamCity support yet for your testing framework, you can report tests progress to TeamCity from the build via service messages or generate one of the supported XML reports in the build.

Also, see notes on integrating with various reporting/metric tools.

See also:

Concepts: Build State | Build Runner
User's Guide: Viewing Tests and Configuration Problems
Administrator's Guide: NUnit Support | MSTest Support | NAnt

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 Scheme and LDAP Integration sections for more details).

Each user account:

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

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

See also:

Concepts: User Group | Role and Permission | Authentication Scheme
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 as well be included into several user groups.

"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

Already Fixed In

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

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

"After" means here that

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

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

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

See also:

> **Concepts**: First Failure

Wildcards

TeamCity supports wildcards in different configuration options.

Ant-like wildcard are:

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

You can read more on Ant wildcards in the corresponding section of Ant documentation.

Examples
For a directory structure:

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

Description	Pattern	Matching files

all the files	**	`\a`
		` -\b`
		`   -\c`
		`     -file1.txt`
		`   -file2.txt`
		`   -file3.log`
		` -\d`
		`   -file4.log`
		` -file5.log`

all log files	**/*.log	`\a`
		` -\b`
		`   -file3.log`
		` -\d`
		`   -file4.log`
		` -file5.log`

all files in a/b directory incl. those in subfolders	a/b/**	`\b`
		` -\c`
		`   -file1.txt`
		` -file2.txt`
		` -file3.log`

files directly in a/b directory	a/b/*	`\b`
		` -file2.txt`
		` -file3.log`

First Failure

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

This is the build where TeamCity detected the first failure of this test for the same Build Configuration.
I.e. starting from the current build, TeamCity goes back through the build history to find out when this test failed the first time. Builds without this test are skipped, if successful test run was found, searching stops.

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

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

See also:

> **Concepts**: Already Fixed In

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 familiariaze 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 1.6 JRE.
Linux, MacOS X	TeamCity<version number>.tar.gz	Package bundled with Tomcat servlet container for Linux, MacOS X or manual installation.
J2EE container	TeamCity<version number>.war	Package for installation into an existing J2EE container.

Install TeamCity

Following the installation instructions:

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

Install Additional Build Agents

Though, the TeamCity server in .exe and .tar.gz distributions is installed with a default build agent that runs on the same machine as the server, this setup may result in degraded TeamCity web UI performance and if your builds are CPU-intensive, it is recommended to install the build agent on separate machines or ensure 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 new TeamCity server installation. For upgrade instructions, please refer to Upgrade.

The installation procedure consists of:

- Choose appropriate TeamCity distribution (.exe, .tar.gz or .war) based on details below
- Download the distribution
- Review software requirements and hardware requirements notes
- Review Licensing Policy and TeamCity Editions
- Install and configure the TeamCity server per instructions below

This page covers:

Installing the TeamCity Server

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

- Windows .exe distribution - executable which provides installation wizard for Windows platforms and allows to install server as a Windows service;
- .tar.gz distribution - archive with "portable" version suitable for all platforms;
- .war distribution - for experienced users that want to run TeamCity in separately installed Web application server.

After installation, TeamCity web UI can be accessed via web browser. Default addresses are http://localhost/ for Windows distribution and http://localhost:8111/ for tar.gz distribution.

If you cannot access TeamCity web UI after successful installation, please refer to Troubleshooting TeamCity Installation Issues section.

> The build server and one build agent will be installed by default for Windows, Linux or MacOS X. If you need more build agents, refer to the section Installing Additional Build Agents.

> By default, TeamCity uses an HSQLDB database that does not require special configuring. This database works fine for testing and evaluating the system.
> For production purposes using a standalone external database is recommended.
>
> - Setting up an External Database
> - Migrating to an External Database

Installing TeamCity via Windows installation package

For the Windows platform, run the executable file and follow the installation instructions. You have options to install the TeamCity web server and one build agent that can be run as a Windows service.

If you opted to install the services, use standard Windows `Services` applet to manage the service.
Otherwise, use standard scripts.

If you did not change the default port (80) during installation, the TeamCity web UI can be accessed via "http://localhost/" address in a web browser. Please note that 80 port 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, please refer to the Configuring TeamCity Server Startup Properties section.

Service account
Please make sure the service account has:

- write permissions for the TeamCity Data Directory,
- 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 which TeamCity server service runs, and ClearCase view owner are the same (if ClearCase integration is used).

By default, Windows service in installed under 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)

Please review software requirements before the installation.

Unpack `TeamCity<version number>.tar.gz` archive (for example, using `tar xfz TeamCity<version number>.tar.gz` command under Linux, or WinZip, WinRar or alike utility under Windows).
Please use GNU tar to unpack. (for exapmple, Solaris 10 tar is reported to truncate too long file names and may cause a `ClassNotFoundException`. Consider getting GNU tar at Solaris packages or using `gtar xfz` command)

Ensure you have JRE or JDK installed and JAVA_HOME environment variable is pointing to Java installation directory. Latest Oracle Java 1.6 update is recommended.

Starting TeamCity server

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

No matter how TeamCity is installed, TeamCity server can be started and stopped by the scripts provided in the `<TeamCity home>/bin` directory.

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

To start/stop only the TeamCity server, use `teamcity-server` script.

For example:

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

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

See below for changing the server port.

If you need to pass special properties to the server, please refer to Configuring TeamCity Server Startup Properties.

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

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

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

Installing TeamCity into Existing J2EE Container

1. Copy the downloaded `TeamCity<version number>.war` file into the web applications directory of your J2EE container under `teamCity.war` name or deploy the .war following documentation of the web server. Please make sure there is no other version of TeamCity deployed (e.g. do not preserve old TeamCity web application directory under web server applications directory).
2. Configure TeamCity logging properties by specifying `log4j.configuration` and `teamcity_logs` properties. Up-to-date values and `conf/teamcity-server-log4j.xml` file can be looked up in the `bin/teamcity-server` script available in .exe and tar.gz distributions. Sample `teamcity-server-log4j.xml` file.
3. Ensure TeamCity web application is devoted sufficient amount of memory. Please increase the sizes accordingly if you have other web applications running in the same JVM.
4. If you are deploying TeamCity to **Tomcat** container, please add `useBodyEncodingForURI="true"` attribute to the `Connector` tag for the server in `Tomcat/conf/server.xml` file.
5. If you are deploying TeamCity to **Jetty** container version >7.5.5 (including 8.x.x) please make sure system property `org.apache.jasper.compiler.disablejsr199` is set to `true`
6. Ensure servlet container is configured to unpack deployed war files. Though for most servlet containers it is the default behaviour, for some it is not (e.g. Jetty version >7.0.2) and must be explicitly configured. TeamCity can not start while packed and will prompt about this in logs and UI.
7. Configure appropriate TeamCity Data Directory to be used by TeamCity.
8. Restart the server or deploy the application via servlet container administration interface and access http://server/TeamCity-NNN/, where "TeamCity-NNN" is the name of the `war` file. You might also need to rename the file to exclude build number from the name.

TeamCity J2EE container distribution is tested to work with Tomcat 7 servlet container. (See also Supported Platforms and Environments#The TeamCity Server)

Autostart TeamCity server on Mac OS X

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

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

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

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

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

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

Using another Version of Tomcat

If you want to use another version of Tomcat web server instead of bundled one, you have the choices of whether to use .war TeamCity distribution or do Tomcat upgrade/patch for TeamCity installed from .exe or .tar.gz distributions.
For the latter, you might want to:

- backup current TeamCity home
- delete/move out the directories from TeamCity home which are also present in Tomcat distribution
- unpack Tomcat distribution into 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 default Tomcat `conf` directory and replace it with TeamCity-provided one
    - delete default Tomcat `webapps/ROOT` directory and replace it with TeamCity-provided one

Installation Configuration

Troubleshooting TeamCity Installation

Upon successful installation, 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 TeamCity web UI cannot be accessed, please check:

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

One of the most common issues with the server installation is using a port that is already used by another program. See
more on changing the default port.

Changing Server Port

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

Use the following instructions to change the port if you use .tar.gz distribution
If another application uses the same port that TeamCity server, 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's port, in the `<TeamCity Home>/conf/server.xml` file, change the port number in the HTTP/1.1 connector (here the port number is 8111):

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

To apply changes, you should restart the server.

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

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

Java Installation

TeamCity server is a web application that runs in an J2EE application server (a JVM application). A JVM application requires a JRE installation to run.

TeamCity (both server and agent) requires JRE 1.6 (or later) to operate. Using latest Oracle JSDK 1.6 is recommended (download page). Please also note that TeamCity agent needs JDK (not JRE) to operate properly.

The necessary steps to prepare Java installation depends on the distribution used.

- Windows Installer (.exe) has JRE bundled (in `jre` directory). If you need to update the JRE used by the installation:
  - if you run the server from console refer to instructions for `.tar.gz` distribution below.
  - if you run as Windows service and want to upgrade JRE to newer 32 bit version, you can replace `<TeamCity home>\jre` with JRE from the newer installation (just install JRE per installation instructions and copy the content of the resulting directory to replace the content of the existing "jre" directory).
  - if you run as Windows service and want to upgrade JRE to 64 bit version, you will need to replace `<TeamCity home>\jre` with appropriate JRE and also replace/update bundled Tomcat Windows binaries: copy/overwrite content of `bin\x64` directory to `bin`.
- `.war` distribution depends on the application server used. Please refer to the manual of the server.
- To use `.tar.gz` distribution and `teamcity-server` or `runAll` scripts you need to have JRE installed either in `<TeamCity home>\jre` or into another location. If another location is used, ensure there is no `<TeamCity home>\jre` directory present and one of the environment variables is defined: `JRE_HOME` (pointing to home directory of installed JRE), or `JAVA_HOME` (pointing to home directory of installed JSDK).

Setting Up Memory settings for TeamCity Server

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

- For initial use of TeamCity for production purposes (assuming 32 bit JVM) minimum recommended settings are: `-Xmx750m -XX:MaxPermSize=270m`. If slowness or OutOfMemory error occurs, please increase the settings to `-Xmx1300m -XX:MaxPermSize=270m`.
- Maximum settings that you will ever probably need are (x64 JVM should be used): `-Xmx4g -XX:MaxPermSize=270m`. These settings will suit for an installation with more than a hundred of agents and thousands of build configurations.
- If you run TeamCity via `runAll` or `teamcity-server` scripts or via Windows service installed, the default settings used are: 512 Mb for the heap and 150 Mb for the PermGen.

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

Tips:

- 32 bit JVM can use up to 1.3Gb memory. If more memory is necessary, 64 bit JVM should be used assigning not less than 2.5Gb. It's highly unlikely that you will need to dedicate more than 4Gb of memory to the TeamCity process.
- A rule of thumb is that 64 bit JVM should be assigned twice as much memory as 32 bit for the same application. If you switch to 64 bit JVM please make sure you adjust the memory settings (both `-Xmx` and `-XX:MaxPermSize`) accordingly. It does not make sense to switch to 64 bit if you dedicate less than double amount of memory to the application.

The recommended approach is to start with initial settings and increase them whenever OutOfMemory error occurs (see also TW-13452).
You can also monitor for the percentage of used memory at the **Administration** | **Diagnostics** page. If the server uses more then 80% of memory consistently without drops for tens of minutes, that is probably a sign to increase the memory values by another 20%.

Using 64 bit Java to Run TeamCity Server

TeamCity can run under both 32 and 64 bit JVM.
It is recommended to use 32 bit JVM unless you need to dedicate more than 1.3Gb of memory to the TeamCity process.

If you choose to use x64 JVM please note that the memory usage is almost doubled when switching from 32 to 64 bit JVM, so please make sure you specify at least twice as much memory as for 32 bit JVM, see Setting Up Memory settings for TeamCity Server.

If you run TeamCity as a service and switch to x64 bit, you will also need to use x64 Tomcat executables, see more.

Configuring the TeamCity Server

✓

- If you have a lot of projects or build configurations, we recommend you avoid using the **Default agent** in order to free up the TeamCity server resources. The TeamCity Administrator can disable the default agent on the **Agents** page of the web UI.
- When changing the TeamCity data directory or database make sure they do not get out of sync.

Configuring TeamCity Data Directory

The default placement of the TeamCity data directory can be changed. See corresponding section: TeamCity Data Directory for details.

Editing Server Configuration

After successful server start, any TeamCity page request will redirect to prompt for the server administrator username and password. Please make sure that no one can access the server pages until the administrator account is setup.

After administration account setup you may begin to create Project and Build Configurations in the TeamCity server. You may also want to configure the following settings in the Server Administration section:

- Server URL
- Email server address and settings
- Jabber server address and settings

See also:

> **Installation and Upgrade**: Setting up and Running Additional Build Agents

Setting up and Running Additional Build Agents

This page covers:

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 Agent's JVM parameters to include `-server` option.

Installing Additional Build Agents

Before the installation, please review Known Issues#Conflicting Software section.

Necessary OS and environment permissions

Please note that in order to run a TeamCity build agent, the user account under which the Agent is running should have the correct privileges, as described below.

Network

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

- address of the network interfaces on the agent machine

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

Common

- agent process (java) should be able to open outbound HTTP connections to the server address (the same address you use in the browser to view TeamCity UI) and accept inbound HTTP connections from the server to the port specified as "ownPort" property in "<TeamCity agent home>/conf/buildAgent.properties" file (9090 by default). Please ensure that any firewalls installed on agent, server machine or in the network and network configuration comply with these requirements.
- have full permissions (read/write/delete) to the following directories: `<agent home>` (necessary for automatic agent upgrade), `<agent work>`, and `<agent temp>`.
- launch processes (to run builds).

Windows

- Log on as a service (to run as Windows service)
- Start/Stop service (to run as Windows service, necessary for agent upgrade to work, see also Microsoft KB article)
- Debug programs (for take process dump functionality to work)
- Reboot the machine (for agent reboot functionality to work)

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

Linux

- user should be able to run `shutdown` command (for agent machine reboot functionality and machine shutdown functionality when running in Amazon EC2)

Build-related Permissions
The build process is launched by TeamCity agent and thus shares the environment and is executed under the same OS user that TeamCity agent runs under. Please ensure that TeamCity agent is configured accordingly.
See Known Issues for related Windows Service Limitations.

Server-Agent Data Transfers

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

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

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

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

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

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

Installing Procedure

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

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

After installation, please configure the agent specifying its name and address of TeamCity server in its `conf/buildAgent.properties` file. Then start the agent.

When the newly installed agent connects to the server for the first time, it appears on the `Unauthorized agents` tab under `Agents`, where administrators can authorize it. Please note that the tab is only visible for administrators/users with appropriate permission.

> Agents will not run builds until they are authorized in the TeamCity web UI. The agent running on the same computer as the server is authorized by default.

If the agent does not seem to run correctly, please check the agent logs.

Installing via Java Web Start

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

> You can install the build agent Windows service during the installation process or manually.

Installing via a MS Windows installer

1. Navigate to the **Agents** tab in the TeamCity web UI.
2. Click the "Install Build Agents" link and then click **MS Windows Installer** link to download the installer.
3. Run the `agentInstaller.exe` Windows Installer and follow the installation instructions.

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

Installing via ZIP File

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

> On Windows you may also want to install the build agent windows service instead of manual agent startup.

Installing via Agent Push

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

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

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

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

Platform	Prerequisites

Unix	Installed JDK(JRE) 1.6+ required. JVM should be reachable with JAVA_HOME(JRE_HOME) environment variables or be in paths. Also required 'unzip' utility and either 'wget' or 'curl'.
Windows	• Installed JDK/JRE 1.6+ is required. • `Sysinternals psexec.exe` on TeamCity server required. It has to be accessible in paths. You can install it at **Administration** | **Tools** page. Note, that PsExec applies additional requirements to remote Windows host (for example, administrative share on remote host must be accessible). Read more about PsExec.

Installation Procedure

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

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

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

> ✓ You can use Agent Push presets in Amazon EC2 Cloud profile settings to automatically install build agent to started cloud instance.

Starting the Build Agent

To start the agent manually, run the following script:

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

> ⚠ If you're running build agent on MacOS X and you're going to run Inspection builds, you may need to use the **StartupItemContext** utility:
>
> ```
> sudo /usr/libexec/StartupItemContext agent.sh start
> ```

To configure agent to be **started automatically**, see corresponding sections:
Windows
Mac OS X
Linux: configure daemon process with `agent.sh start` command to start it and `agent.sh stop` command to stop it.

Stopping the Build Agent

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

Use `stop` to request stopping after current build finished.
Use `stop force` to request immediate stop (if a build is running on the agent, it will be stopped abruptly (canceled))
Under Linux, you have one more option top use: `stop kill` to kill the agent process.

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

Automatic Agent Start under Windows

To run agent automatically on machine boot under Windows you can either setup agent to be run as Windows service or use another way of automatic process start.
Using Windows service approach is the easiest way, but Windows applies some constraints to the processes run in this way.
TeamCity agent can work OK under Windows service (provided all the requirements are met), but is often not the case for the build processes that you will configure to be run on the agent.

That is why it is advised to run TeamCity agent as use Windows service only if all the build scripts support this.
Otherwise, it is adviced to use alternative ways to start TeamCity agent automatically.
One of the ways is to configure automatic user logon on Windows start and then configure TeamCity agent start (via `agent.bat start`) on user logon.

Build Agent as a Windows Service

In Windows, you may want to use the build agent Windows service to allow the build agent to run without any user logged on.
If you use Windows agent installer you have an option to install the service in the installation wizard.

> **Service system account**
> To run builds, the build agent should be started under a user with enough permissions for performing a build and managing the service. By default, Windows service in started under SYSTEM account. To change it, use the standard Windows Services applet (Control Panel|Administrative Tools|Services) and change the user for `TeamCity Build Agent` service.

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

To install the service:

1. Make sure there is no **TeamCity Build Agent Service <build number>** service already installed, if installed, uninstall the agent.
2. Check `wrapper.java.command` property in `<agent home>\launcher\conf\wrapper.conf` file to contain valid path to Java executable in the JDK installation directory. You can use `wrapper.java.command=../jre/bin/java` for agent installed with Windows distribution.
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 Windows `net.exe` utility to manage the service once it is installed.
For example (assuming the default service name):

```
net start TCBuildAgent
```

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

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

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

Using LaunchDaemons Startup Files on MacOSx

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

To use LaunchDaemons `plist` file:

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

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

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

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

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

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

1. To stop build agent, run the following command:

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

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

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

Configuring Java

TeamCity Agent is a Java application and it requires JDK version 1.6 or later to work. Oracle JDK is recommended.
(Windows) .exe TeamCity distribution comes with appropriate Java bundled. If you run previous version of TeamCity agent you might need to repeat agent installation to update used Java.

Using x32 bit JDK is recommended. If you do not have Java builds, you may install JRE instead of JDK.
Using of x64 bit Java is possible, but you might need to double -Xmx and -XX:MaxPermSize memory values for the main agent process (see Configuring Build Agent Startup Properties and alike section for the server).

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

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

You can download Java installation from Oracle, select Java SE, JDK, version 1.6, 32 bit.

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. This may happen when upgrading to a newer TeamCity version, which doesn't support an old Java version anymore. To update Java on agents you can do one of the following:

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

Installing Several Build Agents on the Same Machine

TeamCity treats equally all agents no matter if they are installed on the same or on different machines. However, before installing several TeamCity build agents on the same machine, please consider the following:

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

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

- The agents are installed in the separate directories.
- The agents have distinctive `workDir` and `tempDir` directories in `buildAgent.properties` file.

- Values for `name` and `ownPort` properties of `buildAgent.properties` are unique.
- No builds running on the agents have absolute checkout directory specified.

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

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

If you want to install additional agents as services under Windows, do not opt for service installation during installer wizard or install manually (see also a feature request), then
modify the `<agent>\launcher\conf\wrapper.conf` file so that `wrapper.console.title`, `wrapper.ntservice.name`, `wrapper.ntservice.displayname` and `wrapper.ntservice.description` properties have unique values within the computer. Then run `<agent>\bin\service.install.bat` script under user with sufficient privileges to register the new agent service.
See above for the service start/stop instructions.

See also:

Concepts: Build Agent

Build Agent Configuration

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

The file is a Java properties file.
A quick guide is:

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

This is an example of the file:

```
## The address of the TeamCity server. The same as is used to open TeamCity web interface in the
browser.
serverUrl=http://localhost:8111/

## The unique name of the agent used to identify this agent on the TeamCity server
name=Default agent

## Container directory to create default checkout directories for the build configurations.
workDir=../work

## Container directory for the temporary directories.
## Please note that the directory may be cleaned between the builds.
tempDir=../temp

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

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

> ⚠ Please make sure that the file is writable for the build agent process itself. For example the file is updated to store its authorization token that is generated on the server-side.

If "name" property is not specified, the server will generate build agent name automatically. By default, this name would be created from the build agent's host name.

The file can be edited while the agent is running: the agent detects the change and restarts loading the new changes automatically.

See also:

> **Concepts**: Build Agent
> **Administrator's Guide**: Predefined Build Parameters | Configuring Agent Requirements | Configuring Build Parameters

Setting Up TeamCity for Amazon EC2

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

For integrations with other cloud solutions, see Cloud-VMWare plugin and Implementing Cloud support.

General Description

Once a cloud profile is configured in TeamCity with one or several images, TeamCity does a test start for all the new images to learn about the agents configured on them.
Once agents are connected, TeamCity stores their parameters to be able to correctly process build configurations-to-agents compatibility.

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

Once an agent is connected from a cloud instance started by TeamCity, it is automatically authorized (provided there are available agent licenses). After that the agent is processed as a regular agent.
If running timeout is configured on the cloud profile and it is reached, the instance is terminated. **Since TeamCity 7.1**, if EBS-based instance id is specified in the images list, the instance is stopped instead. On instance terminating/stopping, its disconnected agent is removed from authorized agents list and is deleted from the system.

Configuration

Understanding Amazon EC2 and ability to perform EC2 tasks is a prerequisite for configuring and using TeamCity Amazon EC2 integration.

Basic TeamCity EC2 setup involves:

- preparing Amazon EC2 image (AMI) with installed TeamCity agent
- configuring EC2 integration on TeamCity server

> ⚠ Please note that the number of EC2 agents is limited by the total number of agent licenses you have in TeamCity.

Please ensure that the server URL specified on **Global Settings** page in **Administration** area is correct since agents will use it to connect to the server.

Preparing AMI 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 Amazon AMI 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 AMI preparation steps:

1. Choose one of existing AMIs.
2. Start AMI.
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).
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)
6. Make AMI from the running instance.
7. Configure TeamCity EC2 support on TeamCity server.

Agent auto-upgrade Note
TeamCity agent auto-upgrades whenever distribution of agent plugins on the server changes (e.g. after TeamCity upgrade). If you want to cut agent startup time, you might want to re-bundle the agent AMI after agent plugins has been auto-updated.

Estimating EC2 Costs

Usual Amazon EC2 pricing applies. Please note that Amazon charges can depend on the specific configuration implemented to deploy TeamCity. We advice you to check your configuration and Amazon account data regularly in order to discover and prevent unexpected charges as early as possible.

Please note that traffic volumes and necessary server and agent machines characteristics depend a big deal on the TeamCity setup and nature of the builds run. See also How To...#Estimate hardware requirements for TeamCity.

Traffic Estimate

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

- If TeamCity server is not located within the same EC2 region or availability zone that is configured in TeamCity EC2 settings for agents, traffic between the server and agent is subject to usual Amazon EC2 external traffic charges.
- When estimating traffic please bear in mind that there are lots types of traffic related to TeamCity (see a non-complete list below).

External connections originated by server:

- VCS servers
- Email and Jabber servers
- Maven repositories

Internal connections originated by server:

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

External connections originated by agent:

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

Internal connections originated by agent:

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

Usual connections served by the server:

- web browsers
- IDE plugins

Uptime Costs

As Amazon rounds machine uptime to the nearest full hour, please adjust timeout setting on the EC2 image setting on TeamCity cloud integration settings according to your usual builds length.

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

Installing Additional Plugins

For a list of available plugins, see plugins list

Steps to install TeamCity plugin:

1. Shutdown TeamCity server
2. Copy the zip archive with the plugin into `<TeamCity Data Directory>/plugins` directory.
3. Start TeamCity server: the plugin files will be unpacked and processed automatically.

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

If the plugin uses obsolete, pre-4.0 packaging:

Installing Jar-packaged Server-side Plugin

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

Installing Agent-side Plugin (pre-4.0 packaging)

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

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

Upgrade Notes

Changes from 7.0.x to 7.1

Windows service configuration
Since version 7.1, TeamCity uses it's 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.

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 depricated 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.

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 manualy.
- 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 anohter 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 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 to use Sybase as 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-intance (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 form `.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 is they need updating to use new text patterns.

REST API Changes
Several objects got additional attributes and sub-elements (e.g. "startDate" in reference to a build, "personal" in a change). Please check that your parsing code still works.

Cleaning Non-default Checkout Directories
In previous releases, if you have specified build checkout directory explicitly using absolute path, TeamCity would not clean the content of the directory to free space on the disk.
This is no longer the case.
So if you have absolute path specified for the checkout directory and you need the directory to be present on agent for other build or for the machine environment, please set `system.teamcity.build.checkoutDir.expireHours` property to "never" and re-run the build. Please take into account that using custom checkout directory is not recommended.

If you are using one of system.teamcity.build.checkoutDir.expireHours properties and it is set to "never" to prevent the checkout directory from automatic deletion, the directory might be deleted once after TeamCity upgrade. Running the build in the build configuration once after the upgrade (and within 8 days from the previous build) will ensure that the directory preserves the "protected" behavior and will not be automatically removed by TeamCity.

Free disk space
This release exposes Free disk space feature in UI that was earlier only available via setting build configuration properties.
While the old properties still work and take precedence, it is highly recommended to remove them and specify the value via "Disk Space" build feature instead. Future TeamCity versions might stop to consider the properties specified manually.

Command line runner
`@echo off` which turns off command-echoing is added to scripts provided by "Custom script" runner parameter. To enable command-echoing add `@echo on` to the script.

Windows Tray Notifier
You will need to upgrade windows tray notifier by uninstalling it and installing it again. Unfortunately, auto-upgrade will not work due to issues in old version of Windows Tray Notifier.

Maven runner

- In earlier TeamCity versions Maven was executed by invoking the 'mvn' shell script. You could specify some parameters in MAVEN_OPTS and some in UI. Maven build runner created its own MAVEN_OPT by concatenating these two (`%MAVEN_OPTS%+jvmArgs`). In this case, if some parameter was specified twice - in MAVEN_OPTS and in UI, only the one specified in MAVEN_OPTS was effective. Starting with TeamCity 6.5 Maven runner forms direct java command. While this approach solves many different problems, it also means that MAVEN_OPTS isn't effective anymore and all JVM command line parameters should be specified in build runner settings instead of MAVEN_OPTS.
- Those who had to manually setup surefire XML reporting for Maven release builds in TeamCity 6.0.x because otherwise tests weren't reported, now can forget about that. Since TeamCity 6.5 surefire tests run by release:prepare or release:perform goals are automatically detected. So don't forget to switch surefire XML reporting off in the build configuration settings to avoid double-reporting!

Email sending settings
Please check email sending settings are working correctly after upgrade (via Test connection on Administration > Server Configuration > EMail Notifier). If no authentication is needed, make sure login and password fields are blank. Non-blank fields may cause email sending errors if SMTP server is not expecting authentication requests.

XML Report Processing
Tests from Ant JUnit XML reports can be reported twice (see TW-19058), as we no longer automatically ignore TESTS-xxx.xml report.
To workaround this avoid using *.xml mask and specify more concrete rules like TEST-*.xml or alike that will not match report with name starting with "TESTS-"

Open API Changes
Several return types have changes in TeamCity open API, so plugins might need recompilation against new TeamCity version to continue working.
Also, some API was deprecated and will be discontinued in later releases. It is recommended to update plugins not to use deprecated API.
See also Open API Changes

Changes from 6.0.2 to 6.0.3

No noteworthy changes

Changes from 6.0.1 to 6.0.2

Maven and XML Test Reporting Load CPU on Agent
If you use Maven or XML test reporter and your build is CPU-intensive, you might find important the known issue. Patch is available, fixed in the following updates.

Changes from 6.0 to 6.0.1

No noteworthy changes

Changes from 5.1.x to 6.0

Visual Studio Add-in and Perforce
There is critical bug in TeamCity 6.0 VS Add-in when Perforce is enabled. This can cause Visual Studio hangs and crashes. The fixed add-in version is available. (related issue). The issue is fixed in TeamCity 6.0.1.

TFS checkout on agent
TFS checkout on agent might refuse to work with errors. Patch is available, see the comment. Related issue. The issue is fixed in TeamCity 6.0.1.

Error Changing Priority class
You may encounter a browser error while changing priority number of a priority class. A patch is available in a related issue. The issue is fixed in TeamCity 6.0.1.

IntelliJ IDEA Compatibility

IntelliJ IDEA 6 and 7 are no longer supported in TeamCity plugin for IntelliJ IDEA.

Also, if you plan to upgrade to IntelliJ IDEA X (or other JetBrains IDE) please review this known issue.

Build Failure Notifications
TeamCity 6.0 differentiates between build failure occurred while running a build script and one occurred while preparing for the build. The errors occurring in the latter case are called "failed to start" errors and are hidden by default from web UI (see "Show canceled and failed to start builds" option on Build Configuration page).

Since TeamCity 6.0, there is a separate notification rule "The build fails to start" which applies for "failed to start" builds. All the rest build failure notifications relate to build script-related failures.

Please note that on upgrade, all users who had "The build fails" notification on, will automatically get "The build fails to start" option to preserve old behavior.

Properties Changes
`teamcity.build.workingDir` property should no longer be used in non-runner settings. For backward compatibility, the property is supported in non-runner settings and is resolved to the working directory of the first defined build step.

Swabra and Build Queue Priorities Plugins are Bundled
If you have installed the plugins previously, please remove them (typically form `.BuildServer/plugins`) before starting upgraded TeamCity version.

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

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

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

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

.Net Framework Properties are Reported as Configuration Parameters
In previous TeamCity versions, installed .Net Frameworks, Visual Studios and Mono were reported as System Properties of the build agents. This made the properties available in the build script.
In order to reduce number of TeamCity-specific properties pushed into the build scripts, the values are now reported via Configuration Parameters (that is, without "system." prefix) and are not available in the build script by default. They still be used in the Build Configuration settings via %-references by their previous names, just without "system." prefix.

Ipr runner is deprecated in favor of IntelliJ IDEA Project runner
Runner for IntelliJ IDEA projects was completely rewritten. It is not named "IntelliJ IDEA Project" runner. Previously available Ipr runner is also preserved but is marked as deprecated and will be removed in one of the further major releases of TeamCity. It is highly recommended to migrate your existing build configurations to the new runner.
Please note that the new runner uses different approach to run tests: you need to have a shared Run Configuration created in IntelliJ IDEA and reference it in the runner settings.

Cleanup for Inspection and Duplicates data
Starting from 6.0 Inspection and Duplicates reports for the builds are cleaned when build is cleaned from history, not when build's artifacts are cleaned as it used to be.

Inspection and Duplicates runners require Java 1.6
"Inspections" and "Duplicates (Java)" runners now require Java JDK 1.6. Please ensure Java 1.6 is installed on relevant agents and check it is specified in the "JDK home path" setting of the runners.

XML Report Validation
If you had invalid settings of "XML Report Processing" section of the build runners, you might find the Build Configurations reporting "Report paths must be specified" messages upon upgrade. In this case, please go to the runner settings and correct the configuration. (related issue)

Open API Changes
See Open API Changes
Several jars in `devPackage` were reordered, some moved under `runtime` subdirectory. Please update your plugin projects to accommodate for these changes.

REST API Changes
Several objects got additional attributes and sub-elements. Please check that your parsing code still works.

Perforce Clean Checkout

All builds using Perforce checkout will do a clean checkout after server upgrade. Please note that this can impose a high load on the server in the first hours after upgrade and server can be unresponsive while many builds are in "transferring sources" stage.

Changes from 5.1.2 to 5.1.3

Path to executable in Command line runner
The bug was fully fixed. The behavior is the same as in pre-5.1 builds.

Changes from 5.1.1 to 5.1.2

Jabber notification sending errors are displayed in web UI for administrators again (these messages were disabled in 5.1.1). If you do not use Jabber notifications, please pause the Jabber notifier on the Jabber settings server settings page.

Changes from 5.1 to 5.1.1

Path to executable in Command line runner
The bug was partly fixed. The behavior is the same as in pre-5.1 builds except for the case when you have the working directory specified and have the script in both checkout and working directory. The script from the working directory is used.

Path to script file in Solution runner and MSBuild runner
The bug was fixed. The behavior is the same as in pre-5.1 builds.

Changes from 5.0.3 to 5.1

> ⓘ If you plan to upgrade from version 3.1.x to 5.1, you will need to modify some dtd files in `<TeamCity Data Directory>/config` before upgrade, read more in the issue: TW-11813

> ⓘ NCover 3 support may not work. See TW-11680

Notification templates change
Since 5.1, TeamCity uses new template engine (Freemarker) to generate notification messages. New default templates are supplied and customizations to the templates made prior to upgrading are no longer effective.

If you customized notification templates prior to this upgrade, please review the new notification templates and make changes to them if necessary. Old notification templates are copied into `<TeamCity Data Directory>/config/_trash/_notifications` directory. Hope, you will enjoy the new templates and new extended customization capabilities.

External database drivers location
JDBC drivers can now be placed into `<TeamCity Data Directory>/lib/jdbc` directory instead of `WEB-INF/lib`. It is recommended to use the new location. See details at Setting up an External Database#Database Driver Installation.

PostgresSQL 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 PostrgeSQL 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 then regular one. This time depends on the size of your builds history, average number of tests in a build and the server hardware.

Users re-login will be forced after upgrade
Upon upgrade, all users will be automatically logged off and will need to re-login in their browsers to TeamCity web UI. After the first login since upgrade, **Remember me** functionality will work as usual.

Previous IntelliJ IDEA versions support
IntelliJ IDEA plugin in this release is no longer compatible with IntelliJ IDEA 6.x versions. Supported IDEA versions are 7.0.3 and 8.0.

Using VCS revisions in the build
`build.vcs.number.N` system properties are replaced with `build.vcs.number.<escaped VCS root name>` properties (or just `build.vcs.number` if there is only one root). If you used the properties in the build script you should update the usages manually or switch compatibility mode on. References to the properties in the build configuration settings are updated automatically. Corresponding environment variable has been affected too.
Read more.

Test suite
Due to the fact that TeamCity started to handle tests suites, the tests with suite name defined will be treated as new tests (thus, test history can start from scratch for these tests.)

Artifact dependency pattern
Artifact dependencies patterns now support Ant-like wildcards.
If you relied on "**" pattern to match directory names, please adjust your pattern to use "/**" instead of single "*".
If you relied on the "**" pattern to download only the files without extension, please update your pattern to use "**." for that.

Downloading of artifacts with help of Ivy
If you downloaded artifacts from the build scripts (like Ant build.xml) with help of Ivy tasks you should modify your ivyconf.xml file and remove all

statuses from there except "integration". You can take the ivyconf.xml file from the following page as reference: http://www.jetbrains.net/confluence/display/TCD4/Configuring+Dependencies

Browser caches (IE)
To force Internet Explorer to use updated icons (i.e. for the Run button) you may need to force page reload (Ctrl+Shift+R) or delete "Temporary Internet Files".

Changes from 3.1.1 to 3.1.2

No noteworthy changes.

Changes from 3.1 to 3.1.1

No noteworthy changes.

Changes from 3.0.1 to 3.1

Guest User and Agent Details

Starting from version 3.1, the Guest user does not have access to the agent details page. This has been done to reduce exposing potentially sensitive information regarding the agents' environment. In the Enterprise Edition, the Guest user's roles can be edited at the **Users and Groups** page to provide the needed level of permission.

StarTeam Support

Working Folders in Use

Since version 3.1 when checking out files from a StarTeam repository TeamCity builds directory structure on the base of the working folder names, not just folder names as it was in earlier versions. So if you are satisfied with the way TeamCity worked with StarTeam folders in version 3.0, ensure the working folders' names are equal to the corresponding folder names (which is so by default).

Also note, that although StarTeam allows using absolute paths as working folders, TeamCity supports relative paths only and doesn't detect absolute paths presence. So be careful and review your configuration.

StarTeam URL Parser Fixed

In version 3.0 a user must have followed a wrong URL scheme. It was like *starteam://server:49201/project/view/rootFolder/subfolder/...* and didn't work when user tried to refer a non-default view. In version 3.1 the native StarTeam URL parser is utilized. This means you now don't have to specify the root folder in the URL, and the previous example should look like *starteam://server:49201/project/view/subfolder/...*

Changes from 3.0 to 3.0.1

Linux Agent Upgrade

- Due to an issue with Agent upgrade under Linux, Agent auxiliary Launcher processes may have been left running after agent upgrades. Versions 3.0.1 and up fix the issue. To get rid of the stale running processes, after automatic agent upgrade, please stop the agent (via `agent.sh kill` command) and kill any running `java jetbrains.buildServer.agent.Launcher` processes and start the agent again.

Changes from 2.x to 3.0

Incompatible changes

Please note that TeamCity 3.0 introduces several changes incompatible with TeamCity 2.x:

- **build.working.dir** system property is renamed to **teamcity.build.checkoutDir**. If you use the property in you 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 then 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:

> **Concepts**: TeamCity Editions
> **Administrator's Guide**: Licensing Policy

Upgrade

> ⛔ TeamCity doesn't support downgrade. It is strongly recommended to back up your data before any upgrade.

Before upgrading TeamCity:

1. Check your license
2. Download new TeamCity version
3. Carefully review the Upgrade Notes

To upgrade the server:

1. Back up 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 checking it's functioning in your environment before upgrading the main one.

Licensing

Before upgrading please make sure the maintenance period of your licenses is not yet elapsed (use **Administration | Licenses** TeamCity server web UI page to list your license keys). The licenses are valid only for the versions of TeamCity with the effective release date within the maintenance period. See the effective release date at the page.

Please note that TeamCity versions 5.0 and above use licensing policy different from that of previous TeamCity versions. Please review the Licensing Policy page and the Licensing and Upgrade section on the official site.
If you are evaluating the 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, please contact JetBrains sales department.

Upgrading TeamCity Server

TeamCity supports upgrades from any of the previous versions to the current one. Downgrades are not supported unless specifically noted.
On upgrade, all the configuration settings and other data are preserved unless noted in Upgrade Notes.

Important note on TeamCity data structure upgrade
TeamCity 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 complete the upgrade, the data is updated to a newer format.
Once the data is converted, downgrade 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 have a backup of the data before upgrading TeamCity.
- Both database and the data directory should be upgraded simultaneously. Ensure that during the first start of the newer server it uses correct TeamCity Data Directory that in its turn has correct database configured in the `<TeamCity Data Directory>\config\database.properties` file.

Upgrading Using Windows Installer

The following procedure applies only if you upgrade your TeamCity instance to version 6.0 or higher. If you are upgrading to another TeamCity version, please refer to corresponding documentation. The upgrade procedure for earlier TeamCity versions is different.

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

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

Manual Upgrading using .tar.gz or .war Distributions

1. Create a backup. Ensure you have configuration files and database data backed up.
2. Remove old installation files (either the entire TeamCity directory or `[TOMCAT_HOME]/webapps/TeamCity/*` if you are installing from a **war** file). It's advised to backup the directory beforehand.
3. Unpack the new archive where TeamCity was previously installed.
4. If you use Tomcat server (your own or bundled in .tar.gz TeamCity distribution), it is recommended to delete content of the `work` directory. Please note that this may affect other web applications deployed into the same web server.
5. Make sure you have the external database driver installed (this applies only if you use external database).
6. If you use custom plugins that do not reside in <TeamCity Data Directory>, install them.
7. Specify additional TeamCity server startup properties, if required.
8. Start up the TeamCity server.
9. Review the TeamCity Maintenance Mode page to make sure there are no problems encountered, and confirm the upgrade by clicking corresponding button. Only after that all the configuration data and database scheme are updated by TeamCity converters.

Upgrading Build Agents

- Automatic Build Agent Upgrading
- Upgrading the Build Agent to 4.0
  - Upgrading Build Agent Launcher
  - Upgrading the Build Agent Windows Service Wrapper

Automatic Build Agent Upgrading

On starting newer TeamCity server, TeamCity agents connected to the server are updated automatically. The agent (`agent.bat`, `agent.sh`, or agent service) will download the latest agent upgrade from the TeamCity server. When the download is complete and the agent is idle, it will start the upgrade process (the agent is stopped, the agent files are updated, and agent is restarted). This process may take several minutes depending on the agent hardware and network bandwidth. **Do not interrupt the upgrade process**, as doing so may cause the upgrade to fail and you will need to manually reinstall the agent.

If you see that an agent is identified as "Agent disconnected (Will upgrade)" in the TeamCity web UI, do not close the agent console or restart the agent process, but wait for several minutes.

Various console windows can open and close during the agent upgrade. Please be patient and do not interrupt the process until the agent upgrade is finished.

Upgrading Build Agents Manually

All connected agents upgrade automatically, provided they are correctly installed, so manual upgrade is not necessary.

If you need to upgrade agent manually, you can follow the steps below:

As TeamCity agent does not hold any unique information, the easiest way to upgrade an agent if to

- back up `<Agent Home>/conf/buildAgent.properties` file.
- uninstall/delete existing agent.
- install the new agent version.
- restore previously saved `buildAgent.properties` file to the same location.
- start the agent.

If you need to preserve all the agent data (e.g. to eliminate clean checkouts after the upgrade), you can:

- stop the agent.
- delete all the directories in the agent installation present in the agent .zip distribution except `conf`.
- unpack the .zip distribution to the agent installation directory, skipping the "conf" directory.
- start the agent.

In the latter case if you run agent under Windows using service, you can also need to upgrade Windows service as described below.

Upgrading the Build Agent Windows Service Wrapper

Upgrading from TeamCity version 1.x

Version 2.0 of TeamCity migrated to new way of managing Windows service (service wrapper) for the build agent: Java Service Wrapper library.

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

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

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

To use new service wrapper you should uninstall old version of the agent (with **Control Panel | Add/Remove Programs**) and then install a new one.

If you customized the user under which the service is started, do not forget to customize it in the newly installed service.

Upgrading from TeamCity version 2.x

If the service wrapper needs an update, the new version is downloaded into `the <agent>/launcher.latest` folder, however the changes are not applied automatically.

To upgrade the service wrapper manually, do the following:

1. Ensure the `<agent>/launcher.latest` folder exists.
2. Stop the service using `<agent>\bin\service.stop.bat`.
3. Uninstall the service using `service.uninstall.bat`.
4. Backup `<agent>/launcher/conf/wrapper.conf` file.
5. Delete `<agent>/launcher`.
6. Rename `<agent>/launcher.latest` to `<agent>/launcher`.
7. Edit `<agent>/launcher/conf/wrapper.conf` file. Check that the `'wrapper.java.command'` property points to the `java.exe` file. Leave it blank to use registry to lookup for java. Leave 'java.exe' to lookup `java.exe` in PATH. For a standalone agent the service value should be `../jre/bin/java`, for and agent installation on the server the value should be `../../jre/bin/java`. The backup version of `wrapper.conf` file may be used.
8. Install the service using `<agent>\bin\service.install.bat`.
9. Make sure the service is running under the proper user account. Please note that using SYSTEM can result in failing builds which use MSBuild/Sln2005 configurations.

10. Start the service using `<agent>\bin\service.start.bat`.

> ⚠ This procedure is applicable ONLY for an agent running with *new* service wrapper. Make sure you are not running the **agentd** service.

See also:

Concepts: TeamCity Data Directory
Administrator's Guide: TeamCity Maintenance Mode | TeamCity Data Backup

TeamCity Maintenance Mode

If on TeamCity startup you see the TeamCity Maintenance page, that means the TeamCity instance requires technical maintenance which for security reasons should be performed by **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 data to the newer format.
If you do **not** have access to the computer where TeamCity is installed, please inform your system administrator that TeamCity requires technical maintenance.

If you are TeamCity system administrator, to confirm that enter the *Maintenance Authentication Token* into corresponding field on this page. This token can be found in the `teamcity-server.log` file under `<TeamCity home>/logs`. Refer to TeamCity Server Logs for more information about TeamCity server logs.

After you have provided this token, you can review the details on what kind of maintenance is required. The need in technical maintenance may be caused, for example, by one of the following:

- TeamCity Data Upgrade
- TeamCity Data Format is Newer
- TeamCity Startup Error

TeamCity Data Upgrade

When upgrading your TeamCity instance, the newly installed version of TeamCity checks if the TeamCity data directory and database use 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 will need to return to earlier TeamCity version, you will be able to do so only by restoring the data from corresponding backup. Please refer to the TeamCity Data Backup section for instructions.
When you are sure you have your data backed up, click **Upgrade**.

> ✓ If you are upgrading from TeamCity 6.0(or higher), this page contains an option to perform backup automatically.

TeamCity Data Format is Newer

TeamCity has detected that the data format corresponds to more recent TeamCity version than you try to run.
Since downgrade is not supported, TeamCity cannot start until you provide the data that matches the format of the TeamCity version you want to run. To do so, please restore the required data from backup. Refer to the Restoring TeamCity Data from Backup page for the instructions.

TeamCity Startup Error

If on TeamCity startup a critical error was encountered, this page will display the error message.
Please, fix the error cause and restart the TeamCity server.

See also:

> **Installation and Upgrade**: Upgrade
> **Administrator's Guide**: TeamCity Data Backup

Setting up an External Database

By default, TeamCity runs using an internal database based on the HSQLDB database engine. The internal database suits evaluation purposes since 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.
External database is usually more reliable and provides better performance.

Internal database may crash and lose all your data (e.g. on out of disk space condition). Also, internal database can become extremely slow on large data sets (say, database storage files over 200Mb). Please also note that our support does not cover any performance or database data loss issues if you are using internal database.

In short, **do not EVER use internal HSQLDB database for production TeamCity instances**.

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

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

This page covers:

Selecting External Database Engine

TeamCity supports MySQL, PostgreSQL, Oracle and MS SQL databases.
As a general rule you should use the database that better suits your environment and that you can maintain/configure better in your organization.

While we strive to make sure TeamCity functions equally well under all of the supported databases, issues can surface in some of them under high TeamCity-generated load.
Our order of preference for the databases would be: MySQL, Oracle, PostgreSQL, MS SQL.

We recommend using MySQL. TeamCity is tested most extensively and as a result might be a bit more stable with MySQL (see also the recommended settings).

General Steps

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

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

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

⚠ Please note that TeamCity actively modifies its own database schema. The user account used by TeamCity should have permissions to create new, modify and delete existing tables in its schema, in addition to usual read/write permissions on all tables.

Database Configuration Properties

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

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

Database Driver Installation

Due to licensing terms, TeamCity does not bundle driver jars for external databases. You will need to download the Java driver and put the appropriate jars (see below) from it into `<TeamCity Data Directory>/lib/jdbc` directory (create it if necessary).
In prior TeamCity versions, put the driver jar(s) into `webapps/ROOT/WEB-INF/lib` directory of TeamCity installation (for .exe and .tar.gz distributions). Please note that you will need to repeat the step next time you upgrade TeamCity.

MySQL

Supported versions
Recommended database server settings:

- use InnoDB storage engine
- use UTF-8 character set
- use case-sensitive collation

Installation:

1. Download the MySQL JDBC driver from http://dev.mysql.com/downloads/connector/j/.
2. Install MySQL connector driver jar (`mysql-connector-java-*-bin.jar` from the downloaded archive).
3. Create an empty database for TeamCity in MySQL and grant permissions to modify this database to a user from which TeamCity will work with this database.
4. In the `<TeamCity data directory>/config` folder rename `database.mysql.properties` file to `database.properties` and specify the required settings in this file:

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

PostgreSQL

Supported versions

1. Download the PostgreSQL JDBC driver from http://jdbc.postgresql.org/download.html and place it into the `<TeamCity data directory>/lib/jdbc`.
2. Create an empty database for TeamCity in PostgreSQL and grant permissions to modify this database to a user from which TeamCity will work with this database. Be sure to set up it to use UTF8.
3. In the `<TeamCity data directory>/config` folder create file `database.properties` and specify the required settings in this file:

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

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

1. Create a schema which name is exactly like the user name; it can be done using the `pgAdmin` tool or with the following SQL:

   ```
   create schema teamcity authorization teamcity;
   ```

   The username should be specified in the 'database.properties' in TeamCity, and has to be in lower case.
   Schema has to be empty (don't contain tables).

2. Start TeamCity.

Oracle

Supported versions

1. Create an Oracle user account for TeamCity (with CREATE SESSION, CREATE TABLE, EXECUTE ON SYS.DBMS_LOCK permissions).
2. Get the Oracle JDBC driver from your Oracle server installation or download it from Oracle web site.
   Supported driver versions are 10.2.0.1.0 and higher.
   It should be two files:
   - ojdbc6.jar
   - orai18n.jar (can be omitted if missing in the driver version)
     Place them into `<TeamCity data directory>/lib/jdbc` directory.
3. In the `<TeamCity data directory>/config` folder create file `database.properties` and specify the required settings in this file:

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

   ⚠ Make sure TeamCity user have quota for accessing table space

Microsoft SQL Server

Supported versions

On MS SQL server side

1. Create a new database. If you're planning to use unix build agents, ensure that the collation is case sensitive.
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.

On TeamCity server side

You can use either JTDS JDBC driver (open source) or MS native JDBC driver (free for downloading).

JTDS driver

1. Download the latest jTDS driver ditributive file (`zip` file), unpack the `jtds-*.jar` driver jar and place it to `<TeamCity data directory>/lib/jdbc`.
2. In the `<TeamCity data directory>/config` folder create file `database.properties` and specify the required settings in this file:

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

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

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

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

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

Native driver

1. Download the MS **sqljdbc** package from [http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=11774] and unpack it. Let us assume the directory where you've unpacked the package into is called *sqljdbc_home*.
2. Copy the sqljdbc4.jar from the just downloaded package into the `TeamCity Data Directory/lib/jdbc` directory.
3. In the `<TeamCity data directory>/config` folder create `database.properties` file and specify the following required settings in this file:

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

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

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

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

1. Ensure that your Java bitness is the same as Windows bitness (in other words, use 64-bit Java with 64-bit Windows and 32-bit Java with 32-bit Windows).
2. Copy the *sqljdbc_home* /enu/auth/x86/sqljdbc_auth.dll (in case of 32-bit system) or *sqljdbc_home* /enu/auth/x64/sqljdbc_auth.dll (in case of 64-bit system) into your Windows/system32 directory (or another directory denoted in %PATH%). Ensure that there are no other sqljdbc_auth.dll files in your system).
3. In the `<TeamCity data directory>/config` folder create file `database.properties` and specify the connection URL (with no user names or passwords) in this file:

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

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

See also:

Installation and Upgrade: Migrating to an External Database

Migrating to an External Database

Please refer to Setting up an External Database page for the general external database information and the database-specific configuration steps. If you want to start using external database from the first TeamCity start, those are the only steps you need.

This section covers steps you need to perform if you need to migrate TeamCity data from one database to another.
Most typical case is when you evaluated TeamCity with internal database and need to switch to an external database to prepare your TeamCity installation for production use.

There are several ways to migrate data into new database:

- Switch with no data migration: build configurations settings will be preserved, but not the historical builds data or users.
- Data Migration: all the data is preserved except for any database-stored data provided by the third-party plugins.
- Backup and then restore: same as migration, but using two-step approach.

> ⚠ Database migration cannot be combined with 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.

Switching to Another Database

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 following steps the server will start with 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. Install and set up an external database.
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 `messages` and `artifacts` folders from `/system` folder of your <TeamCity data directory>; you **may** delete old HSQLDB files: `buildserver.*` to remove the no longer needed internal storage data.
5. Start the TeamCity server.

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 database data.

Full Migration

`maintainDB` command line utility is used to migrate data between databases. `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 TeamCity .tar.gz and .exe distributions.

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 should NOT contain any tables).

> ⚠ If an error occurs during migration, do not use the new database as it can result in database data corruption various errors that can uncover later. In case of error investigate the reason that is logged into console or in the migration logs (see below), and if solution is found, clean the target database and repeat the migration process

To migrate all your existing data to a new external database:

1. Set up an external database to be used by TeamCity. At this point you should only create and configure the database and also install the database driver into TeamCity. Do not modify any TeamCity settings at this time.
2. Shut the TeamCity server down.
3. Create new properties file with a custom name (for example, `database.<database_type>.properties`) for the target database according to its settings from a corresponding template (`<TeamCity Data Directory>/config/database.<database_type>.properties.dist`), entering actual values. Place this file into a temporary directory to your liking.
4. Run the `maintainDB` tool with the `migrate` command and specify the absolute path to the newly created target database properties file

with -T option:

```
maintainDB.[cmd|sh] migrate -A <path to TeamCity Data Directory> -T <path to database.properties
file>
```

You may need to set TEAMCITY_APP_DIR environment variable to path of TeamCity web application directory (default is <TeamCity_home>\webapps\ROOT). Also, if you do not have TEAMCITY_DATA_PATH environment set (pointing to TeamCity Data Directory), you will need to specify the directory via -A parameter of the tool.
Upon the successful completion of the database migration, a database.properties file will be replaced with the file specified via -T option. The old database.properties file will be automatically re-named in the following format: database.properties.before.<timestamp>.

1. Start TeamCity server. This should be the same TeamCity version that was run last time (TeamCity upgrade should be performed as a separate procedure).

Databases Properties Table

DB Name	Driver Class Name	Driver jar Name	Driver is Bundled	JDBC URL format
MySQL	com.mysql.jdbc.Driver	mysql-connector-java-5.1.12-bin.jar	no, download page	jdbc:mysql://<host>[:<port>]/<database>
PostgreSQL	org.postgresql.Driver	postgresql-8.2-505.jdbc3.jar	no, download page	jdbc:postgresql://<host>[:<port>]/<databas
Oracle	oracle.jdbc.driver.OracleDriver	ojdbc16.jar orai18n.jar	no, grab files ojdbc16.jar and orai18n.jar from Oracle server installation or download them form download page	jdbc:oracle:thin:@<host>:<port>:<databas
HSQLDB	org.hsqldb.jdbcDriver	hsqldb.jar	yes	jdbc:hsqldb:hsql://<host>[:<port>]/<databa
MS SQL	net.sourceforge.jtds.jdbc.Driver	jtds-1.x.y.jar where 1.x.y - jtds version; TC supports versions 1.2.2 and higher	no, download page	jdbc:jtds:sqlserver://<host>[:<port>]/<data name>

Troubleshooting

- Extended information during migration execution is logged into logs\teamcity-dbmt.log file. Also, logs\teamcity-dbmt-truncation.log contains extended information on possible data truncations 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.

Alternative approach is to run HSQLDB in 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 running the Migration tool pointing to the database as the source: jdbc:hsqldb:hsql://localhost/buildserver sa ''

- If you get "The input line is too long." error while running the tool (e.g. this can happen on Windows 2000), please change the script to use 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

The User's Guide can be useful for anyone who uses TeamCity. Explore the TeamCity:

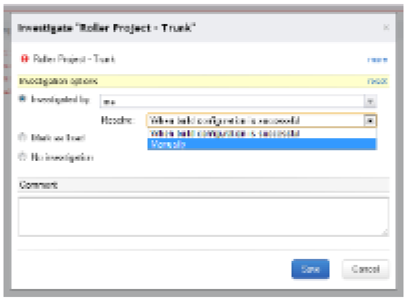
- Modify your user account
- Subscribe to notifications
- View how your changes have affected the builds
- View failing tests and build configuration problems
- Investigate build problems
- Use TeamCity search
- View build configuration and project statistics
- Review build results

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 start investigation yourself. When an investigator is set, he/she receives corresponding notification.
A subject for investigation is not necessarily the whole build, it can also be some perticular test. For each project you can find out what problems are currently being investigated and by whom at the dedicated **Investigations** tab.

To start investigation of a failed build:

1. Navigate to the **Overview** tab of the Build Configuration Home page, or **Overview** tab of the Build Results page, click the *Start investigation* link.
2. Select team member's name from the **User** drop-down list and click **Assign**.



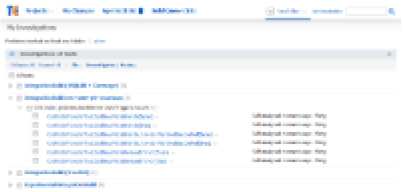
> Note, that normally an investigation is automatically removed once the build configuration becomes green or the test passes. However, **starting with TeamCity 7.1** you can specify that investigation should be resolved manually. This is useful in case of so called "flickering tests": when a test fails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

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

To start 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 a box with a number next to your name.



From **My Investigations** page you can not only see investigations assigned to you in different projects, but you can also manage them: mark as fixed, give up investigation, re-assign, mute a test failure.

Note that for each failed test on this page you can get a bunch of information instantly, without a need to leave this page. For example, you can see all build configurations where this test is currently failing. You can see current stacktrace and information about a build where the test is currently failing. You can also see information about the first failure of this test, again with stacktrace and build.

Since TeamCity 7.1 investigations assigned to you are also highlighted in Web UI, if you have 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

Managing your User Account

In this section:

- Changing Your Password
- Customizing UI
- Managing Version Control Username Settings
- Viewing your Roles and Permissions

Changing Your Password

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.
2. 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 default, and it is not possible to change your password in TeamCity.

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 date and time (for example, build start, vcs change time, etc.) according to it.

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 VCS username. Click **Edit** to provide actual usernames for version control systems you use to be able to view your changes on the My Changes page. Make sure the user names are correct, otherwise you will not be able to track your changes status in the My Changes view.
☆ These settings are not used for authentication for the particular VCS, etc.

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

Search

After you have installed and started running TeamCity, it collects the information on builds, tests and so on and indexes it.
In TeamCity you can search builds by build number, tag, build configuration name and other different parameters specifying one or several keywords and use Lucene search query syntax to get more precise results.

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

In this section:

Search Query

In TeamCity you can search for builds using the Lucene query syntax. Though in TeamCity search query has two major differences, please refer to the Lucene documentation for complete syntax rules description.
To narrow your search and get more precise results you can use available search fields - indexed parameters of each build. Please refer to the Complete list of available search fields for more details.

Differences from Lucene Syntax

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

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

Performing Fuzzy Search

You also have a possibility to perform fuzzy search using the tilde, "~", symbol at the end of a single word term which to search items similar in spelling.

Boolean Operators and Wildcards Support

You can combine multiple terms with Boolean operators to create more complex search queries. In TeamCity, you can use AND, "+", OR, NOT and "-".

> ⚠ When using Boolean operators, type them ALL CAPS.

- AND (same as a plus sign). All words that are linked by the "AND" are included in the search results.

  > ⚠ This operator is used by default.

- NOT (same as minus sign in front of the query word). Exclude a word or phrase from search results.
- OR operator helps you to fetch the search terms that contain either of the terms you specify in the search field.

TeamCity also supports usage of "*" and "?" wildcards in the build query.

> ⚠ Please do not type an asterisk sign, *, at the beginning of the search term because it can take a significant amount of time for TeamCity to search its database. For example, `*onfiguration` search term is incorrect.

Complete List of Available Search Fields, Shortcuts, and Keywords

Search Fields

When using search keywords, use the following query syntax:

```
<search field name>:<value to search>
```

Search Field	Shortcut	Description	Example
agent		Find all builds that were run on specific agent.	`agent:unit-77`, or `agent:agent14*`
build		Find all builds that include changes with specific string.	`build:254` or `build:failed`
changes		Find all builds that include changes with specific string.	`changes:(fix test)`
comitters		Find all build that include changes committed by specific developer.	`comitters:ivan_ivanov`

configuration	c	Find all builds from the specific build configuration.	`configuration:IPR`
			`c:(Nightly Build)`

file_revision		Find all builds that contain a file with specific revision.	`file_revision:5`
files		Find all build that include files with specific filename.	`files:`
labels	l	Find all builds that include changes with specific VCS label.	`label:EAP` `l:release`
pin_comment		Find all builds that were pinned and have specific word (string) in the pin comment.	`pin_comment:publish`
project	p	Find builds from specific project.	`project:Diana` `p:Calcutta`
revision		Find all builds that include changes with specific revision (e.g., you can search for builds with specific changelist from Perforce, or revision number in Subversion, etc.).	`revision:4536`
stamp		Find all builds that started at the specific time (search by timestamp).	`stamp:200811271753`
status		Find all builds with specific status.	`status:failed`
tags	t	Find all builds with specific tag.	`tags:buildserver` `t:release`
tests		Find all builds that include specific tests.	`tests:`
triggerer		Find all builds that were triggered by specific user.	`triggerer:ivan.ivanov`
vcs		Find builds that have specific VCS.	`vcs:perforce`

Shortcuts

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

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

Shortcut	Description	Example
#	Search for build number.	`#<number>`, e.g. `#1234`

@	Find all builds that were run on the specific agent.	`@<agent's name>`, e.g. `@buildAgent1`

Using Double-Colon

You can use double-colon sign (`::`) to search for project and/or build configuration by name:

- `pro::best` — search for builds of configurations with the names starting with "best", and in the projects with the names starting with "pro".
- `mega::` — search for builds in all projects with names starting with "mega"
- `::super` — search for builds of build configurations with names starting with "super"

"Magic" Keywords

TeamCity also provides "magic" keywords (for the list see table below). These magic keywords are formed of the `'$'` sign and a word itself. The word can be shortened up to the one (first) syllable, that is, the `$labeled`, `$l`, and `$lab` keywords will be equal in query. For example, to search for pinned builds of the "Nightly build" configuration in the "Mega" project you can type any of the following queries:

- `configuration:nightly project:Mega $pinned`
- `c:nigh p:mega $pin`
- `M::night $pin`

Magic word	Description
$tagged	Search for builds with tags. For example, `Calcutta::Master $t` query will result in a list of all builds marked with any tag of build configurations which name is started with "Master" from projects with names started with "Calcutta".
$pinned	Search for pinned builds.
$labeled	Search for builds that have been labeled in VCS. For example, to find labeled builds of the Main project you can use following queries: `p:Main $labeled`, or `project:Mai $l`, or `m:: $lab`, etc.
$commented	Search for builds that have been commented.
$personal	Search for personal builds. For example, using `-$p` expression in your query will exclude all personal builds from search results.

Subscribing to Notifications

TeamCity provides a wide range of notification possibilities to keep developers informed about the status of their projects. Notifications can be sent by e-mail, Jabber/XMPP instant messages or can be displayed in the IDE (with the help of TeamCity plugins) or the Windows system tray (using TeamCity Windows tray notifier).

Subscribing to Email, Jabber, IDE, Windows Tray Notifications

TeamCity allows you to flexibly adjust the notification rules, so you will receive notifications only on the events that are interesting to you. 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, you should have your email address specified at the **General** area on the **My Settings & Tools** page.

     > ℹ 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, you should have TeamCity plugin installed in your IDE. For the details on installing TeamCity IDE plugins, please 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 here your Google Talk account, if it is configured be System Administrator to be used.
    - **Windows Tray Notifier**: to receive this type of notifications you should have Windows Tray Notifier installed.
3. For the selected notifications type, specify the notification rules, which are comprised of two parts: what should be watched and notification conditions - the events you want to be notified about. See the description below.

> ⚠ Email and Jabber notifications are sent only if System Administrator has configured TeamCity server email and Jabber settings. System Administrator can also change the templates used for notifications.

To Watch

Builds affected by my changes	Check to monitor only the builds that contain your changes.
Builds from the project	Select a project which builds you want to monitor. Select **All projects** to monitor all of the builds of all the projects' build configurations.
Builds from the selected build configurations	Check to monitor all of the builds of the selected build configurations. Hold the `Ctrl` key to select several build configurations.
System wide events	Select to be notified about system wide events.

Notification Conditions

The build fails	Check this option to receive notifications about all of the failed builds in the specified projects and build configurations. ⚠ • If the **Builds affected by my changes** option is selected in the **Watch** area, you will ***not*** receive notifications about the next failed builds, if they fail with the same errors. • If someone is investigating the build failure, all the subsequent notifications will be sent only to that developer.
Ignore failures not caused by my changes	*This option can only be enabled when the **Watch builds affected by my changes** option is used*. Check this option **not** to be notified when build that includes your changes fails, but fails because of the previous (other people's) changes (e.g. the build fails "the same").
Only notify on the first failed build after successful	Check this option to be notified when the build fails after a successful build. Using this option you will not be notified about subsequent failed builds.
The build is successful	Check this option to receive notifications when a build of the specified build configurations has executed successfully.
Only notify on the first successful build after failed	Check this option to receive notification when only the first successful build occurs after a failed build. Notifications about subsequent successful builds will *not* be sent.
Notify when the first error occurs	Check this option to receive notifications when the first errors are detected, even before the build has finished.
The build starts	Check this option to receive notifications when a build of the specified build configurations starts.
The build fails to start	Check this option to receive notifications when a build of the specified build configurations fails to start.
The build is probably hanging	Check this option to receive notifications when TeamCity suspects that a build of the specified build configurations is hanging.
Investigation is updated	Check this option to receive notifications about the investigation status, like someone is investigating the problem, or problems were fixed, or the investigator has changed, and the first successful build after the fixes.
Tests are muted or unmuted	
Investigation assigned on me	*This option is available only if **System wide events** option is selected in the **Watch** area*. Check the option to be notified each time you start investigating a problem.

ℹ TeamCity applies the notification rules in the order they are presented. TeamCity checks whether a build matches any notification rule, and sends notification according to *the first matching* rule; further check for matching conditions for the same build does not performed.
Note, that you can reorder the notification rules specified.

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

- To unsubscribe from group notifications you can add your own rule with same watched builds and different notification events.
- To unsubscribe from all events, add a rule with corresponding watched builds and no events selected.

Customizing RSS Feed Notifications

TeamCity allows to obtain information about the finished builds, or about the builds with the changes of particular users, via RSS feed. You can customize RSS feed from the TeamCity Tools sidebar of **My Settings&Tools** page (click **customize** to open Feed URL Generator options), or from the home page of a build configuration. TeamCity produces a URL to the syndication feed on the base of the values, which you specify in the Feed URL Generator page.

Feed URL Generator Options

Option	Description
Select Build Configurations	
List build configurations	Specify which build configurations display in the field **Select build configurations or projects**. The following options are available: • **With external status enabled**: if this option is selected, the next field shows the list of build configurations, for which the option **Enable status widget** is set in the General Settings, and build configurations visible for everybody without authentication. • **Available to 'Guest' user**: Select this option to show the list of build configurations, which are visible to the user with the Guest permissions. • **All**: Select this option to show a complete list of build configurations, which requires HTTP authorization. Selecting this option enables the field **Feed authentication settings**.If 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 syndication feed.
Feed Entries Selection	
Generate feed items for	Specify the events, which trigger syndication feed generation. You can opt to select builds, changes or both.
Include builds	Specify the types of builds to be informed about: • 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 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 only available, when **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 **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 base of the values, specified above. You can either copy and paste it to your feed reader, or click the **Subscribe** link.

Supported Additional URL parameters

In addition to the URL parameters available in the Feed URL Generator, the following parameters are supported:

Parameter Name	Description
itemsCount	number, limits the number of items to return in a feed. Defaults to 100.
sinceDate	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 corresponding section on the file syntax.

By default the feed is generated as 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 internal id "bt1", limit the builds to the those started with the last hour but no more then 200 items:

```
http://teamcity.server:8111/httpAuth/feed.html?buildTypeId=bt1&itemsType=builds&sinceDate=-60&itemsCount=
```

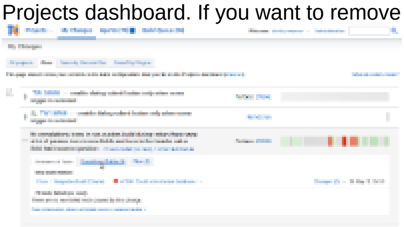
See also:

> **Administrator's Guide**: Customizing Notifications

Viewing Your Changes

For a project developer it is important to understand whether his commit brought a build failure or not. On the **My Changes** page you can review the commits you've made and how they have affected builds.

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

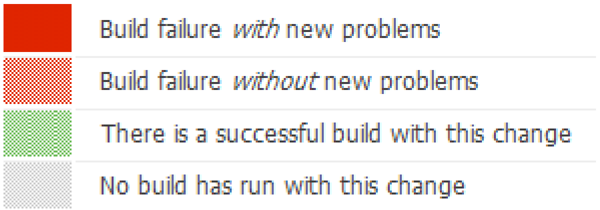


From this page you can:

- View all your commits, and changes included into your personal builds.
- View how your changes have affected the builds.
- See whether there are new failed tests caused by your changes.
- Navigate to the issue tracker, if issue tracker integration is configured.
- Open possibly problematic files in your IDE.
- Navigate to change details.
- View detailed data for each change in dedicated tabs. To switch between tabs for the currently selected change use `Tab/Shift+Tab` or mnemonics: 'T' for tests, 'B' for builds, 'F' for files.
- View suspicious builds with your change. For personal changes, all builds are shown.

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

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



🟥	Build failure *with* new problems
▨	Build failure *without* new problems
🟩	There is a successful build with this change
▨	No build has run with this change

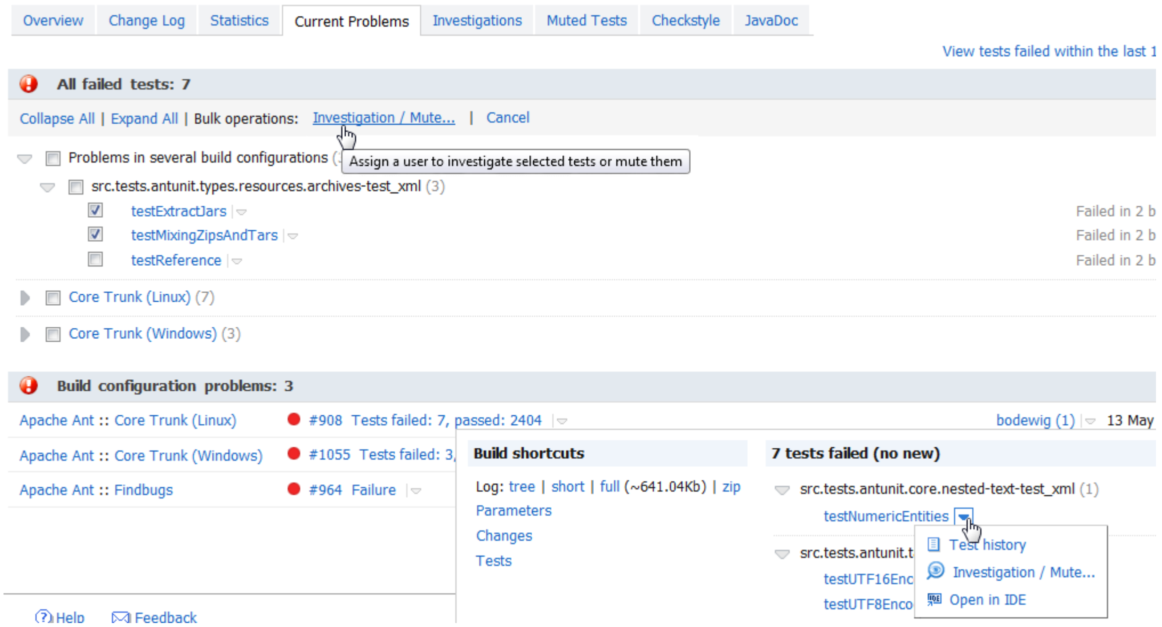
See also:

Concepts: Build State | Change

Viewing Tests and Configuration Problems

Viewing Currently Failing Tests and Build Configuration Problems for a Project

To view tests and build configurations that are currently failing in a project, open Project Home page and go to the **Current Problems** tab.



Overview
Change Log
Statistics
Current Problems
Investigations
Muted Tests
Checkstyle
JavaDoc
View tests failed within the last 1
Collapse All | Expand All | Bulk operations: Investigation / Mute... | Cancel
Assign a user to investigate selected tests or mute them
Help
Feedback

All failed tests: 7

☐ Problems in several build configurations (

☐ src.tests.antunit.types.resources.archives-test_xml (3)

☑ testExtractJars — Failed in 2 b

☑ testMixingZipsAndTars — Failed in 2 b

☐ testReference — Failed in 2 b

☐ Core Trunk (Linux) (7)

☐ Core Trunk (Windows) (3)

Build configuration problems: 3

Apache Ant :: Core Trunk (Linux)	● #908 Tests failed: 7, passed: 2404	bodewig (1)	13 May
Apache Ant :: Core Trunk (Windows)	● #1055 Tests failed: 3,		
Apache Ant :: Findbugs	● #964 Failure		

Build shortcuts

Log: tree | short | full (~641.04Kb) | zip

Parameters

Changes

Tests

7 tests failed (no new)

src.tests.antunit.core.nested-text-test_xml (1)

testNumericEntities

src.tests.antunit.t

testUTF16Enc

testUTF8Enco

Test history

Investigation / Mute...

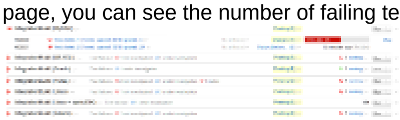
Open in IDE

From this page you can:

- Navigate to all tests failed within the last 120 hours for the project or for the particular build configuration within the project: to do so, click the corresponding link.
- Navigate to a test's stacktrace by clicking it's name.
- Navigate to build results, view short or full build log, download build log, and more: click the arrow next to the build results link.
- Start investigating particular test failure.
- Open particular test in IDE.

Overview of Current Problems

Starting with TeamCity 7.1 to get a glance of current problems in your project, or in several projects, you don't need to go to the Project home page, you can see the number of failing tests in build configurations and other problems right on the Overview page.

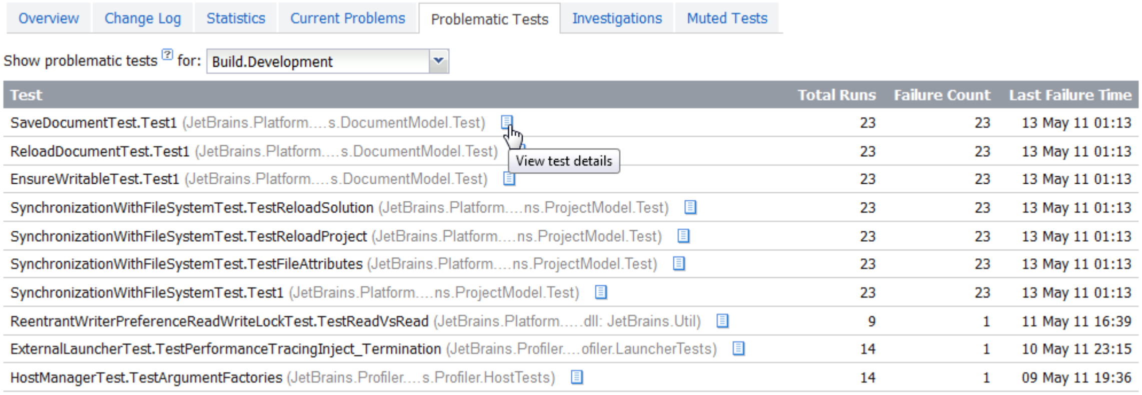


Note the presentation of problems is shortened if your browser does not have enough horizontal space.

Viewing Tests Failed within Last 120 Hours

To view all the tests failed within last 120 hours for a project, from the Project Home page go to the **Current Problems** tab and click the **view all tests failed within the last 120 hours** link. The **Problematic Tests** tab is opened, where for each test you can view the total number of test runs, failure count and last time the test failed.

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

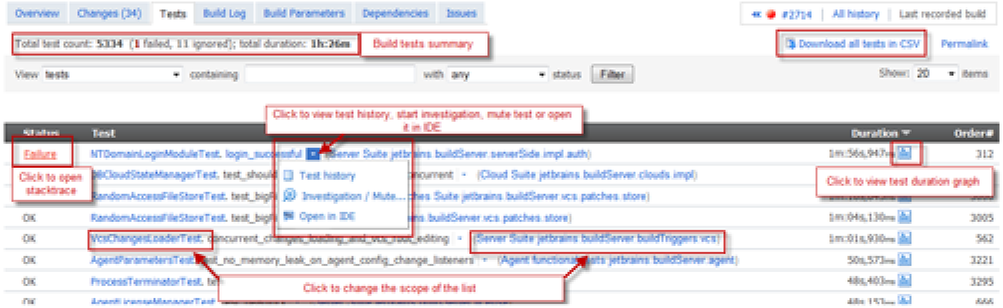


Show problematic tests for: Build.Development

Test	Total Runs	Failure Count	Last Failure Time
SaveDocumentTest.Test1 (JetBrains.Platform.…s.DocumentModel.Test)	23	23	13 May 11 01:13
ReloadDocumentTest.Test1 (JetBrains.Platform.…s.DocumentModel.Test)	23	23	13 May 11 01:13
EnsureWritableTest.Test1 (JetBrains.Platform.…s.DocumentModel.Test)	23	23	13 May 11 01:13
SynchronizationWithFileSystemTest.TestReloadSolution (JetBrains.Platform.…ns.ProjectModel.Test)	23	23	13 May 11 01:13
SynchronizationWithFileSystemTest.TestReloadProject (JetBrains.Platform.…ns.ProjectModel.Test)	23	23	13 May 11 01:13
SynchronizationWithFileSystemTest.TestFileAttributes (JetBrains.Platform.…ns.ProjectModel.Test)	23	23	13 May 11 01:13
SynchronizationWithFileSystemTest.Test1 (JetBrains.Platform.…ns.ProjectModel.Test)	23	23	13 May 11 01:13
ReentrantWriterPreferenceReadWriteLockTest.TestReadVsRead (JetBrains.Platform.….dll: JetBrains.Util)	9	1	11 May 11 16:39
ExternalLauncherTest.TestPerformanceTracingInject_Termination (JetBrains.Profiler.…ofiler.LauncherTests)	14	1	10 May 11 23:15
HostManagerTest.TestArgumentFactories (JetBrains.Profiler.…s.Profiler.HostTests)	14	1	09 May 11 19:36

Viewing All Tests for a Build

To view all the tests for a particular build, open the build results page, and navigate to the **Tests** tab.


Overview
Changes (34)
Tests
Build Log
Build Parameters
Dependencies
Issues
#2714
All history
Last recorded build
Total test count: 5334 (1 failed, 11 ignored); total duration: 1h:26m
Download all tests in CSV
Permalink
View tests
containing
with any
status
Filter
Show: 20
items
Status
Test
Duration
Order#
Failure
NTDomainLoginModuleTest. login_successful
Test history
Investigation / Mute...
Open in IDE
OK
RandomAccessFileStoreTest. test_big
VcsChangesLoaderTest. concurrent_changes_loading_and_vcs_root_editing
Server Suite jetbrains buildServer buildTriggers vcs
AgentParametersTest. test_no_memory_leak_on_agent_config_change_listeners
Agent functional tests jetbrains buildServer agent
ProcessTerminatorTest.
1m:56s,947ms
312
1m:04s,130ms
3005
1m:01s,930ms
562
50s,573ms
3221
48s,403ms
3295

Build tests summary

Click to view test history, start investigation, mute test or open it in IDE

Click to open stacktrace

Click to view test duration graph

Click to change the scope of the list

On this page:

Item	Description
Download all tests in CSV	Click the link to download a file containing all the build tests results.
Filtering options	Use this area to filter the test list thus excluding unnecessary items: • 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 status of test.
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 in a red **Failure** link, 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 name of a class, a namespace/package, or a suite to view only items that are included in it. Click the arrow next to the test name to view test history, start investigation of the failed test, or open the failed test in IDE.
Duration	Shows the time it took to complete the test. Click the 📈 button to view the **Test Duration Graph** popup (see below).
Order#	Shows the sequence in which the tests were run. Click the header above this column to sort by test order number.

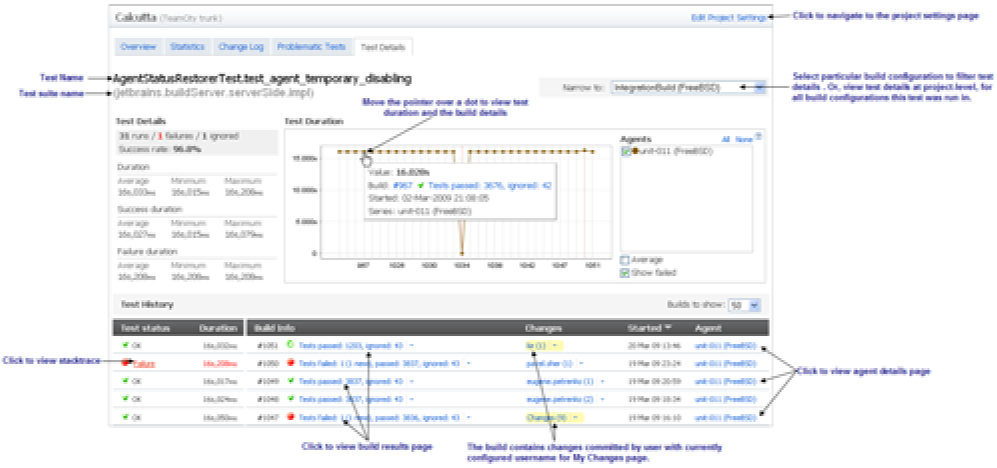
Viewing Particular Test's History

To navigate to a particular test's history, click the arrow next to the test name and select **Test History** from the drop-down as shown in the screenshot above.
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:

- Test details section including test success rate and test's run duration data:
- Test duration graph. For more information, please refer to the Test Duration Graph description below.
- Complete test history table, that contains information about the test status, its duration, and information on a build this test was run.



Calcutta (TeamCity trunk)

Overview | Statistics | Change Log | Problematic Tests | Test Details

AgentStatusRestorerTest.test_agent_temporary_disabling
(jetbrains.buildServer.serverSide.impl)

Narrow to: IntegrationBuild (FreeBSD)

Test Details
31 runs / 1 failures / 1 ignored
Success rate: 96.8%

Value: 16.020s
Build: #967 ✓ Tests passed: 3676, ignored: 42
Started: 02-Mar-2009 21:08:05
Series: unit-011 (FreeBSD)

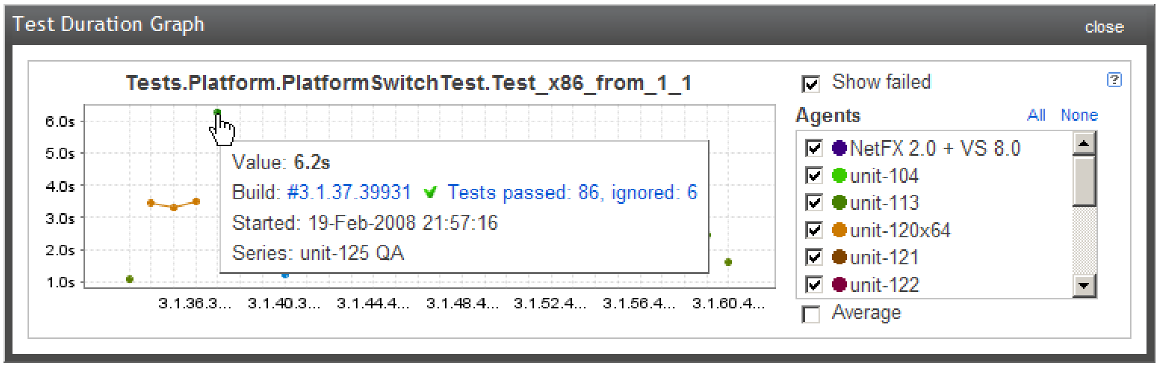
Test History

Test status	Duration	Build info	Changes	Started	Agent

Click to navigate to the project settings page
Edit Project Settings
Test Name
Test suite name
Move the pointer over a dot to view test duration and the build details
Select particular build configuration to filter test details . Or, view test details at project level, for all build configurations this test was run in.
Test Duration
Agents
All None
unit-011 (FreeBSD)
Average
Show failed
Duration
Success duration
Failure duration
Average
Minimum
Maximum
Builds to show: 50
Click to view stacktrace
Click to view agent details page
Click to view build results page
The build contains changes committed by user with currently configured username for My Changes page.

Test Duration Graph

The test duration graph is useful for comparing the amount of time it takes individual tests to run on the builds of this build configuration.



Value: 6.2s
Build: #3.1.37.39931 ✓ Tests passed: 86, ignored: 6
Started: 19-Feb-2008 21:57:16
Series: unit-125 QA

☑ Show failed

☑ NetFX 2.0 + VS 8.0
☑ unit-104
☑ unit-113
☑ unit-120x64
☑ unit-121
☑ unit-122

□ Average

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 build's test duration dot plotted on the graph to jump to the corresponding build results **Build Results** page.
- View a build summary in the tooltip of a build's test duration dot, and navigate to the corresponding **Build Results** page.
- Filter information by agents by selecting or clearing a particular agent or by clicking **All** or **None** links to select or clear all agents.

See also:

Concepts: Testing Frameworks

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 in to categories:

- Project-level statistics available at **Project home page** | **Statistics** tab.
- Build Configuration-level statistics available at **Build Configuration home page** | **Statistics** tab.

Regardless of the statistics level in the Statistics tab you can:

- Select time range for each type of statistics from the **Range** drop-down list.
- Filter information by data series, for example, by Agent name or by result type.
- View average values (**Average** check box).

- Filter out failed builds and show only the successful builds (**Show Failed** option).
- View build summary information and navigate to the build results page.

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

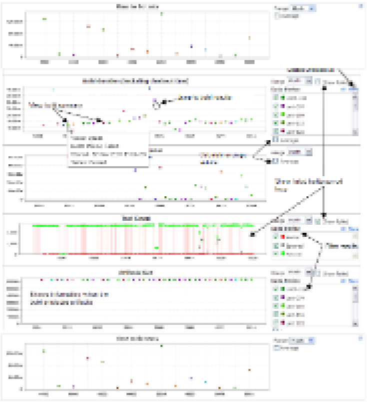
For each project TeamCity provides visual charts with statistics gathered from all build configurations included in the project through entire history of the project. These charts show statistics for code coverage, code inspections and code duplicates for build configurations within the project that produce corresponding reports.

You can adjust the set of project-level charts in 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 build configuration level. These charts demonstrate: successful build rate, 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 they 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 checkout time)**: This chart allows to monitor the duration of the build. To get the best picture of 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 build agent management 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. Information about individual tests is available on the build results page.
- **Artifacts Size**: This chart tracks the total size of all artifacts produced by the build.
- **Time to fix tests**: This chart tracks the maximum amount of time it took to fix the tests of the particular build. If not all build tests were fixed, a red vertical stripe is displayed.
- **Code Coverage**: Blue, green, dark cyan, and purple dots show respectively the percentages of the classes blocks, lines and methods covered by the tests.
- **Code Duplicates**: This chart tracks the number of duplicates discovered in the code.
- **Code Inspection**: This chart displays red and yellow dots to track respectively the number of discovered errors and warnings.

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

Tests Statistics

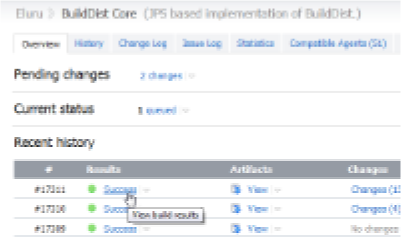
For a particular test you can also find some useful statistics: **Test duration** graph on "Test History" page, which allows to compare the amount of time it takes individual tests to run on the builds of this build configuration. For more details, please refer to related page.

See also:

Concepts: Build Configuration | Build State | Change
Administrator's Guide: Customizing Statistics Charts

Working with Build Results

In TeamCity all information about particular build, regardless if it's still running or already finished, is accumulated on so-called **build results** page. However, some data is accessible only after the build is finished. Build results can be accessed from various places in TeamCity web UI, for example, from Build Configuration home page:



#	Results	Artifacts	Changes
#17311	● Success	View	Changes (1
#17310	● Succe	View	Changes (4)
#17309	● Success	View	No changes

In addition to the general information about the build, like build duration, agent used, trigger, etc., build results page also contains following useful information:

- Tests Failed in the Build
- Changes Included in Build
- Build Log
- Build Parameters
- Dependencies
- Related Issues
- Build Artifacts
- Code Coverage Results
- Code Inspection Results
- Duplicates Found
- Maven Build Info

If another build of this same build configuration is concurrently running, a small window appears on the **Overview** tab with the following information:

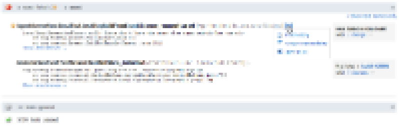
- The build results of that build linking to the build results page;
- A changes link with a drop-down list of changes included in that build;
- The build number, time of start, and a link to the agent it is running on.

If a build is probably hanging, the corresponding notification is displayed on top of the Overview tab of the build results. Moreover, in this case TeamCity provides a link to view the process tree of the running build, and 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 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

Tests Failed in the Build

If the build has failed tests, you can view them right on the **Overview** tab of the build results page:



For each failed you can jump to its history, assign a team member to investigate its failure, open the test in your IDE to start fixing it right away. Moreover, all tests are displayed in the dedicated **Tests** tab. Learn more.

Changes Included in Build

Not only you can review changes included in build, from the **Changes** tab you can also:

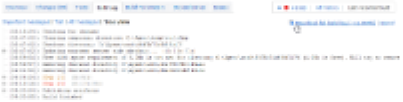
- Label build sources.
- Configure VCS settings of the build configuration (if you have enough permissions).

For each particular change you can (when expanded):

- Trigger custom build with this change.
- Explore change in details.
- Open TeamCity difference viewer for a modified file in a change.
- Open modified files in your IDE.
- Review change in external change viewer, if configured by administrator.

Build Log

For each build you can view and download its build log in preferable way:



Build Parameters

All system properties and environmental variables which were used by the particular build are listed on the **Build Parameters** tab of the build results. Learn more about build parameters.

Dependencies

If the finished build has artifact and/or source dependencies, the **Dependencies** tab is displayed on the build results page, where you can explore builds which artifacts and/or sources were used for creating the particular build as well as the builds which used the artifacts and/or sources of the current build.
Additionally you can opt to view indirect dependencies for the build. That is, for example, 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 the build A.

Related Issues

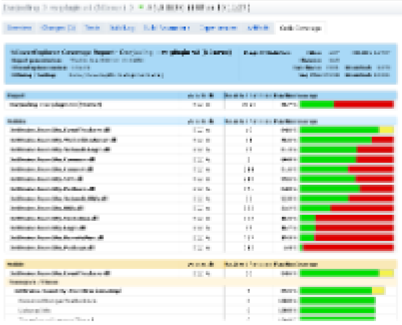
If you have the issue tracker integration configured, and there is at least one issue mentioned in the comments for included changes or in the comments for the build itself, you'll see the list of issues related to the current build in the **Issues** tab.
If you need to view all the issues related to a build configuration and not just for particular build, you can navigate to the **Issues Log** tab available on the build configuration home page, where you can all the issues mapped to the comments or filter the list to particular range of builds.

Build Artifacts

If the build has produced some artifacts, they are accumulated at the dedicated **Artifacts** tab.

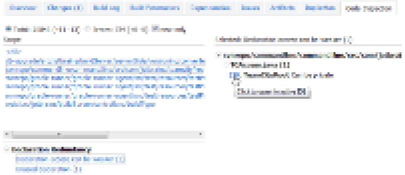
Code Coverage Results

If you have code coverage configured in your build runner, dedicated tab appears in build results that shows the full HTML code coverage report.



Code Inspection Results

The results of the **Code Inspection** build step are gathered at the **Code Inspection** tab in the build results.

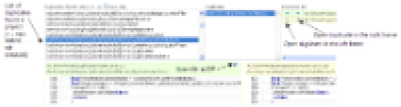


Use the left pane to navigate through inspection results, the filtered inspections are shown in the right pane.

- Switch from **Total** to **Errors** option , if you're not interested in warnings.
- Use 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 Found

If your build configuration has Duplicates build runner as one a build step, 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 specific code of interest.

Maven Build Info

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.

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

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 User Accounts, Groups and Permissions
- Customizing Notifications
- Managing Licenses
- Assigning Build Configurations to Specific Build Agents
- Patterns For Accessing Build Artifacts
- Tracking User Actions
- Mono Support
- Maven Support
- Integrating TeamCity with Other Tools

TeamCity Configuration and Maintenance

> Server configuration is only available to the System Administrators.

To edit the server configuration:

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

This section also includes:

- Managing Data and Backups
- Configuring Authentication Settings
- TeamCity Startup Properties
- Changing user password with default authentication scheme
- Configuring Server URL
- Configuring TeamCity Server Startup Properties
- Configuring UTF8 Character Set for MySQL
- Enabling Guest Login
- 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

Managing Data and Backups

In this section:

- TeamCity Data Backup

TeamCity Data Backup

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

TeamCity provides several ways to back up its data:

- Backup from web UI: an action in 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 maintainDB console tool. Some limitations on backed up data apply. This option is also available on upgrade in the maintenance screen - on the first start of newer version of TeamCity server.
- Backup via maintainDB command-line tool: Same as via UI, but requires server not running and has no limitations. Restore is possible via maintainDB console tool.

- Manual backup: is suitable if you want to manage the backup procedure manually.

If you need to back up build agent's data, refer to Backing up Build Agent's Data.
For instructions on how to restore the data from backup, please refer to the Restoring TeamCity Data from Backup page.

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

Recommended approach is either to perform backup described under Manual Backup and Restore or run backup from web UI regularly (e.g. regularly via REST API) with level "All except build artifacts" - this will ensure backup for all important data except for Build logs and artifacts. Build logs and artifacts (if necessary) can be backed up manually by copying files under `.BuildServer/system/messages` and `.BuildServer/system/artifacts`. See TeamCity Data Directory for details on what is stored in artifacts and build logs.

See also:

> **Installation and Upgrade**: Upgrade

Creating Backup via maintainDB command-line tool

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

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

To get short reference for all available `maintainDB` options, run `maintainDB` from 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.
- TeamCity application manual customizations under `<TeamCity server home>`, including used server port number
- TeamCity application logs (they also reside under `<TeamCity server home>`.
- Any manually created files under `<TeamCity Data Directory>` that do not fall into previously mentioned items.

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

The default directory for the backup files is the `<`*`TeamCity Data Directory`*`>\backup`.

> ⚠ If not specified otherwise with the `-A` option, TeamCity will read the TeamCity Data Directory path from the `TEAMCITY_DATA_PATH` environment variable, or the default path (`$HOME\.Buildserver`) will be used.

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

Performing TeamCity Data Backup with maintainDB Utility

⚠ Before starting data backup you need to stop TeamCity server.

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

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

To specify type of data to include in the backup file, use the following options:

- -C or --include-config — includes build configurations settings
- -D or --include-database — includes database
- -L or --include-build-logs — includes build logs
- -P or --include-personal-changes — includes personal changes

Specifying different combinations of the above options, you can control the content of the backup file. For example, to create backup with all supported types of data, run

```
maintainDB backup -C -D -L -P
```

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

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.

Creating Backup from TeamCity Web UI

TeamCity allows creating a backup of TeamCity data via the Web UI.
To create a backup file, navigate to the **TeamCity Backup** page of the **Administration** section, specify backup parameters, as described below and start backup process.

Option	Description

Backup file	Specify the name for the backup file, the extention (.zip) will be added automatically. By default, TeamCity will store the backup file under `<TeamCity Data Directory>/backup` folder. For security reasons you cannot explicitly change this path in the UI. To modify this setting specify an absolute or relative path (the path should be relative to TeamCity Data Directory) in the `<TeamCity Data Directory>/config/backup-config.xml` file. For example:
	`<backup-settings>`
	`  ...`
	`  <general>`
	`    <backup-dir path="C:/TC-Backups"/>`
	`  </general>`
	`  ...`
	`</backup-settings>`

add time stamp suffix	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. • If the directory where backup files are stored already contains a file with the name specified above, TeamCity won't run backup - you will need either to specify another name, or enable *time stamp suffix* option, which allows to avoid this. • Time stamp suffix has specific format: sorting backup files alphabetically will also sort them chronologically.
Backup scope	Specify what kind of data you want to back up. The contents of the backup file depending on the scope is described right in the UI, when you select a scope. Note, that the size of the backup file and the time the backup process will take depends on the scope you select. You can select the "basic" scope, which includes server settings, projects and builds configurations, plugins and database, to reduce the resulting file size and the time spent on backup. However, you'll be able to restore only those settings which were backed up.

When you start backup, TeamCity will display its status and details of the current process including progress and estimates.

> **Important notes**
>
> - Backup process takes some time that depends on how many builds there are in system. During this process the system's state can change, e.g. some builds may finish, other builds that were waiting in the build queue may start, new builds may appear in the build queue, etc. Note, that these changes won't influence the backup. TeamCity will backup only the data actual by the time the backup process was started.
> - The resulting backup file is a *.zip archive which has specific structure that doesn't depend on OS or database type you use. Thus, you can use the backup file to restore your data even on another Operating System, or with another database. If you'll change the contents of this file manually, TeamCity won't be able to restore your data.

On the **History** tab of **Administration** | **TeamCity Backup** page you can review the list of created backup files, their size and date when the files were created. Note that only backup files created from web UI are shown here. If you have perfomed backup by means of maintainDB utility, they are not displayed on the **History** tab.

See also:

Installation and Upgrade: Upgrade
Concepts: TeamCity Data Directory
Administrator's Guide: TeamCity Data Backup

Manual Backup and Restore

Server Manual Backup

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

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

The following is needed to be backed up:

TeamCity data directory

TeamCity Data Directory directory stores:

- server settings, projects and build configurations with their settings (i.e. all that is configured on Administration web UI)
- build logs and build artifacts
- current operation files, internal data structure, etc.

You may refer to TeamCity Data Directory section on the directory structure and more details. If necessary, you can exclude parts of the directory from backup to save space — you will lose only the excluded data. You may safely exclude "`system/caches`" directory from backup — the necessary data will be rebuild from scratch on TeamCity startup.
If you decide to skip data backup under <`TeamCity Data Directory`>`/system` directory, make sure you note the most recent files in each of `artifacts`, `messages` and `changes` subdirectories and save this information. It will be needed if you decide to restore the database backup with TeamCity Data Directory corresponding to a newer state then 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 <`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, please back up your database schema used by TeamCity using database-specific tools.
  External database connection settings used by TeamCity can be looked up in the <`TeamCity Data Directory`>`/config/database.properties` file. See also 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 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 *non-war* distribution: back up **everything** under <TeamCity home directory> except for `temp` and `work` directories.
- If you use *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 `<TeamCity home directory>/logs` directory.

⚠ You may also want to back up TeamCity Windows Service settings, if they were modified.

Manual Restoration of Server Backup

If you need to restore backup created with the web UI or `maintainDB` utility, please refer to Restoring TeamCity Data from Backup. This section describes restoration of manually created backup.

You should always restore both data in <TeamCity Data Directory> and data in the database. Both database and the directory should be backed up/restored in sync.

TeamCity Data Directory restoring

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, please make sure the files created after the backup was done are cleared. Especially the newly appeared files under the `artifacts`, `messages`, `changes` directories under `<TeamCity Data Directory>/system`.

TeamCity Database restoring

When restoring database, please ensure there are no extra tables in the schema used by TeamCity.

Restoring to a new server

If you want to run a copy of the server, make sure the servers use distinct data directories and databases. For external database make sure you modify settings in `<TeamCity Data Directory>/config/database.properties` file to point to another database.

Restoring TeamCity Data from Backup

TeamCity allows administrator to restore previously backed up data using the `maintainDB` command line utility.

⚠ Please note that restoration of the backup created with TeamCity versions earlier then 6.0 can only be performed with **the same TeamCity version** which created the backup.
Backups created with TeamCity 6.0 or more recent versions can be restored by the same or more recent TeamCity versions.

To restore TeamCity server from previously saved backup file, ensure TeamCity server is not running and target TeamCity Data Directory and database are present, but empty.
Then use `maintainDB` utility which is located in the <TeamCity Home>/bin directory. That is only available in TeamCity .tar.gz and .exe distributions.
Use the `restore` command:

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

-A argument can be omitted if you have TEAMCITY_DATA_PATH environment variable set.
-T argument can be omitted if you want to restore the data into the same database the backup was created from. For restoration into internal database, use `.BuildServer\config\database.hsqldb.properties.dist` file available as the default content of the TeamCity Data Directory.

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

By default, if no other option that `-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 corresponding options of the `maintainDB` utility: `-C`, `-D`, `-L`, and `-P`.
To get the reference for the available options of the `maintainDB`, run the utility without any command or option.

Restoring Data from Backup to Another Database

You can restore data into another empty database. Types of the source (from which the data is backed up) and target (to which the data will be restored) databases don't matter. For instance, you can restore data from a HSQL database to a MySQL database, as well as restore a backup of MySQL database to a new MySQL database.
Essentially, restoring data from a backup file to another database is a migration process.

To restore database data to another database:

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

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

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

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

See also:

Administrator's Guide: Creating Backup via maintainDB command-line tool

Configuring Authentication Settings

Out-of-the-box TeamCity supports three authentication schemes:

- Default Authentication
- Windows Domain Authentication
  - Windows Domain Authentication on Unix-like computers
- LDAP Integration (separate page)

Currently used authentication scheme is displayed on the **Administration** | **Global Settings** page.

Switching Authentication Scheme

To switch from one authentication scheme to another you need to edit `<TeamCity data directory>/config/main-config.xml` file on the server machine. Change the value of `class` attribute of `<login-module>` tag inside `<auth-type>` tag.
Supported values for `class` attribute are:

- `jetbrains.buildServer.serverSide.impl.auth.DefaultLoginModule` for Default Authentication
- `jetbrains.buildServer.serverSide.impl.auth.NTDomainLoginModule` for Windows Domain Authentication
- `jetbrains.buildServer.serverSide.impl.auth.LDAPLoginModule` for LDAP Authentication

Also, TeamCity plugins can provide additional authentication schemes. Please restart the server after editing the file.

Please note that each authentication type maintains own list for users. This means that on switching from one authentication to another you start with no users (and no administrator) and will be prompted for administrator account on first TeamCity start after the authentication change. This also means that all the existing users will need to create their accounts and re-enter their settings anew.

If you are not prompted for administrator account on switching to a new scheme, this means that there are users in the scheme already.
Please refer to How To...#Retrieve Administrator Password section for a resolution.

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

```
<auth-type>
    <!-- Active login module class, see below -->
    <login-module class="jetbrains.buildServer.serverSide.impl.auth.LDAPLoginModule" />
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
    <guest-login allowed="true" />
    <!-- Allow users to self-register (only for modules which support this feature, e.g.
DefaultLoginModule) (true|false) -->
    <free-registration allowed="false" />
</auth-type>
```

Default Authentication

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

```
<auth-type>
    <login-module class="jetbrains.buildServer.serverSide.impl.auth.DefaultLoginModule" />
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
    <guest-login allowed="true" />
    <!-- Allow users to self-register (only for modules which support this feature, e.g.
DefaultLoginModule) (true|false) -->
    <free-registration allowed="true" />
</auth-type>
```

Users database is maintained by TeamCity. New users are added by TeamCity administrator (in administration area at the **Users** page) or user are self-registered if `<free-registration allowed="true" />` tag is specified.

Windows Domain Authentication

See also NTLM HTTP Authentication for transparent login withotu manual credentials entering: Single sign-on based on Windows domain authentication.

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

```
<auth-type>
    <login-module class="jetbrains.buildServer.serverSide.impl.auth.NTDomainLoginModule" />
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are allowed (true|false) -->
    <guest-login allowed="true" />
</auth-type>
```

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

It is also possible to log in using only a username if the domain is specified in `ntlm.defaultDomain` property of `<TeamCity data directory>/config/ntlm-config.properties` file.

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

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

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

jCIFS Library Specific Configuration

The library is configured using the properties specified in `<TeamCity data directory>/config/ntlm-config.properties` file. Changes to the file take effect immediately without server restart.

If default settings does not work for your environment, please refer to http://jcifs.samba.org/src/docs/api/ for all available configuration properties.
If the library does not find domain controller to authenticate against, consider adding `jcifs.netbios.wins` property in the `ntlm-config.properties` file with 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.

See also:

> **Concepts**: Authentication Scheme

LDAP Integration

LDAP integration in TeamCity has two levels: authentication (login) and synchronization.

LDAP integration might be not trivial to configure, so it might require some trial and error approach to get the right settings.
It is recommended to configure LDAP on a test server before switching it on on the production one. LDAP logs should give you enough information to understand possible misconfigurations. If you are experiencing difficulties configuring LDAP integration after going through this document and investigating the logs, please contact us and let us know your LDAP settings with a detailed description of what you want to achieve and what you currently get.

Authentication

If you need to limit the users who can log into TeamCity to LDAP users, you need to set LDAP authentication module in `main-config.xml` file and configure LDAP connection settings in `ldap-config.properties` file. Refer to Authentication Settings page for details.

`main-config.xml` Configuration

Configure `login-module` settings of `<TeamCity data directory>/config/main-config.xml` as follows:

```
<auth-type>
    <login-module class="jetbrains.buildServer.serverSide.impl.auth.LDAPLoginModule" />
    <!-- Welcome message displayed to users on login form -->
    <login-description>Welcome to TeamCity, your team building environment!</login-description>
    <!-- Whether anonymous "view-only" logins are supported -->
    <guest-login allowed="true" />
</auth-type>
```

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

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

`ldap-config.properties` Configuration

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

> ⚠ Please note that all the values in the file should be properly escaped

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

The mandatory property in `ldap-config.properties` file is `java.naming.provider.url` that configures the server and root DN. The property stores URL to the LDAP server node that is used in following LDAP queries. For example, `ldap://dc.example.com:389/CN=Users,DC=Example,DC=Com`. Please note that the value of the property should use URL-escaping if necessary. e.g. use `%20` if you need space character.

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

Configuring User Login

> Since TeamCity 5.0 a new login scheme is applied: the user is *first* searched in LDAP, then authenticated using the DN retrieved during the search. It makes the configuration more flexible and also several properties redundant (formatDN and loginFilter), though in some particular cases using formatDN is simplier.
> Backwards compatibility: the first step (search) is optional, thus one can login with username entered to the login form (formatted using formatDN option, etc).

You might want to configure the integration so that users enter only the username in TeamCity login form, but LDAP server can require prefixes or suffixes to the name. This can be addressed with `teamcity.auth.formatDN` property (**Since TeamCity 4.5**, previous versions require "`formatDN`" as the property name). The property defines the transformation that is applied to the entered username before the LDAP login is tried. The property should reference the entered username as `$login$`. Note that TeamCity store the username for the user in unmodified form as entered by the user. This is necessary to allow mapping of LDAP user back to TeamCity user, refer to the `teamcity.users.username` property description below.

Example:

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

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

Example (allow any username):

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

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

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

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

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

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

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

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

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

Related external link: How To Set Up Secure LDAP Authentication with TeamCity by Alexander Groß.

Active Directory

The following template enables authentication against active directory:

Add the following code to the <`TeamCity Data Directory`>`/config/ldap-config.properties` file (assuming the domain name is "`Example.Com`" and domain controller is "`dc.example.com`").

```
java.naming.provider.url=ldap://dc.example.com:389/DC=Example,DC=Com
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
teamcity.users.login.filter=(sAMAccountName=$capturedLogin$)
teamcity.users.username=sAMAccountName
java.naming.security.authentication=simple
java.naming.referral=follow
```

Advanced Configuration

If you need to fine-tune LDAP connection settings, you can add `java.naming` options to the `ldap-config.properties` file: they will be passed to underlying Java library. Default options are retrieved using `java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory`. Refer to the Java documentation page for more information about property names and values.

You can use an LDAP explorer to browse LDAP directory and verify the settings (for example, http://www.jxplorer.org/).

There is an ability to specify failover servers using the following pattern:

```
java.naming.provider.url=ldap://ldap.mycompany.com:389 ldap://ldap2.mycompany.com:389
ldap://ldap3.mycompany.com:389
```

The servers are contacted until any of them responds. There is no particular order in which the address list is processed.

Synchronization

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

LDAP synchronization allows to:

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

Periodically, TeamCity fetches data from LDAP and updates users in TeamCity. You can review the last synchronization run statistics and schedule new synchronization in **LDAP Synchronization** section of server settings.

Common Configuration

You need to have LDAP authentication configured for the synchronization to function.

By default, the synchronization is turned off. To turn it on, add the following option to `ldap-config.properties` file:

```
teamcity.options.users.synchronize=true
```

You also need to specify the following mandatory properties: `java.naming.security.principal` and `java.naming.security.credentials` (they specify user credentials which are used by TeamCity to connect to LDAP and retrieve data), `teamcity.users.base` and `teamcity.users.filter` (specify the settings to search for users), and `teamcity.users.username` (the name of LDAP attribute containing the username).

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

Example:

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

User Profile Data

When properly configured, TeamCity can retrieve user-related information from LDAP (e-mail, full name, or any custom property) and store it as TeamCity user's details. If updated in LDAP, the data will be updated in TeamCity user's profile. If modified in user's profile in TeamCity, the data will no longer be updated from LDAP for the modified fields.

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

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

The list of supported user settings:

- `teamcity.users.username`
- `teamcity.users.property.displayName`
- `teamcity.users.property.email`
- `teamcity.users.property.plugin:notificator:jabber:jabber-account`
- `teamcity.users.property.plugin:vcs:<VCS type>:anyVcsRoot` — VCS username for all <VCS type> roots. The following VCS types are supported: svn, perforce, jetbrains.git, cvs, tfs, vss, clearcase, starteam.

Example properties can be seen by configuring them for a user in web UI and then listing the properties via REST API Plugin#Users.

User Group Membership

TeamCity can automatically update users membership in groups based on the LDAP-provided data.
You will need to create groups in TeamCity manually and then specify the mapping on LDAP groups in `<TeamCity data directory>/config/ldap-mapping.xml` file. Use `ldap-mapping.xml.dist` .dist file as an example.

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

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

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

Creating and Deleting User

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

By default, automatic user creation is turned off. To turn it on, set `teamcity.options.createUsers` property to `true` in `ldap-config.properties` file.

TeamCity can automatically delete users in TeamCity if they cannot be found in LDAP or do not belong to an LDAP group that is mapped to predefined "**All Users**" group. By default, automatic user deletion is turned off as well; set `teamcity.options.deleteUsers` property to turn it on.

Username migration

The username for the existing users can be updated upon first successful login. For instance, suppose the user had previously logged in using 'DOMAIN\user' name, thus the string 'DOMAIN\user' had been stored in TeamCity as the username. In order to synchronize the data with LDAP user can change the username to 'user' using the following options:

```
teamcity.users.login.capture=DOMAIN\\\\(.*)
teamcity.users.login.filter=(cn=$login$)
teamcity.users.previousUsername=DOMAIN\\\\$login$
```

The first property allows you to capture the username from the input login and use it to authenticate the user (can be particularly useful when the domain 'DOMAIN' isn't stored anywhere in LDAP). The second property `teamcity.users.login.filter` allows you to fetch the username from LDAP by specifying the search filter to find this user (other mandatory properties to use this feature: `teamcity.users.base` and `teamcity.users.username`). The third property allows to find the 'DOMAIN\user' username when login with just 'user', and replace it with either captured login, or with the username from LDAP.

Note that if any of these properties are not set, or cannot be applied, the username isn't changed (the input login name is used).

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.
Also, you can get detailed logs of LDAP login and synchronization processes. See TeamCity Server Logs for details.

Typical LDAP Configurations

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

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

Basic LDAP Login

Backup LDAP server is specified. Provided users can log in into LDAP with "`EXAMPLE\Username`", they log in into TeamCity also as "`EXAMPLE\Username`", username stored in TeamCity is "Username".

```
# The second URL is used when the first server is down.
java.naming.provider.url=ldap://example.com:389/DC=example,DC=com
ldap://failover.example.com:389/DC=example,DC=com

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

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

Basic LDAP Login for Users in a Specific LDAP Group Only

Only users from specific users group allowed to login. Users need to enter only username without domain part to login.

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

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

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

# filtering only users with specified name and belonging to LDAP group "Group1" with DN
"CN=Group1,CN=Users,DC=example,DC=com"
teamcity.users.login.filter=(&(sAMAccountName=$capturedLogin$)(memberOf=CN=Group1,CN=Users,DC=example,DC=
retrieving TeamCity username form the "sAMAccountName" LDAP entry attribute
teamcity.users.username=sAMAccountName

# Allow only username part without domain
teamcity.auth.loginFilter=[^/\\\\@]+

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

Active Directory With User Details Synchronization

Users can log in into TeamCity with their domain name without domain part, there is an account "teamcity" with password "secret" that can read all Active Directory entries. TeamCity user display name and email are synchronized from Active Directory.

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

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

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

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

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

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

Active Directory With Group Synchronization

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

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

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

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

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

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

LDAP Troubleshooting

General advice: if you experience problems with LDAP configuration, please 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 it is:

- login filter doesn't match the entered login ("`User-entered login does not match teamcity.auth.loginFilter=..., aborting`")
- LDAP server rejected login with "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 `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 `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 unescaped sequence. Try to login with this principal using another LDAP tool.
- Try to change the security level (`java.naming.security.authentication`): can be "simple", "strong" or "none".

Users in LDAP are stored in different branches, so `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 do you want to find the user (`teamcity.users.login.filter`), e.g. by 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

TeamCity NTLM HTTP authentication feature employs Integrated Windows Authentication and allows transparent/SSO login to TeamCity web UI when using browsers/clients supporting NTLM, Kerberos or Negotiate HTTP authentications.
Generally, it allows to login into TeamCity server using NT domain account normally without the need for user to enter credentials manually.

This feature is only available **since TeamCity 7.1**.

NTLM HTTP authentication is supported only for TeamCity servers installed on Windows machines and configured to use Windows domain Authentication. If you need it under other platforms, feel free to contact us with details as to why.

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

Requirements

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

Enabling NTLM HTTP Authentication

NTLM HTTP authentication currently works only with "Windows Domain" authentication scheme, so please first make sure you use this scheme on your server. Also please make sure you do NOT use compatibility mode.

With this settings, users will see a link on login screen which, when clicked will force browser to send domain authentication data.

You can force the server to announce NTLM HTTP authentication by setting the following internal property:

```
teamcity.http.auth.forceProtocols=ntlm
```

This will make the server to request domain authentication for any request to the TeamCity web UI. If the user's browser is run in domain environment, the current user will be logged in automatically. If not, browser will popup a dialog asking for domain credentials.

Without this property NTLM HTTP authentication will work only if client explicitly initiates it (e.g. clicks on "Login using NT domain account" link on the login page), and in usual case unauthenticated user will be simply redirected to the TeamCity login page.

Since version 7.1.1 TeamCity server forces NTLM HTTP authentication only for Windows users by default. If you want to enable it for all users, set the following internal property:

```
teamcity.ntlm.ignore.user.agent=true
```

NTLM login URL

There is one more way to force NTLM authentication for certain connection (there is no necessity to set property `teamcity.http.auth.forceProtocols` for this case). Just send request to `<Your TeamCity server URL>/ntlmLogin.html` and TeamCity will initiate NTLM authentication.

Client configuring

According to your environment, you may need to configure your client to make NTLM authentication work.

Internet Explorer

1. Open "Tools" -> "Internet Options".
2. On the "Advanced" tab make sure the option "Security -> Enable Integrated Windows Authentication" is checked.
3. On the "Security" tab select "Local Intranet" -> "Sites" -> "Advanced" and add your TeamCity server URL to the list.

Google Chrome

On Windows Chrome normally uses IE's behaviour, see more information here.

Mozilla Firefox

1. Type `about:config` in the browser's address bar.
2. Add your TeamCity server URL to 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

TeamCity Startup Properties

Please see corresponding section:
Configuring TeamCity Server Startup Properties
Configuring Build Agent Startup Properties

Changing user password with default authentication scheme

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

To change user password:

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

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

   Unix platform:

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

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

See also:

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

Configuring Server URL

> You can modify server URL right from the web UI, at the **Administration** | **Global Settings** page.

In most cases TeamCity correctly autodetects its own URL and uses it for constructing URLs in notification messages. However, sometimes autodetection is impossible (for example, the server might be running on HTTP and HTTPS or it might be hidden behind the Apache web server). For such cases you can specify 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>
```

This URL will be used when URLs in notification messages are constructed.

Configuring TeamCity Server Startup Properties

Various aspects of TeamCity behavior can be customized through a set options passed on 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 TeamCity support team or you know what you are doing.

In this section:

- TeamCity internal properties
- JVM Properties
  - Server is Run Via Shell Script
  - TeamCity Server is Run as Windows Service

TeamCity internal properties

TeamCity has some properties that are not exposed to UI and are meant for debug use only. If you need to set such a property (e.g. asked by TeamCity support), you can set it either as `-D<name>=<value>` JVM property (see below), or you can add TeamCity-specific properties in `<TeamCity Data Directory>/config/internal.properties` file. The file is Java properties file, to add property put `<property name> = <property value>` on a separate line. If you do not have the file, just create a new one.

JVM Properties

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 run the server using Windows service and use TeamCity 7.0 or earlier, please refer to TeamCity Server is Run as Windows Service, if you are using `.war` distribution, use the manual of your Web Application Server. In all other cases, please refer to Server is Run Via Shell Script.

For general notes on the memory settings, please refer to Setting Up Memory settings for TeamCity Server.

You will need to restart the server for the changes to take effect.

Server is Run Via Shell Script

If you run the server using the `runAll` or `teamcity-server` scripts or as a Windows service with TeamCity 7.1 and above, you can set the options via the environment variables:

- `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 user you start TeamCity under. You might need to reboot the machine after the environment change for the changes to have effect.

TeamCity Server is Run as Windows Service

TeamCity 7.1

Since TeamCity 7.1 you will need to set environment variables as described in Server is Run Via Shell Script to affect TeamCity run a a service.

TeamCity 7.0 and previous versions

For TeamCity 7.0 and previous versions please use the instructions below:
To edit JVM server options run Tomcat's service configuration editor by executing the command

```
tomcat7w.exe //ES//TeamCity
```

in `<TeamCity home>\bin` directory and then edit the Java Options on the *Java* tab (for more information see Tomcat 6 documentation).

To change heap memory dedicated to the JVM change the "Maximum memory pool" setting.

See also:

> **Concepts**: TeamCity Data Directory
> **Administrator's Guide**: Configuring Build Agent Startup Properties

Configuring UTF8 Character Set for MySQL

To create a MySQL database which uses the utf8 character set:

1. Create a new database:

   ```
   CREATE DATABASE <database name> DEFAULT CHARACTER SET utf8
   ```

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 TeamCirty 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 uft8 as the default character set:

```
create database <new database name> default character set utf8;
```

4. Modify the `<TeamCity data directory>/config/database.properties` file as follows:
   - change `connectionUrl` property to:

     ```
     jdbc:mysql://<host>/<new database name>
     ```

   - add `characterEncoding` property:

     ```
     connectionProperties.characterEncoding=UTF-8
     ```

5. Import data the new database:

```
maintainDB restore -D -F database_backup -T <TeamCity data directory>/config/database.properties
```

6. Start the TeamCity server up

Enabling Guest Login

To enable guest login to TeamCity:

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

To customize more details of these users' rights:

- Click the **Configure guest user roles** link and specify the desired restrictions on a page which opens.

After you have configured at least one project in TeamCity, you can specify the roles and permissions of the guest users in it.
When a new guest user account is created, it gets the specified roles by default.
Click the **Assign role** link to associate the default guest account with a role for the selected projects.

> Alternatively, you can edit the **auth-type** section of the `main-config.xml` file and set `<guest-login allowed="true"/>`.

See also:

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

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

Using HTTPS to access TeamCity server

This document describes how to configure various TeamCity server clients to use HTTPS for communicating with the server.
We assume that you have already configured HTTPS in your web server. See how to do this for Tomcat here: http://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html. It's also a common approach to setup a middle server like Apache that will handle HTTPS but will use Tomcat to handle the requests. 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:

- To enable HTTPS connections from TeamCity Visual Studio plugin and 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), save server certificate to a file, and then import it into the corresponding Java keystore using `keytool` program. By default, Java keystore is protected by password: `changeit`
- For **Build Agent**, to import certificate, use the following command:

  ```
  keytool -importcert -file <cert file> -keystore <agent installation
  path>/jre/lib/security/cacerts
  ```

  in case you do not use bundled JRE with agent, use the following command:

  ```
  keytool -importcert -file <cert file> -keystore <agent custom JDK/JRE>/lib/security/cacerts
  ```

- For IntelliJ Platform Plugin or Eclipse Plugin:

  ```
  keytool -importcert -file <cert file> -keystore <path to JDK used by
  IDE>/jre/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.

Authenticating with the help of client certificate

Importing client certificate
If you need to use client certificate to access the TeamCity server via https from IntelliJ IDEA, Eclipse or the agents, you will need to add the certificate to Java keystore and supply the keystore to the JVM used by the IDE.

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 it's 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 it's name is <path to root certificate>.

4. Now import the root certificate to the trusted keystore with the command:

```
keytool -importcert -trustcacerts -file <path to root certificate> -keystore <path to trust keystore
file> -storepass <trust keystore password>
```

Here you can use new values for <trust keystore path> and <trust keystore password> (or use existing trust keystore).

Starting IDE

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 agent, see agent startup properties.

TeamCity Disk Space Watcher

TeamCity server regularly checks for free disk space on the server machine (in TeamCity Data Directory) and displays a warning on all the pages of the web UI is the free disk space falls below a certain threshold. The default limit is 10Mb.

The threshold can be changed by `teamcity.diskSpaceWatcher.threshold` internal property. The number is the size in kilobytes, default value is 10000 (10Mb).

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 TeamCity server machine (usually `<TeamCity Server home>\logs`).
The files are appended with messages when TeamCity is running.

> **Enable Debug in Server Logs**
> In web UI Go to **Administration** | **Diagnostics** page and choose 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
- Reading Logs

General Logging Description

TeamCity uses log4j library for the logging and its settings can be customized.

By default, log files are located under `<TeamCity Server home>/logs` directory.

The most important default log files are:

`teamcity-server.log`	General server log
`teamcity-activities.log`	Log of user-initiated and main build-related events
`teamcity-vcs.log`	Log of VCS-related activity
`teamcity-notifications.log`	Notificaitons-related log
`teamcity-clouds.log`	(off by default) Cloud-integration (Amazon EC2) -related log
`teamcity-sql.log`	(off by default) Log of SQL queries, see details
`teamcity-http-auth.log`	(off by default) Log with messages related to NTLM and other authentication for HTML requests
`teamcity-xmlrpc.log`	(off by default) Log of messages sent by the server to agents and IDE plugins via XML-RPC
`vcs-content-cache.log`	(off by default) Log related to individual file content requests from VCS
`teamcity-rest.log`	(off by default) REST-API related logging
`teamcity-freemarker.log`	(off by default) Notification templates processing-related logging
`teamcity-agentPush.log`	(off by default) Logging related to agent push operations
`teamcity-remote-run.log`	(off by default) Logging related to personal builds processing on the server
`teamcity-svn.log`	(off by default) SVN integration log
`teamcity-tfs.log`	(off by default) TFS integration log

`teamcity-starteam.log`	(off by default) StarTeam integration log
`teamcity-clearcase.log`	(off by default) ClearCase integration log
`teamcity-ldap.log`	LDAP-related log

Other files can also be created on changing logging configuration.

Some of the files can have ".N" extensions - that are files with previous logging messages copied on main file rotation. See maxBackupIndex for preserving more files.

Logging-related Diagnostics UI

Users with System Administrator role can view and download server logs right from TeamCity web UI under **Administration** | **Diagnostics** | **Server Logs**.

Also, users with System Administrator role can change active logging preset for the server logs for the server logs under **Administration** | **Diagnostics** page, **Troubleshooting**, **Debug logging** subsection. Choosing a preset will change logging configuration until the server is restarted (see TW-14313).
New presets can also be uploaded via **Diagnostics** | **Logging Presets**.

The available presets are configured by the files available under `<TeamCity Data Directory>/config/_logging` directory with .xml extension. New files can be added into the directory and existing files can be modified (using .dist convention).

Changing Logging Configuration

By default TeamCity searches for log4j configuration in `../conf/teamcity-server-log4j.xml` file (this resolves to `<TeamCity Server home>/conf/teamcity-server-log4j.xml` for TeamCity .exe and .tar.gz distributions when run from "bin"). If no such file is present, default log4j configuration is used.
The logs are saved to `../logs` directory by default.

These values can be changed via corresponding "log4j.configuration" and "teamcity_logs" internal properties. e.g. `log4j.configuration=file:../conf/teamcity-server-log4j.xml` and `teamcity_logs=../logs/`

While TeamCity is running, logging configuration for the server can be switched to a logging preset.

If you want to fine-tune log4j configuration, you can edit `teamcity-server-log4j.xml` while the server is running - the logging configuration will be changed on the fly.

The logs are rotated by default. When debug is enabled it makes sense to increase the number of rotating log files. While doing so please ensure there is sufficient free disk space available.

You can change the log4J configuration files while the server/agent is running. If it is possible (some log4j restrictions apply), the loggers will be reconfigured without process restart.

If it is not possible to enable debug logging mode from the TeamCity web UI, remove from the `conf/teamcity-server-log4j.xml` file all the lines containing following comment

```
<!-- DELETE THIS LINE FOR ENABLING DEBUG LOGGING -->
```

When debug is enabled it makes sense to increase the number of rotating log files. While doing so please ensure there is sufficient free disk space available.

You can change the log4J configuration files while the server/agent is running. If it is possible (some log4j restrictions apply), the loggers will be reconfigured without process restart.

Most useful settings of log4j configuration:
To change the minimum log level to save in the file, tweak "value" attribute of "priority" element:

```
<category ...>
    <priority value="INFO"/>
...
```

To save more rolling files, tweak "value" attribute of "maxBackupIndex" element:

```
<appender ...>
    <param name="maxBackupIndex" value="10"/>
...
```

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 log for such warnings to better understand what is going OK and what is not.

INFO is an informational message that just reports on the current activities.

DEBUG is only useful for issue investigation. e.g. to be analyzed by TeamCity developers.

See also:

> **Administrator's Guide**: Viewing Build Agent Logs
> **Troubleshooting**: Reporting Issues

Build Agents Configuration and Maintenance

In this section:

- Configuring Build Agent Startup Properties
- Viewing Agents Workload
- Viewing Build Agent Details
- Viewing Build Agent Logs
- Agent Pools

Configuring Build Agent Startup Properties

In TeamCity a build agent contains two processes:

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

Whether you run a build agent via the `agent.bat|sh` script or as a Windows service, at first the agent launcher starts and then it starts the agent.
For both processes you can customize the final agent behavior by specifying system properties and variables to run with.

> ⚠ You do not need to specify any of the options unless you are advised to do by 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

Build Agent Is Run Via Script

Before you run the `<Agent Home>\bin\agent.bat|sh` script, set the following environment variables:

- `TEAMCITY_AGENT_MEM_OPTS` — Set agent memory options (JVM options)
- `TEAMCITY_AGENT_OPTS` — additional agent JVM options

Build Agent Is Run As Service

In the `<Agent Home>\launcher\conf\wrapper.conf` file, add the following lines (one per option):

```
wrapper.app.parameter.<N>
```

> ⚠
>
> - You should add additional lines *before* the following line in the `wrapper.conf` file:
>
>   ```
>   wrapper.app.parameter.N=jetbrains.buildServer.agent.AgentMain
>   ```
>
> - Please ensure to re-number all the lines after the inserted ones.

Agent Launcher Properties

It's rare that you would ever need these. Most probably you would need affecting main agent process properties described above.

Build Agent Is Run Via Script

Before you run the `<Agent Home>\bin\agent.bat|sh` script, set the `TEAMCITY_LAUNCHER_OPTS` environment variable.

Build Agent Is Run As Service

In the `<Agent Home>\launcher\conf\wrapper.conf` file, add the following lines (one per option, the `N` number should increase):

```
wrapper.java.additional.<N>
```

> ⚠ Please ensure to re-number all the lines after the inserted ones.

See also:

> **Concepts**: Agent Home Directory
> **Administrator's Guide**: Configuring TeamCity Server Startup Properties

Viewing Agents Workload

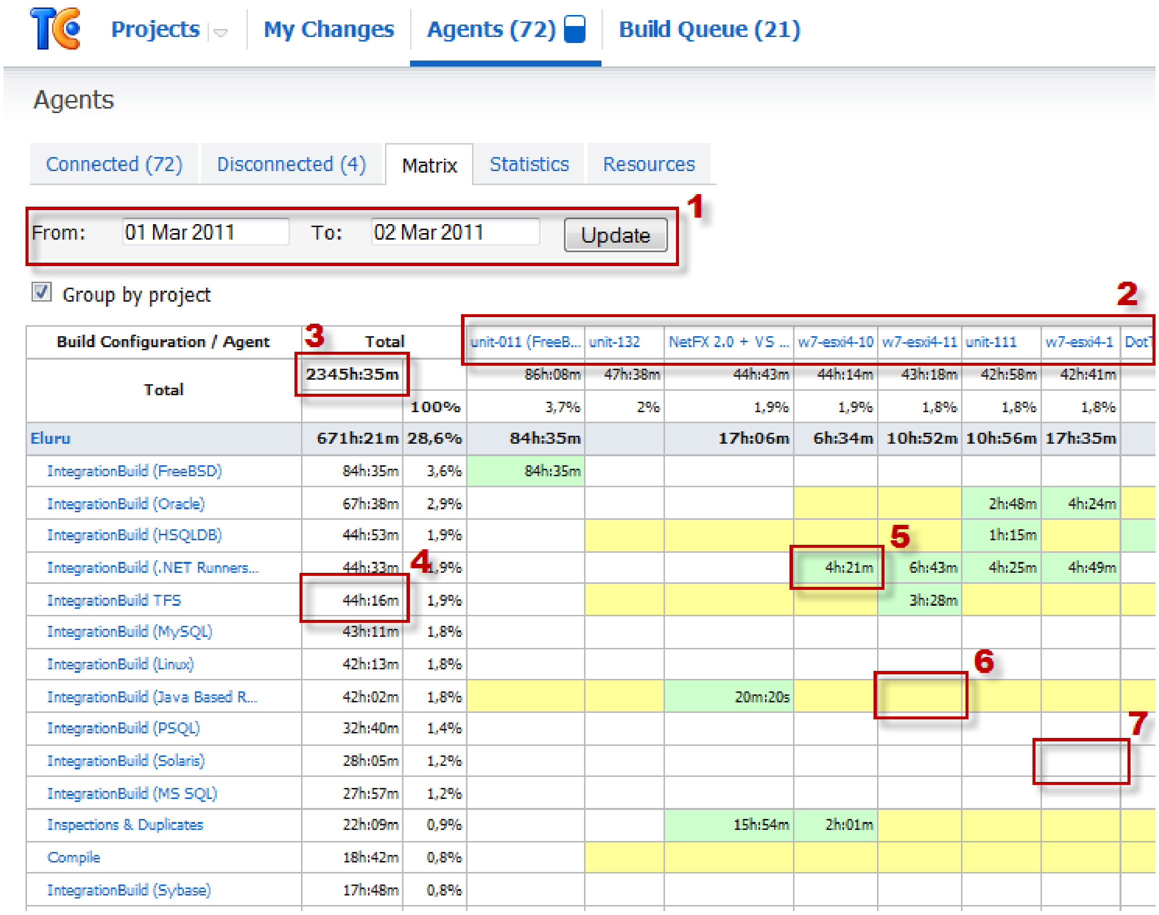
TeamCity provides handy ways to estimate build agents efficiency and help you manage your system:

- Load statistics matrix
- Build Agents' workload statistics

Load Statistics Matrix

The Matrix avialable 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.



From: 01 Mar 2011 To: 02 Mar 2011

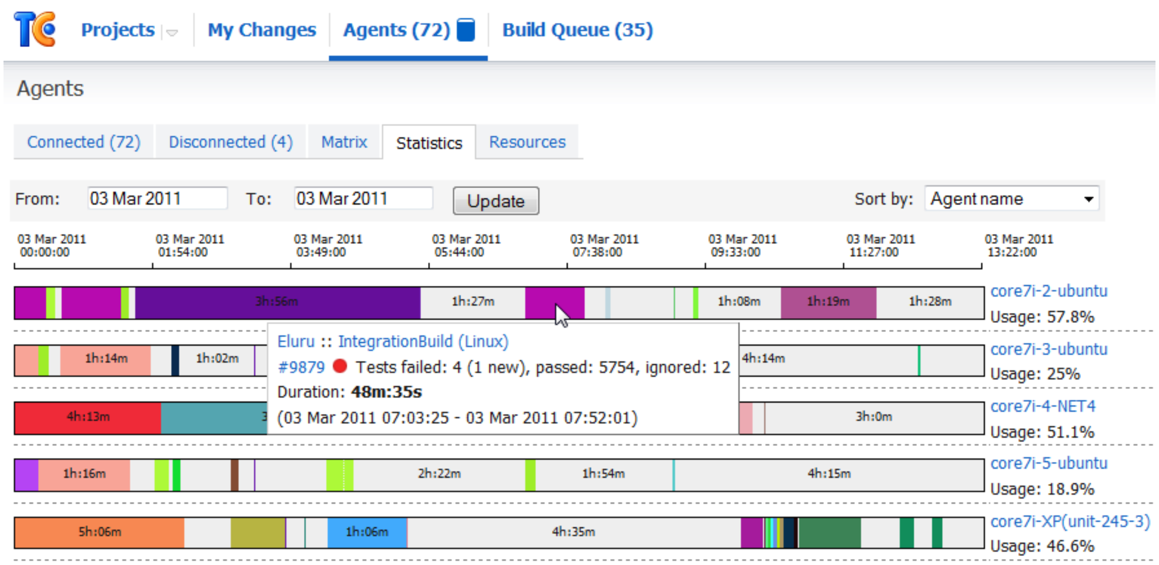
☑ Group by project

Build Configuration / Agent	Total		unit-011 (FreeB...	unit-132	NetFX 2.0 + VS ...	w7-esxi4-10	w7-esxi4-11	unit-111	w7-esxi4-1	DotT
Total	2345h:35m		86h:08m	47h:38m	44h:43m	44h:14m	43h:18m	42h:58m	42h:41m	
		100%	3,7%	2%	1,9%	1,9%	1,8%	1,8%	1,8%	
Eluru	671h:21m	28,6%	84h:35m		17h:06m	6h:34m	10h:52m	10h:56m	17h:35m	
IntegrationBuild (FreeBSD)	84h:35m	3,6%	84h:35m							
IntegrationBuild (Oracle)	67h:38m	2,9%				🟨	🟨	2h:48m	4h:24m	🟨
IntegrationBuild (HSQLDB)	44h:53m	1,9%		🟨	🟨	🟨	🟨	1h:15m	🟨	🟩
IntegrationBuild (.NET Runners...	44h:33m	1,9%				4h:21m	6h:43m	4h:25m	4h:49m	
IntegrationBuild TFS	44h:16m	1,9%		🟨	🟨	🟨	3h:28m	🟨	🟨	🟨
IntegrationBuild (MySQL)	43h:11m	1,8%								
IntegrationBuild (Linux)	42h:13m	1,8%								
IntegrationBuild (Java Based R...	42h:02m	1,8%	🟨	🟨	20m:20s	🟨	🟨	🟨	🟨	🟨
IntegrationBuild (PSQL)	32h:40m	1,4%								
IntegrationBuild (Solaris)	28h:05m	1,2%								
IntegrationBuild (MS SQL)	27h:57m	1,2%								
Inspections & Duplicates	22h:09m	0,9%			15h:54m	2h:01m	🟨	🟨	🟨	🟨
Compile	18h:42m	0,8%		🟨	🟨	🟨	🟨	🟨	🟨	🟨
IntegrationBuild (Sybase)	17h:48m	0,8%								

1. Select the time range and click **Update** to reload the matrix.
2. Click to navigate to agent's details.
3. Total build duration during specified time.
4. Total build duration for particular build configuration, for builds that were run during specified time range.
5. Total duration of the particular build configuration builds that were run on the particular build agent during specified time period.
6. Build agent is compatible with the build configuration, but no builds were run during specified time range.
7. Build agent is incompatible with the configuration, or disconnected.

Build Agents' Workload Statistics

TeamCity provides a number of visual metrics, namely, statistics of Build Agents' activity and their usage during a particular time period. These information is available at the **Statistics** tab on the **Agents** page.



From: 03 Mar 2011 To: 03 Mar 2011 Sort by: Agent name

Eluru :: IntegrationBuild (Linux)
#9879 ● Tests failed: 4 (1 new), passed: 5754, ignored: 12
Duration: **48m:35s**
(03 Mar 2011 07:03:25 - 03 Mar 2011 07:52:01)

core7i-2-ubuntu
Usage: 57.8%

core7i-3-ubuntu
Usage: 25%

core7i-4-NET4
Usage: 51.1%

core7i-5-ubuntu
Usage: 18.9%

core7i-XP(unit-245-3)
Usage: 46.6%

You'll find this feature helpful in:

- your daily administration activities aimed at lowering your company's network workload
- locating and eliminating the gap between the most frequently used computers and those which are often idle
- reducing the cost of your hardware resources ownership

See also:

> **Concepts**: Build Agent
> **Installation and Upgrade**: Installing and Running Additional Build Agents

Viewing Build Agent Details

To view the state and information about an agent click it's name, or navigate to the **Agents** page, find it in the list of connected, disconnected or authorized agents and click its name.
For each connected agent TeamCity provides following details:

- Agent's status. Learn more about enabled/disabled agents.
- Agent's name, host name and IP address, port number, operating system, agent's CPU rank and current version number.
- Information about build currently running on the agent with the link to the build results.
- Options to:
  - Clean sources
  - Open Remote Desktop. Available for agents running on the Windows operating system, if RDP client is installed and registered on the agent.
  - Reboot agent machine. Available to users who have "Reboot build agent machines" permissions. See below for additional configuration of the feature.
  - Dump threads on agent.
- Compatible and incompatible build configurations with the reason of incompatibility.
- Build runners supported by build agent.
- Agent parameters including system properties, environment variables and configuration parameters. Refer to the Configuring Build Parameters page for more information on different types of parameters.

Configuring Agent Reboot Command

Agent reboot is performed by executing an OS-specific command.
Under certain circumstances the command might need customization specific to the OS environment. If the agent reboot does not work by default, you can add `teamcity.agent.reboot.command` agent configuration parameter in `buildagent.properties` with the command to execute when reboot is required.
Example configuration:

```
teamcity.agent.reboot.command=sudo shutdown -r 60
or
teamcity.agent.reboot.command=shutdown -r -t 60 -c "TeamCity Agent reboot command" -f
```

See also:

> **Concepts**: Build Agent | Run Configuration Policy
> **Installation and Upgrade**: Installing and Running Additional Build Agents

Viewing Build Agent Logs

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.

Build agent logs are placed into `<agent home>/logs` directory.

Log Files

File name	Description
`teamcity-agent.log`	General build agent log
`teamcity-build.log`	`stdout` and `stderr` output of builds run by the agent
`upgrade.log`	log of the build agent upgrade (logged by the upgrading process)
`launcher.log`	log of the agent's monitoring/launching process
`wrapper.log`	(only present when the agent is run as Windows service or by Java Service Wrapper) output of the process build agent launching process

You can configure location of the logs by altering the value of **teamcity_logs** property (passed to JVM via `-D` option).
You can also change Log4J configuration file location by changing value of **log4j.configuration** property.
See corresponding documentation section on how to pass the options.

For additional option on logging tweaking please consult TeamCity Server Logs#log4jConfiguration section.

Specific Debug Logging

To get dump of the data sent from the agent to the server, enable agent XML-RPC log, by uncommenting the line below in the `<agent home>/conf/teamcity-agent-log4j.xml` file.

```
<category name="jetbrains.buildServer.XMLRPC">
    <!--<priority value="DEBUG"/>-->
    <appender-ref ref="ROLL.XMLRPC"/>
  </category>
```

Then, see teamcity-xmlrpc.log
To turn it off, make the line "`<priority value="INFO"/>`".

See also:

> **Administrator's Guide**: TeamCity Server Logs
> **Troubleshooting**: Reporting Issues

Agent Pools

Instead of having one common set of agents you can break it 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 **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 **Agents** | **Pools** tab. To be able to add\edit pools you need to have "Manage agent pools" permission. In default TeamCity per-project authorization mode following users have this permission: system administrator, agent manager and project administrator.

To create a new agent pool you need only 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 can possibly cause compatibility problems TeamCity will give a warning.

When you have configured agent pools, you can:

- Filter build queue by pools.
- Group by pool on Agent Matrix and Agent Statistics pages.

See also:

Concepts: Agent Requirements
Administrator's Guide: Viewing Agents Workload

Managing Projects and Build Configurations

Project

Project is a collection of build configurations. The project has a name (usually corresponds to the name of a real project) and can have some meaningful description.

TeamCity uses project level for managing security. It is possible to assign per-project roles to users. This makes it possible to assign users different permissions for each project.

Projects can be created/edited from the **Administration** page. To create new project you need only to specify its name and, additionally, description. Then you can create as many build configurations within the project as you need.

Build Configuration

Build Configuration describes a class of builds of a particular type, or a procedure used to create builds. Examples of build configurations are *integration builds*, *release builds*, *nightly builds*, and others.. For more details, please refer to the Build Configuration concept page.

See also:

Concepts: Project | Build Configuration
Administrator's Guide: Archiving Projects | Creating and Editing Build Configurations

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 the agents become idle. The build is assigned to the agent only when it is started on the agent, no pre-assignment is made while the build still waits in the build queue.

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

- Manually Reordering Builds in Queue
- Managing Build Priorities
- Removing Builds From Build Queue

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 number is - the higher 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 **System Administator** 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.

See also:

> **Concepts**: Build Queue

Creating and Editing Build Configurations

Build Configuration describes a class of builds of a particular type, or a procedure used to create builds. Examples of build configurations are *integration builds*, *release builds*, *nightly builds*, and others. To learn more about build configurations in TeamCity, refer to the Build Configuration page.
TeamCity provides several ways to create new build configuration for a project. You can:

- Create new build configuration from scratch.
- Create build configuration template, and then create build configuration based on the template.
- Create Maven build configuration by importing settings from `pom.xml`

Creating New Build Configuration

1. On the **Administration** page, locate the desired project in the list and click **Create build configuration** link which is available right after the list of project's configurations. Alternatively, you can find this link on the project settings page.
2. Specify General Settings for the build configuration, then click the **VCS settings** button.
3. Create/edit VCS roots and specify VCS-specific settings, then click the **Add Build Step** button.
4. On the **New Build Step** page, configure first build step - select the desired build runner from the drop-down list, and define its settings. Click Save.
   You can add as many build steps as you need within one build configuration. Note, that they will be executed sequentially. In the end, the build gets merged status and the output goes into the same build log. If some step fails, the rest are not executed.
5. Additionally, configure build triggering options, dependencies, properties and variables and agent requirements.

Creating Build Configuration Template

To learn more about build configuration templates, refer to the Build Configuration Template page.
Creating a build configuration template is similar to creating a new configuration. On the Project settings page click the **Create template** link and proceed with configuring general settings, VCS settings and build steps.

Creating Build Configuration From Template

To create a build configuration based on the template, on the Project settings page locate the desired template and click **Edit**. Then, click the **Create Build Configuration From Template** button and specify the name for new configuration. The settings set up in the template cannot be edited in a configuration created from this template. If you want to change them, modify them in the template's settings. However, note that this will affect all configurations based on this template.

Creating Maven Build Configuration

To create new build configuration automatically from the `POM.xml` file, on the Project settings page click the **Create Maven build configuration** link and 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 `pom.xml` file in a repository or upload `pom.xml` file from your local working PC. The URL to the POM file should 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.

> From the provided POM file TeamCity reads the name and the VCS root URL parameters for the new build configuration. For non-Maven projects, if there's no VCS root URL provided in the `pom.xml`, then the process will be aborted with error.

When the 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 delimiter you can use either 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, please 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:starteam:starteam://host.com/trunk/pom.xml`" URL will valid in the "`starteam://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 as 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
  ```

See also:

Administrator's Guide: Configuring Dependencies | Configuring Build Parameters | Configuring VCS Settings

Configuring General Settings

When creating a build configuration, on the first page of the wizard specify the following options:

- Name and Description
- Build Number Format
- Build Counter
- Artifact Paths
- Build Options
  - Hanging Build Detection
  - Enable Status Widget
    - HTML Status Widget
  - Limit the number of simultaneously running builds

Name and Description

Use these fields to provide the name for the build configuration and optional description. Note that the **Name** field should not contain special characters.

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
{0}	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.{0}.%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 please refer to the Build Script Interaction with TeamCity page.

Build Counter

Use the **Build counter** field to 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

To learn what is called a build artifact in TeamCity, please refer to the Build Artifact concept page. On the **General Settings** page of the build configuration you can specify artifacts of a build using comma-, or newline- separated values of the format:

```
file_name|directory_name|wildcard [ => target_directory|target_archive ]
```

The format contains:

- `file_name` — to publish the file. The file 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 ("*" and "**" wildcards are only 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.
- `target_archive` — the path to the archive to be created by TeamCity by packing build artifacts determined in the left part of the pattern. TeamCity treats the right part of the pattern as `target_archive` whenever it ends with a supported archive extension, i.e. `.zip`, `.jar`, `.tar.gz`, or `.tgz`.

The names can be paths relative to the build checkout directory or absolute paths. The usage of absolute paths is discouraged, please try to use path relative to the build checkout directory.

An optional part starting with => symbols and followed by the target directory name can be used to publish the files into the specified target directory. If target directory is omitted the files are published in the root of the build artifacts. You can use "." (dot) as reference to the build checkout directory.

> 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

Specify additional options for the builds of this build configuration.

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 did not send any messages since the estimation 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 will have enough information to estimate the average run time.

Enable Status Widget

This option enables retrieving the status and basic details of the last build in the build configuration without requiring any user authentication. Please note that this also allows to get status of any specific build in the build configuration (however, builds cannot be listed and no other information except for the build status (success/failure/internal error/cancelled) is available).

The status can be retrieved via the HTML status widget described below, or via 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 snipet can be included into an external web page and will display current build configuration status.
By means of the status widget, the following build process information is provided:

- The latest build results,
- Build ID,
- Build data,
- Link to the latest build artifacts.
  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's 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 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=<buildConfigurationId>
```

It is also possible to show the status of all a project's build configurations by replacing `"&buildTypeId=<buildConfigurationId>"` with `"&projectId=<projectId>"`. 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 RSS feed for the build configuration.

Limit the number of simultaneously running builds

Specify the number of builds of the same configuration that can run simultaneously on all agents. This option helps avoid the situation, when all of the agents are busy with the builds of a single project. Enter 0 to allow an unlimited number of builds to run simultaneously.

See also:

> **Concepts**: Build Configuration

Configuring VCS Settings

A Version Control System (VCS) is a system for tracking the revisions of the project source files. It is also known as SCM (source code management) or a revision control system. The following VCSs are supported by TeamCity out-of-the-box: ClearCase, CVS, Git, Mercurial, Perforce, StarTeam, Subversion, Team Foundation Server, SourceGear Vault, Visual SourceSafe.

Setting up VCS parameters is one of the mandatory steps during creating a build configuration in TeamCity.
On the 2nd page of the build configuration (**Version Control Settings** page) do the following :

1. Attach an existing VCS root to your build configuration, or create a new one to be attached. This is the main part of VCS parameters setup; a VCS Root is a description of a version control system where project sources are located. Learn more about VCS Roots and configuration details here.
2. When VCS root is attached, specify the checkout rules for it — specifying checkout rules provides advanced possibilities to control sources checkout. With the rules you can exclude and/or map paths to a different location on the Build Agent during checkout.
3. Define how project sources reach an agent via VCS Checkout Mode.
4. Additionally, you can add a label into the version control system for the sources used for a particular build by means of VCS Labeling feature.

See also:

> **Administrator's Guide**: Configuring VCS Roots | VCS Checkout Rules | VCS Checkout Mode | VCS Labeling

Configuring VCS Roots

VCS Roots in TeamCity

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 you can add as many VCS Roots to it as you need. VCS roots are set up on the **Version Control Settings** page of a build configuration settings. Please, refer to the following pages for VCS-specific configuration details:

- ClearCase
- CVS
- Git (JetBrains)
- Mercurial
- Perforce
- StarTeam
- Subversion
- Team Foundation Server

- SourceGear Vault
- Visual SourceSafe

When 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 new changes in any of the build configuration's VCS roots, which suits for most cases. When a build starts, TeamCity gets the changed files from the version control and applies the changes into the Build Checkout Directory.

> ⚠ 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)

> ℹ Note that you can manage all VCS root configured within a project, or in TeamCity:
>
> - From **Administration** | **VCS Roots** page: you can view all VCS roots configured in TeamCity, edit/delete them, and attach/detach to build configurations. Only System Administrators can edit all VCS roots on this page. Project Administrators can edit roots of the projects where they have this role.
> - From **Administration** | **Projects** | **<Project_Name> settings** | **VCS Roots** tab: you can view all VCS roots configured within the project and create/edit/delete/detach them.

Sharing VCS Roots Between Projects

By default, VCS roots can be used by all build configurations of the project. However, they can also be "shared", i.e. they can be used by different projects in TeamCity.
To share a VCS root enable the **Make this VCS root available to all of the projects** option on the root's settings page.

Sharing VCS roots saves the administrator's time and the hassle of creating and maintaining identical VCS roots in the repository.

If someone attempts to modify a VCS root that is shared, TeamCity will issue a warning that the changes to the VCS root could potentially affect more that one project. The user is then prompted to either save the changes and apply them to all the projects using the VCS root, or to make a copy of the VCS root for use by either a specific build configuration or project.

> ⚠ To be able to share VCS Roots, you need to have **Change shared VCS root** permission which can be assigned to you by the TeamCity System Administrator.

ClearCase

This page contains description of the fields and options available when setting up VCS roots using the ClearCase Version Control System:

- Initial Setup
- ClearCase Settings
- Changes Checking Interval
- VCS Root Sharing

Initial Setup

First of all you need to have an installed ClearCase client on TeamCity server to make TeamCity CleaCase integration work. You also need to create a ClearCase view on TeamCity server machine (no matter whether you plan to use server-side or agent-side checkout). This view will be used for collecting changes in ClearCase VCS roots and for checkout in case of server-side checkout mode. In case of agent-side checkout mode config spec of this view will be also used to automatically create views on agents.

> ℹ If you plan to use agent-side checkout mode, make sure ClearCase client (cleartool) is also installed on agents and properly configured, i.e. the tool must be in system paths and have access to the ClearCase Registry.

ClearCase Settings

Option	Description
ClearCase view path	Local path on the TeamCity server machine to the ClearCase view created during the initial setup. Snapshot view is preferred as there is no benefit in using dynamic view with TeamCity. Also, dynamic views are not supported for agent-side checkout (TW-21545). ⚠ Do not use the view you are currently working with. TeamCity calls update procedure *before* checking for changes and building patch, and thus it might lead to problems with checking in changes.
Relative path within the view	Path relative to the "ClearCase view path" that limits the sources to watch and checkout for the build.
Branches	Branches are used in "-branch" parameter for "lshistory" command to significally improve its performance. You can either let TeamCity detect the needed branches automatically (via view's config spec) or specify your own list of needed branches. Press "Detect now" button to see the branches those TeamCity automatically detects. You can also choose "use custom" option and leave the text field blank - then "-branch" parameter will not be used for "lshistory" command at all. If you specify or TeamCity detects several branches, then "lshistory" will be called for every branch and all results will be merged.
Use ClearCase	Use the drop-down menu to select either UCM or BASE.
Global labeling	Check this option if you want to use global labels
Global labels VOB	Pathname of the VOB tag (whether or not the VOB is mounted) or of any file system object within the VOB (if the VOB is mounted)

Make sure that the user that runs the TeamCity server process is also a ClearCase view owner.

Changes Checking Interval

Option	Description
Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. ⚠ Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

See also:

Administrator's Guide: VCS Checkout Mode

CVS

This page contains descriptions of fields and options available when setting up a VCS root using CVS. Depending on the selected access method, the page shows different fields, that help you to easily define the CVS settings:

Common Options

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: • pserver • ext • ssh • local
Checkout Options	Click one of the radio buttons to define the way CVS will fill in and update the working directory. The following options are available: • Checkout HEAD revision • Checkout from branch • Checkout by Tag
Quiet period	Since there are no atomic commits in CVS with help of this setting you can instruct TeamCity not to take (detect) change if specified period of time is not passed since the change date. This should avoid situations when one commit is shown as two different changes in the TeamCity web UI.

PServer Protocol Settings

Option	Description
CVS Password	Click this radio button if you want to access the CVS repository by entering password.
Password File Path	Click this radio button to specify the path to the `.cvspass` file.
Connection Timeout	Specify connection timeout.
Proxy Settings	Specify the proxy settings.
Use proxy	Check this option if you want to use a proxy, and select the desired protocol from the drop-down list.
Proxy Host	Specify the name of the proxy.
Proxy Port	Specify the port number.
Login	Specify the login name.
Password	Specify the password.

Ext Protocol Settings

Option	Description
Path to external rsh	Specify the path to the rsh program used to connect to the repository.
Path to private key file	Specify the path to the file that contains user authentication information.

Additional parameters	Enter any required rsh parameters that will be passed to the program. Multiple parameters are separated by spaces.

SSH Protocol Settings (internal implementation)

Option	Description
SSH version	Select a supported version of SSH.
SSH port	Specify SSH port number.
SSH Password	Click this radio button if you want to authenticate via an SSH password.
Private Key	Click this radio button to use private key authentication. In this case specify the path to the private key file.
SSH Proxy Settings	See proxy options above.

Local CVS Settings

Option	Description
Path to the CVS Client	Specify the path to the local CVS client.
Server Command	Type the server command. The server command is the command which makes the CVS client to work in server mode. Default value is `'server'`

Advanced Options

Option	Description
Use GZIP compression	Check this option to apply gzip compression to the data passed between CVS server and CVS client.
Send all environment variables to the CVS server	Check this option to send environment variables to the server for the sake of compatibility with certain servers.
History Command Supported	Check this option to use the history file on server to search for newly committed changes. Enable this option to improve CVS support performance. By default, the option is checked off because not every CVS server supports keeping a history log file.

Changes Checking Interval

Option	Description
Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. ⚠ Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

Git (JetBrains)

This page contains description of the fields and options available when setting up VCS roots using the Git Version Control System. The VCS is visible as "Git (JetBrains)" in VCS chooser to eliminate confusion with third-party plugin if third-party plugin is installed and as "Git" otherwise.

Important Notes

- **Remote run** is supported in IntelliJ IDEA and Eclipse plugins, pre-tested commit is not yet supported in any of the IDE plugins.
- Initial Git checkout may take significant time (sometimes hours), depending on the size of your project history, because the whole project history is downloaded during the initial checkout.

General Settings

Option	Description
Fetch URL	The URL of the remote Git repository.
Push URL	The URL of the target remote Git repository.
Ref name	The name of the git ref (e.g. "master", "refs/tags/v5.1"), or left this field blank to use "master" branch.
Clone repository to	The path to a directory on TeamCity server where a bare repository should be created. Leave it blank to use default path.
User Name Style	Changing user name 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	Custom username used for labeling

Feature Branches in TeamCity 7.1
Starting with TeamCity 7.1 you can configure the branches you want to monitor in a build configurations right in the VCS root:

Branch Specification	In this area list all the branches you want to be monitored for changes. The syntax is similar to checkout rules: `+|-:branch_name`, where `branch_name` is specific to the VCS e.g. `refs/heads/` in Git (with optional * placeholder). Note that only one asterisk is allowed, and each rule has to start with a new line. • 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 TeamCity interface you'll see `feature1` only as a branch name. The short name of the branch is determined as: • 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 VCS trigger and a change is found in some branch, TeamCity will trigger a build in this branch.
Branch Name	Use this field to define a so-called default branch. Default branch is used in situations when branch name was not specified. For example, if someone clicks on a **Run** button TeamCity will create build in default branch. Note that parameter references are allowed in branch specification as well.

The following protocols are supported for the server-side checkout mode:

- ssh: (e.g. ssh://git.somwhere.org/repos/test.git, ssh://git@git.somwhereElse.org/repos/test.git, scp-like syntax: git@git.somwhere.org:repos/test.git)

  Be Careful
  Scp-like syntax requires a colon after hostname, while 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 a 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.
Default Private Key	Valid only for SSH protocol and applicable to both fetch and push urls. Uses mapping specified in `<USER_HOME>\.ssh\config` if that file exists. Specify a valid username if there is no username in the clone URL. The username specified here overrides username from the URL.
Password	Specify username and password to be used to clone the repository. (Supported only for server-side checkout, see TW-18711)
Private Key	Valid only for SSH protocol and applicable to both fetch and push urls. (Supported only for server-side checkout, see TW-18449)

Agent Settings

Please note that agent-side checkout has limited support for SSH. The only supported authentication method is "Default Private Key".
Also you should set "StrictHostKeyChecking" to "no" in `<USER_HOME>\.ssh\config`, or add keys of remote hosts to the Known-Hosts database manually.
If you plan to use agent-side checkout, you need to have Git 1.6.4+ installed on the agents.

Option	Description
Path to git	Provide path to a git executable to be used on agent. When set to `%env.TEAMCITY_GIT_PATH%` - will use automatically detected git, see Git executable on the agent for details
Clean Policy/Clean Files Policy	Specify here when "git clean" command is run on agent, and which files should be removed.

Git executable on the agent

Path to git executable can be configured in the agent properties by setting the value of an environment variable `TEAMCITY_GIT_PATH`.

If path to git is not configured, git-plugin tries to detect installed git on the start of the agent. It first tries to run git at 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 at any of these locations, try to run git accessible from the `$PATH`.

If compatible git (1.6.4+) is found - it is reported in the `TEAMCITY_GIT_PATH` environment variable. This variable can be used in **Path to git** field in VCS root settings. As a result, configuration with such a VCS root will run only on agents where git was detected or specified in agent properties.

Changes Checking Interval

Option	Description

Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. ⚠ Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

Internal Properties

For Git VCS it is possible to configure the following internal properties:

Property	Default	Description
teamcity.git.idle.timeout.seconds	600	Idle timeout for communication with remote repository. If no data were sent or received during this timeout - plugin throws a timeout error
teamcity.git.fetch.timeout	600	Fetch process idle timeout. Fetch process reports what it is doing in the stdout and is killed if there is no output during this timeout.
teamcity.git.fetch.separate.process	true	Should TeamCity run git fetch in a separate process
teamcity.git.fetch.process.max.memory	512M	Value of JVM -Xmx parameter for separate fetch process
teamcity.server.git.gc.enabled	false	Whether TeamCity should run `git gc` during server cleanup (native git is used)
teamcity.server.git.executable.path	git	Path to native git executable on the server
teamcity.server.git.gc.quota.minutes	60	Maximum amount of time to run `git gc`
teamcity.git.stream.file.threshold.mb	128	Threshold in megabytes after which JGit uses streams to inflate objects. Increase it if you have big binary files in the repository and see symptoms described in TW-14947
teamcity.git.mirror.expiration.timeout.days	7	Number of days after which unused clone of repository will be removed from the server machine. Repository considered unused, if there were no TeamCity operations on this repository, like checking for changes or getting current version. These operations are quite frequent, so 7 days is reasonable high value.
teamcity.git.commit.debug.info	false	Log additional debug info on each found commit

Agent configuration for Git:

Property	Default	Description
teamcity.git.use.native.ssh	false	When checkout on agent: whether TeamCity should use native SSH implementation.
teamcity.git.use.local.mirrors	false	When checkout on agent: whether TeamCity should clone to local agent mirror first and then clone to working directory from this local mirror. This option speed-ups clean checkout, because only build working directory is cleaned. Also if single root is used in several build configurations clone will be faster.
teamcity.git.use.shallow.clone	false	When checkout on agent: run fetch with option '--depth=1' if agent uses local mirrors. This property can be set either in agent properties or as an parameter in build configuration.

Limitations

- When using checkout on agent, limited subset of checkout rules is supported, because Git cannot clone a subdirectory of repository. You can only map whole repository to a specific directory using following checkout rule `+:.=>subdir`. The rest checkout rules are not supported.

Known Issues

- Tagging is not supported over HTTP protocol
- java.lang.OutOfMemoryError while fetch repository. Usually happens when there are big files in repository. By default TeamCity runs fetch in a separate process, to run fetch in the server process set an internal property `teamcity.git.fetch.separate.process` to `false`.
- Teamcity ran 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.
- checkout on agent using ssh protocol can be slow due to java SSH implementation (see TW-14598 for details). To use native SSH implementation set `teamcity.git.use.native.ssh` to `true`.
- inflation using streams in JGit prevents `OutOfMemoryError` but can be time consuming (see related thread at jgit-dev for details and TW-14947 issue related to the problem). If you meet conditions similar to described in the issue - try to increase `teamcity.git.stream.file.threshold.mb`. Additionally it is recommended to increase 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

Mercurial

TeamCity uses typical Mercurial command line client: hg command. Supported version of hg is 1.5.2+.

Mercurial should be installed in the server machine, and, if agent-side checkout is used, on the agents.

Note that:

- **Remote Run** form IDE is not supported.
- Checkout rules for agent-side checkout are not supported except for `.=><target_dir>` rule.

Changes Checking Interval

Option	Description
Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

Path to hg executable detection

When an agent starts, hg-plugin detects installed mercurial on the agent machine. Plugin tries to run `hg version` command using the path specified by `teamcity.hg.agent.path` parameter. You can change this parameter in buildAgent.properties. If this parameter is not set, plugin uses hg as a path to command, assuming it is somewhere in the $PATH. If command executed successfully and mercurial has an appropriate version (1.5.2+), then hg-plugin reports path to hg in the `teamcity.hg.agent.path` parameter. During the build plugin uses hg specified in the **HG command path** field of a VCS root settings, to use detected hg you should put `%teamcity.hg.agent.path%` in this field. Configurations with such settings will be run only on agents which report path to hg.

A server side of plugin first checks value of internal property `teamcity.hg.server.path` and if property is set, its value is used. Second, plugin tries to use path from the settings of VCS root: if path is equal to %teamcity.hg.agent.path% - it uses `hg` as a path, otherwise use a value from the root.

Feature Branches in TeamCity 7.1
Starting with TeamCity 7.1 you can configure the branches you want to monitor in a build configurations right in the VCS root:

Branch Specification	In this area list all the branches you want to be monitored for changes. The syntax is similar to checkout rules: `+|-:branch_name`, where `branch_name` is specific to the VCS e.g. `refs/heads/` in Git (with optional * placeholder). Note that only one asterisk is allowed, and each rule has to start with a new line. • 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 TeamCity interface you'll see `feature1` only as a branch name. The short name of the branch is determined as: • 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 VCS trigger and a change is found in some branch, TeamCity will trigger a build in this branch.
Branch Name	Use this field to define a so-called default branch. Default branch is used in situations when branch name was not specified. For example, if someone clicks on a **Run** button TeamCity will create build in default branch. Note that parameter references are allowed in branch specification as well.

Internal Properties

Server-side internal properties:

Property	Default	Description
teamcity.hg.pull.timeout.seconds	3600	Maximum time in seconds for pull operation to run
teamcity.hg.server.path	hg	Path to hg executable on the server (see Path to hg executable detection for the details).

Agent configuration for Mercurial:

Property	Default	Description
teamcity.hg.use.local.mirrors	false	When checkout on agent: whether TeamCity should clone to local agent mirror first and then clone to working directory from this local mirror. This option speed-ups clean checkout, because only build working directory is cleaned. Also if single root is used in several build configurations clone will be faster.
teamcity.hg.pull.timeout.seconds	3600	Maximum time in seconds for pull operation to run
teamcity.hg.agent.path	hg	Path to hg executable on the agent (see 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:

If you plan to use agent-side checkout mode, note that Perforce client must be installed on the agents, and path to p4 executable must be added to the PATH environment variable.

P4 Connection Settings

Option	Description
Port	Specify the Perforce server address. The format is host:port.
Client	Click this radio button to directly specify the client workspace. The workspace should already be created by Perforce client applications like P4V or P4Win. Only mapping rules are used from the configured client workspace. The client name is ignored. **Performance impact** When this option is used, internal TeamCity source caching on the server side is disabled, which may worsen the performance of clean checkouts. For maximum performance, we recommend using **Client Mapping** option (see below)

Client Mapping	Click this radio button to specify the mapping of the depot to the client computer.
	If you have **Client mapping** selected, TeamCity handles file separators according to OS/platform of a build agent where a build is run. To enforce specific line separator for all build agents, use **Client** having `LineEnd` option specified in Perforce instead of **Client mapping**. Alternatively you can add an agent requirement to run builds only on specific platform.
	Tip
	Use **team-city-agent** instead of the client name in the mapping.
	Example:
	`//depot/MPS/... //team-city-agent/...`
	`//depot/MPS/lib/tools/... //team-city-agent/tools/...`

User	Specify the user login name.
Password	Specify the password.
Ticket-based authentication	Check this option to enable ticket-based authentication.

Checkout Settings

Option	Description
Label to checkout	**Since TeamCity 7.1**. If you need to check out sources not with the latest revision, but with specific Perforce label (with selective changes), you can specify this label here. For instance, this can be useful to produce a milestone/release build, or a reproduce build. If the field is left blank, the latest sources revision will be used for checkout. It is recommended to use agent-side checkout if you use symbolic labels. With server-side checkout on label, TeamCity will perform full checkout.
Workspace options (checkout on agent)	If needed, you can set here the following options for `p4 client` command: `Options`, `SubmitOptions`, and `LineEnd`.

Other Settings

Path to P4 executable	Specify the path to the Perforce command-line client: `p4.exe` file).
Charset	Select the character set used on the client computer.

Support UTF-16 encoding	Enable this option if you have UTF-16 files in your project.

When agent-side checkout is used, TeamCity creates a Perforce workspace for each checkout directory/VCS root. These workspaces are automatically created when necessary and are automatically deleted after some idle time. It is possible to customize the name generated by TeamCity: add `teamcity.perforce.workspace.prefix` configuration parameter at the **Build Parameters** page with the prefix in the value.

Changes Checking Interval

Option	Description
Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. ⚠ Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

See also:

Administrator's Guide: VCS Checkout Mode

StarTeam

This page describes the fields and options available when setting up VCS roots using StarTeam:

StarTeam Connection Settings

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: • **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: - **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.

Changes Checking Interval

Option	Description
Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. ⚠ Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

Subversion

This page contains descriptions of the fields and options available when setting up VCS roots using Subversion:

You do not need Subversion client to be installed on TeamCity server or agents. TeamCity bundles Java implementation of SVN client (SVNKit).

SVN Connection Settings

Option	Description
URL	Specify the SVN URL that points to the project sources.
User Name	Specify the SVN user name.
Password	Specify the SVN password.
Default Config Directory	Check this option to make this the default configuration directory for the SVN connection.
Configuration Directory	If the **Default Config Directory** option is checked, you must specify the configuration directory.
Externals Support	*Check one of the following options to control the SVN externals processing*
Full support (load changes and checkout)	If selected, TeamCity will check out all configuration's sources (including sources from the externals) and will gather and display information about externals' changes in the **Changes** tab.
Checkout, but ignore changes	If selected, TeamCity will check out the sources from externals but any changes in externals' source files will not be gathered and displayed in the **Changes** tab. You might use this option if you have several SVN externals and do not want to get information about any changes made in the externals' source files.
Ignore externals	If selected, TeamCity will ignore any configured "`svn:externals`" property, and thus TeamCity will not check for changes or check out any source file from the SVN externals.

⚠ Note that if you have anonymous access for some path within SVN, entered username will never be used to authenticate when accessing any of its subfolders. Anonymous access will be used instead. This rule only applies for `svn://` and `http(s)://` protocols.

I.e. if you have a build configuration, which uses a combination of this VCS Root + VCS Checkout Rules referencing a non-restricted path above the restricted one for another build configuration, changes under the restricted path will be ignored *even* if you specify correct username/password for the VCS Root itself.

SSH settings

Option	Description
Private Key File Path	Specify the full path to the file that contains the SSH private key.
Private Key File Password	Enter the password to the SSH private key.
SSH Port	Specify the port that SSH is using.

Checkout on agent settings

Option	Description
Working copy format	Select format of the working copy. Available values for this option are 1.4 (used as default in versions previous to TeamCity 5.0), 1.5 (current default), and 1.6. This option defines format version of Subversion files, which are located in `.svn` directories, when **checkout on agent** mode is used. Specified format is important in two cases: • If you run command-line `svn` commands on the files checked out by TeamCity. For example, if your working copy has version 1.5, you will not be able to use Subversion 1.4 binaries to work with it. • If you use new Subversion features; for example, file-based externals which were added in Subversion 1.6. Thus, unless you set the working copy format to 1.6, the file-based externals will not be available in **checkout on agent** mode.
Revert before update	If the option is selected, then TeamCity always runs "`svn revert`" command before updating sources; that is, it will revert all changes in versioned files located in the working directory. When the option is disabled, and local modifications are detected during the update process, TeamCity runs "`svn revert`" after the update. TeamCity does not delete non-versioned files in working directory during the revert.

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.

Changes Checking Interval

Option	Description
Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. ⚠ Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

Authentication for SVN externals

TeamCity doesn't allow to specify SVN externals authentication parameters explicitly, in user interface. To authenticate on the SVN externals server, the following approaches are used:

- authenticate using same credentials (username/password) as for main repository
- authenticate without explicit username/password. In this case, credentials should be already available to svn process (usually, they stored in subversion configuration directory). So, this require setting correct "Configuration Directory" or "Default Config Directory" option under SVN Connection Settings

When TeamCity has to connect to a SVN external, it uses the following sequence:

- if SVN external URL has the same prefix, as the main repository (there is a match > 20 characters), TeamCity tries main repository credentials first, and if failed, tries to connect without username/password (so they picked up from SVN configuration directory)
- if SVN external URL noticeably differs from the main repository, TeamCity tries to connect without username/password, and if failed, tries using credentials from the main repository

See also:

Administrator's Guide: Configuring VCS Settings | VCS Checkout Mode

Team Foundation Server

This page contains descriptions of the fields and options available when setting up a VCS root for connection to Microsoft Team Foundation Server:

> ⚠ TFS integration is only supported for Windows machines. It is also required to have `Team Explorer` installed and correctly functioning on TeamCity server.
>
> If you want to use agent-side checkout, Team Explorer should also be installed on all the agents which will run the related builds.
>
> See the section for the supported versions.

Option	Description

Team Foundation Server Url	Team Foundation Server URL in the following format:
	`http[s]://<server>:<port>`
	Starting with TFS 2010, the URL can also include sub-path:
	`http[s]://<server>:<port>/<path>`

User Name	Specify user to access Team Foundation Server. This can be a user name or `DOMAIN\UserName` string. Use blank to let TFS select user account that is used to run TeamCity Server (or Agent for agent-side checkout)
Password	Enter the password of the user entered above.
Root	Specify the root using the following format: `$<project name><project catalogue>`

Changes Checking Interval

Option	Description
Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. ⚠ Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

Agent-Side Checkout

Agent-Side checkout is only supported on Windows agent machine (for Linux and Mac agent you may use server-side checkout). It is also required to have `Team Explorer` installed on agent machine.

TeamCity automatically creates TFS workspace for each checkout directory used. The workspace is created on behalf of user account that is specified in the VCS root settings.

Created TFS workspaces are automatically removed after a two-week idle time.

TFS does not allow to have several workspaces on a machine mapped to the same directory. TeamCity TFS agent-side checkout may attempt to remove intersecting workspaces in order to create new workspace that matches specified VCS root and checkout rules. Note, it is unable to remove workspaces created by another user.

SourceGear Vault

SourceGear Vault Version Control System support is implemented as a plugin. Please, refer to the plugin's page for the configuration details.

Visual SourceSafe

This page contains descriptions of the fields and options available when setting up VCS roots using Visual SourceSafe:

- VSS Settings
- Changes Checking Interval

- VCS Root Sharing

⚠ **Notes and Limitations**

- TeamCity supports Microsoft Visual SourceSafe 6.0 and 2005 (*English versions* only!).
- Microsoft Visual SourceSafe only works if the TeamCity server is installed on a computer running a Windows® operating system.
- Make sure the TeamCity server process is run by a user that has permission to access the VSS databases.

TeamCity has the following limitations with Visual SourceSafe:

- Shared (not branched) files cannot be checked out.
- Comments for add and delete operations for file and directories are not shown in VSS 6.0. All such operations will have "No Comment" instead of a real VSS comment. (This limitation is imposed by the VSS API, which makes it impossible to retrieve comments with acceptable performance).
- The timestamps on VSS check in are driven by local time on client computer. Therefore if the time on clients and build server are not synchronized, TeamCity cannot properly order check-ins. There is also problem with timezone, however it was already addressed in VSS client version 2005, when configured properly. To avoid these problems follow the recommendations:
    1. Sync client machines via timeserver: http://support.microsoft.com/kb/131715/EN-US.
    2. Setup timezone for VSS database: Start VSS admin-> Tools-> Options-> TimeZone and pick one.
    3. Use VSS 2005.

VSS Settings

Option	Description
Path to `srcsafe.ini`	The full path of the VSS configuration file `srcsafe.ini` 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 `\\vss-server\share\srcsafe.ini` where vss-server is a server name and share is a name of the shared directory.
Project	Specify the mandatory path to the project tree, starting with `$/`.
User Name	Specify the mandatory name of the user on Visual SourceSafe server.
Password	Enter the password, that corresponds to the user name.

Changes Checking Interval

Option	Description
Checking interval	Select here how often TeamCity should check for VCS changes. By default the global predefined server setting is used, that can be modified at the **Administration** | **Global Settings** page. The interval's time starts being counted as soon as the last VCS server poll is finished. Here you can specify here custom interval for the current VCS root. ⚠ Some public servers can block access if polled frequently.

VCS Root Sharing

Option	Description
VCS Root Sharing	Enable this option to use this VCS root in other projects or build configurations. See Shared VCS Roots for more information.

VCS Checkout Rules

VCS Checkout Rules allow you to exclude version control/repository paths and map paths (copy directories and all their contents) to a different location on the Build Agent during checkout.
The rules affect the changes displayed in the TeamCity for the build and the files checked out for the build. To display changes, but do not trigger a build for a change, use VCS Trigger Rules.

Exclude checkout rules will 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.

To add a checkout rule, click the **edit checkout rules** link on the build configuration's **Version Control Settings** page and a pop-up window will appear where you can enter the rule.

The general syntax of a single checkout rule is as follows:

```
+|- : VCSPath [=> AgentPath]
```

When entering rules please note the following:

- To enter multiple rules, each rule should be entered on a separate line.
- For each file the most specific rule will apply if the file is included, regardless of what order the rules are listed in.
- If you don't enter an operator it will default to `+:`

Rules can be used to perform the following operations:

Syntax	Explanation
`+:.=>Path`	Checks out the root into `Path` directory
`-:PathName`	Excludes `PathName` (note: the path must be a directory and not a filename)
`+:VCSPath=>.`	Maps the `VCSPath` from the VCS to the Build Agent's default work directory
`VCSPath=>NewAgentPath`	Maps the `VCSPath` from the VCS to the `NewAgentPath` directory on the Build Agent
`+:VCSPath`	Maps the `VCSPath` from the VCS to the same-named directory (`VCSPath`) on the Build Agent

An example with three VCS checkout rules:

```
-:src/help
+:src=>production/sources
+:src/samples=>./samples
```

In the above example, the first rule excludes the `src/help` directory and its contents from checkout. The third rule is more specific than the second rule and maps the `scr/samples` path to the `samples` path in the Build Agent's default work directory. The second rule maps the contents of the `scr` path to the `production/sources` on the build agent, except `src/help` which was excluded by the first rule and `scr/samples` which was mapped to a different location by the third rule.

See also:

Administrator's Guide: VCS Checkout Mode

VCS Checkout Mode

The VCS Checkout mode is a configuration that sets how project sources reach an agent.
Please note that this mode affects only sources checkout. Current revision and changes data retrieving logic is executed by the TeamCity server and thus TeamCity server should have access to VCS server in any mode.

Agents do not need any additional software for automatic checkout modes.

TeamCity has three different VCS checkout modes:

Checkout mode	Description

Automatically on server	This is the default setting. The TeamCity server will export the sources and pass them to an agent before each build. Since sources are exported rather than checked out, no administrative data is stored in the file system and version control operations (like check in or label or update) can not be preformed from the agent. TeamCity optimizes communications with the VCS servers by caching the sources and retrieving from the VCS server only the necessary changes (read more on that). Unless clean checkout is performed, server sends to the agent incremental patches to update only the files changed since the last build on the agent in the given checkout directory. **Note about usage** • Server side checkout simplifies administration overhead. Using this checkout mode you need to install VCS client software on the server only (applicable to Perforce, Mercurial, TFS, Clearcase, VSS). Network access to VCS repository can also be opened to the server only. Thus, if you want to control who has access to the source repositories, server side checkout is usually more effective. • Note that in some cases this checkout mode can lower load produced on VCS repositories, especially if Clean Checkout is performed often, due to the caching of clean patches by the server. • Note that in server checkout mode the administration directories (like `.svn`, `CVS`) are not created on the agent.
Automatically on agent	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 (.svn, CVS, .git); that is, the build script can perform VCS operations (like check-ins into the version control) — in this case please ensure the build script uses necessary credentials for the check-in. **Note** • Agent checkout is usually more effective with regard to data transfers and VCS server communications. As long as agent side checkout is just a source update or checkout, it creates necessary administration directories (like `.svn`, `CVS`), 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.) should be configured on each machine using agent-side checkout. • "Exclude" VCS Checkout Rules in most cases cannot improve agent checkout performance because an agent checks out the entire top-level directory included into a build, then deletes the files that were excluded. Perforce and TFS are exceptions to the rule, because before performing checkout, specific client mapping(Perforce)/workspace(TFS) is created based on checkout rules. • "Exclude" VCS Checkout Rules are not supported for Git when using checkout on agent due to Git limitations. • Certain version controls can provide additional options when agent-side checkout is used. For example, Subversion.
Do not check out files automatically	TeamCity will not check out any sources. The build script should 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.

See also:

Administrator's Guide: VCS Checkout Rules

VCS Labeling

TeamCity can optionally add a label into the version control system for the sources used for a particular build. This is useful if you need to collect all of the sources for a specific build in order to recreate it. You can choose to apply the VCS label for all builds or only for successful ones when configuring **VCS Settings**, or label a particular finished build manually.

To label a finished build manually:

1. Open **Changes** tab of the build results page.
2. Click **Label this build sources** link.
3. In a dialog that appears, type the desired label name and select VCS roots to apply the label to.

The labeling process takes place after the build finish and doesn't affect the build status. If the label fails it will not change the build's status.

VCS labeling is supported for ClearCase, CVS, Perforce, StarTeam, Subversion, Mercurial, Git and Team Foundation Server.

- Make sure the credentials you specify allow write access to the sources repository.
- "Moving" labels (label with the same name for different builds e.g. "SNAPSHOT") are currently supported only for CVS.
- Note that if you change VCS settings of a build configuration they will be used for labeling only in the new builds. Manual "Label this build sources" action available form build's Changes tab uses the settings actual for the build.

Subversion

Subversion VCS roots need additional configuration for VCS labeling to work. The labeling rules need to be specified to define the SVN repository structure.

Labeling rules are specified as newline-delimited rules each of uses 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`, then no label will be created.
The path `tagDirectoryRepositoryPath` should already exist in the repository.
If `tagDirectoryRepositoryPath` directory already contains 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
```

VCS Labeling Failure Notifications

TeamCity starts the VCS labeling process, when the build is finished. Thus VCS Labeling does not affect the build status, and therefore is not a standard notification event. However, TeamCity allows notifying about labeling failure.
If the labeling process failed due to some reason, TeamCity sends notification to all users, that have subscribed for notification about failed builds of the current build configuration.

Configuring Build Steps

When creating a build configuration it is important to configure the sequence of build steps to be executed. Each build step is represented by a build runner and provides integration with a specific build or test tool. You can add as many build steps to your build configuration as needed. For example, call a NAnt script before compiling VS solutions.

Build steps are invoked sequentially.
In TeamCity 7.0 if a build step fails the rest are not executed.
A build step failure can be caused by:

- A build process has returned exit code other than 0, and **Fail build if** condition "build process exit code is not zero" is enabled in the General Settings of build configuration. (This behaviour doesn't apply to NUnit, MSTest, FxCop, Duplicates (.NET) runners).

⚠ If you have enabled "Fail build if at least one test failed" condition in the General Settings of build configuration, TeamCity will continue executing build steps that go after test runner, but the build will be marked as failed.

In TeamCity 7.1: You can specify step execution policy via "Execute step" option:

- **Only if previous steps were successful**: select to keep the behavior as described above.
- **Even if some of previous steps failed**: select to make TeamCity execute this step regardless of the status of previous steps.
- **Always, even if build stop command was issued**: select to ensure this step is always executed, even if build was canceled by a user.

In the end, the build gets merged status and the output goes into the same build log.

- You can copy a build step from one build configuration to another from the original build configuration settings page.
- You can reorder build steps as needed. Note, that if you have a build configuration inherited from a template, you cannot reorder inherited build steps. However, you can insert custom build steps (not inherited) at any place and in any order, even before or between inherited build steps. Inherited build steps can be reordered in the original template only.
- You can disable a build step temporarily or permanently, even if it is inherited from a build configuration template.

For the details on configuring individual build steps, refer to:

- .NET Process Runner
- Ant
- Command Line
- Duplicates Finder (.NET)
- Duplicates Finder (Java)
- FxCop
- Gradle
- Inspections
- Inspections (.NET)
- IntelliJ IDEA Project
- Maven
- MSBuild
- MSpec
- MSTest
- NAnt
- NuGet
- NUnit
- PowerShell
- Rake
- Visual Studio (sln)
- Visual Studio 2003
- Ipr (deprecated)
- Adding Build Features
- Xcode Project

See also:

Concepts: Build Runner

.NET Process Runner

.NET Process Runner is able to run any .NET assembly under selected .NET Framework version and platform, optionally with .NET code coverage.
For example, you can use it to run xUnit, Gallio or other .NET tests, for which there is no dedicated build runner.

To configure .NET process runner:

1. Specify path to .NET executable (for example to xUnit console) and commandline parameters to be passed to it.
2. Specify .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 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.

3. If needed, add code coverage. To learn about configuring code coverage options, please refer to the corresponding page.

Note, that you don't need to write any additional build scripts.

See also:

> **Administrator's Guide**: Configuring .NET Code Coverage

Ant

In this section:

- Support for Running Parallel Tests
- Ant Runner Settings
  - Ant Parameters
  - Java Parameters
  - Test parameters
  - Code Coverage

Support for Running Parallel Tests

By using the `<parallel>` tag in your Ant script it is possible to have the JUnit and TestNG tasks run in parallel. TeamCity supports this and should concurrently log the parallel processes correctly.

Ant Runner Settings

Ant Parameters

Option	Description
Path to build.xml file	If you choose the option, you can type the path to a Ant build script file of the project. The path is relative to the project root directory. Alternatively, click to choose the file using VCS repository browser (Currently VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)
Build file content	If you choose this option, click the link **Type build file content,** and type source code of your custom build file in the text area. Note that the text area is resizeable. Use **Hide** link to close the text area.
Working directory	Specify the build working directory.
Targets	Use this text field to specify valid Ant targets as a list of space-separated strings. If this field is left empty, default target specified in the build script file will be run.
Ant home path	Specify path to the distributive of your custom Ant. You do not need to specify this parameter, if you are going to use Ant distributive that comes bundled with TeamCity (Ant 1.8). Please note, that you should use Ant 1.7 if you want to run JUnit4 tests in your builds.

Additional Ant command line parameters	Optionally, specify additional command line parameters as a space-separated list.

Java Parameters

Option	Description
JDK home path	Use this field to specify the path to JDK which should be used to run the build (launch Ant). If the field is left blank, the the value of **JAVA_HOME** environment variable is used. (Which can come from Build Configuration settings or agent environment or properties). If JAVA_HOME is not found, TeamCity uses Java home of the build agent process itself. ⚠ Starting with TeamCity 5.1 in many cases agents are able to detect installed Java automatically. Agent will set appropriate environment variables for each detected Java version: JDK_14, JDK_15, JDK_16 and so on. You can specify reference to such environment variable in JDK home path, like: **%env.JDK_15%**. References to parameters can generate an implicit requirement.
JVM command line parameters	You can specify such JVM command line parameters as, for example, *maximum heap size* or parameters enabling *remote debugging*. These values are passed by the JVM used to run your build. Example: `-Xmx512m -Xms256m`

Test parameters

Tests reordering works the following way: TeamCity provides tests that should be run first (test classes), after that when a JUnit task starts, it checks whether it includes these tests. If at least one test is included, TeamCity generates a new fileset containing included tests only and processes it before all other filesets. It also patches other filesets to exclude tests added to the automatically generated fileset. After that JUnit starts and runs as usual.

Option	Description
Reduce test failure feedback time:	Use following two options to instruct TeamCity to run some tests before others.
Run recently failed tests first	If checked, in the first place TeamCity will run tests failed in previous finished or running builds as well as tests having high failure rate (a so called *blinking* tests)
Run new and modified tests first	If checked, before any other test, TeamCity will run tests added or modified in change lists included in the running build.

> ⚠ If both options are enabled at the same time, tests of the **new and modified tests** group will have higher priority, and will be executed in the first place.

Code Coverage

To learn about configuring code coverage options, please 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. *Option is available if "Executable with parameters" is selected in Run dropdown.*
Command parameters	Specify parameters as a space-separated list. *Option is available if "Executable with parameters" is selected in Run dropdown.*
Custom script	Platform specific script which will be executed as *.cmd file on Windows or as an executable script in Unix-like environments. *Option is available if "Custom script" is selected in Run dropdown.* 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 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 and 2010 solutions.

> In order to launch this runner, .NET Framework 3.5 (or higher) should be installed on an agent where build will run.

In this section:

- Sources
- Duplicate Searcher Settings

Sources

Option	Description
Include	Use newline-delimited Ant-like wildcards relative to 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 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 *namespace specifications* will be recognized as duplicates.
	`NLog.Logger.GetInstance().Log("abcd");`
	`A.Log.Logger.GetInstance().Log("abcd");`

Discard literals	If this option is checked, similar lines of code with different literals will be recognized as duplicates.
	`myStatusBar.SetText("Not Logged In");`
	`myStatusBar.SetText("Logging In...");`

Discard local variables	If this option is checked, similar code fragments with different local variable names will be recognized as duplicates.
	`int a = 5; a += 6;`
	`int b = 5; b += 6;`

Discard class fields name	If this option is checked, the similar code fragments with different field names will be recognized as duplicates.
	`Console.WriteLine(myFoo);`
	`Console.WriteLine(myBar);`
	`... where myFoo and myBar are declared in the containing class`

Discard types	If this option is checked, similar content with different type names will be recognized as duplicates. That includes all possible type references (as shown below):
	`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);`

Ignore duplicates with complexity simpler than	Use this field to specify the lowest level of complexity of code blocks to be taken into consideration when detecting duplicates.
Skip files with opening comment	Enter newline-delimited keywords to exclude files that contain the keyword in a 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".
Debug	Check this option to include debug messages in the build log and publish the file with additional logs (**dotnet-tools-dupfinder.log**) as 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 being built by Maven2.

> ⚠ In order to run inspections for your project you should have either an IntelliJ IDEA project file (.ipr)/project directory (.idea), or Maven2 pom.xml of your project checked into your version control.

In this section:

- IntelliJ IDEA Project Settings
- Unresolved Project Modules and Path Variables
- Project JDKs
- Java Parameters
- Duplicate Finder Settings

IntelliJ IDEA Project Settings

Option	Description
Project file type	To be able to run IntelliJ IDEA duplicates finder engine, TeamCity requires either IntelliJ IDEA project file/directory, or Maven2 pom.xml to be specified here.
Path to the project	Depending on what type of project you have selected in the **Project file type**, specify here: • **For IntelliJ IDEA project**: the path to the project file (.ipr) or path to the project directory (root directory of the project that contains `.idea` folder). • **For Maven2 project**: path to the `pom.xml` file. This information is required by this build runner to understand the structure of the project. ⓘ Specified path should be relative to the checkout directory.
Detect global libraries and module-based JDK in the *.iml files	In IntelliJ IDEA 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 you ensure 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	Click this button to reparse your Maven2/IntelliJ IDEA project and import build settings right from the project, for example the list of JDKs. ⚠ If you update your project settings in IntelliJ IDEA or in the `pom.xml` - add new jdks, libraries, don't forget to update build runner settings by clicking **Check/Reparse Project**.
Working directory	Enter a path to a Build Working Directory, if it differs from the Build Checkout Directory. Optional, specify if differs from the checkout directory.

Unresolved Project Modules and Path Variables

This section is displayed, when an IntelliJ IDEA module file (.iml) referenced from IPR-file:

- cannot be found
- allows you to enter the values of path variables used in the IPR-file.

To refresh values in this section click **Check/Reparse Project**.

Option	Description
<path_variable_name>	This field appears, if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In the **Set value to field**, specify a path to project resources, to be used on different build agents.

Project JDKs

This section provides the list of JDKs detected in the project.

Option	Description
JDK Home	Use this field to specify JDK home for the project. ⚠ When building with the **Ipr** runner, this JDK will be used to compile the sources of corresponding IDEA modules. For **Inspections** and **Duplicate Finder** builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process itself the JDK specified in the `JAVA_HOME` environment variable will be used.

JDK Jar File Patterns	Click this link to open a text area, where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns.
	The default value is used for Linux and Windows operating systems:
	`jre/lib/*.jar`
	For Mac OS X, use the following lines:
	`lib/*.jar`
	`../Classes/*.jar`

IDEA Home	If your project uses the IDEA JDK, specify the location of IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

> ⚠ You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a requirement for the corresponding property.

Java Parameters

Option	Description
JDK home path	Use this field to specify the path to your custom JDK which should be used to run the build. If the field is left blank, the path to JDK Home is read either from the `JAVA_HOME` environment variable on agent computer, or from `env.JAVA_HOME` property specified in the build agent configuration file (buildAgent.properties). If these both values are not specified, TeamCity uses Java home of the build agent process itself.
JVM command line parameters	Specify the desired Java Virtual Machine parameters, such as maximum heap size or parameters that enable remote debugging. These settings are passed to the JVM used to run your build. Example: `-Xmx512m -Xms256m`

Duplicate Finder Settings

Option	Description

<table>
<tr><td>Test sources</td><td>If this option is checked, the test sources will be included in the duplicates analysis.
Tests contain the test data which can be duplicated. Thus, it does not make much sense to verify tests for duplicates. We recommend you not to select this option to avoid too long builds creation.</td></tr>
<tr><td>Detalization level</td><td>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:</td></tr>
<tr><td>Distinguish Variables</td><td>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></td></tr>
<tr><td>Distinguish Fields</td><td>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></td></tr>
</table>

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 duplicates analysis 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.

```
new MyIDE().updateStatus()

new TheirIDE().updateStatus()
```

Distinguish Literals

If this option is checked, similar line of code with different litarels will be considered different If this option is not checked, such lines will be recognized as duplicates.

```
myWatchedLabel.setToolTipText("Not Logged In");
```

```
myWatchedLabel.setToolTipText("Logging In...");
```

Ignore Duplicates with Complexity Simpler 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 Simpler 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.

If you need to restrict the sources scope on which to run the Inspections or Duplicates runner, you can specify additional include / exclude patterns.
Include / exclude patterns are newline-delimited set of rules of the form:

```
[+:|-:]pattern
```

Where 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 reporting possible design, localization, performance, and security improvements.
If you want TeamCity to display FxCop reports, you can either configure corresponding build runner, or import XML reports by means of service messages, if you prefer to run the FxCop tool directly from the script. On this page you will find:

- Description of FxCop build runner settings
- Details on using dedicated service messages.

Please see Supported Platforms and Environments page for the supported FxCop versions list.

FxCop Settings

When a build agent is started, it detects automatically whether FxCop is installed. If FxCop is detected, TeamCity defines agent system property `%system.FxCopRoot%` which is then used as a default value for the **FxCop installation root** parameter of the FxCop build runner settings. If you do not have FxCop installed on a build agent, you still can specify a path to FxCop executable (e.g. you can place FxCop tool to your source control, and check it out with a build sources). This path should be relative to the Build Checkout Directory.

If you want to have **line numbers information** and **"Open in IDE"** feature, you should **run FxCop build on the same machine as your compilation build**, because FxCop requires source code to be present to display links to it.

Option	Description
FxCop installation root (**mandatory parameter**)	Specify the path to the FxCop installation root on an agent machine, or a path to the FxCop executable relative to the Build Checkout Directory. Default value set to "`%system.FxCopRoot%`" means that FxCop is detected automatically on agent machine.
FxCop version	If you have several versions of FxCop installed on your build agents, it is recommended to enter here specific version of FxCop you want to use to run inspections in your build to avoid inconsistency. If you leave default value of the field ('Not Specified'), 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 previous build, thus the number of new problems found will be different from actual state.

FxCop Options

Option	Description
What to inspect	Check one of the following options to specify object to inspect (either particular assemblies or a project)
Assemblies	Enter paths, relative to the Build Checkout Directory and separated by spaces, to inspected assemblies (use ant-like wildcards to select files by mask). FxCop will use default settings to inspect them. Enter exclude wildcards to refine included assemblies list.
Project	Enter the path (relative to the Build Checkout Directory) to previously prepared FxCop project from FxCop GUI
Search referenced assemblies in GAC	Search assemblies, referenced by targets, in Global Assembly Cache
Search referenced assemblies in directories	Search assemblies, referenced by targets, in specified directories separated by spaces
Ignore generated code	New option introduced in FxCop 1.36. Speeds up inspection
Report XSLT file	The path to XSLT transformation file, relative to the Build Checkout Directory or absolute on agent machine. You can use path to detected FxCop on target machine (i.e. "%system.FxCopRoot%/Xml/FxCopReport.xsl"). When **Report XSLT file** option is set, the build runner will apply XSLT transform to FxCop XML output and display resulting HTML to a new tab "FxCop" 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 command line parameters

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 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>']
```

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

In order to run builds with Gradle, you need to have Gradle 0.9-rc-1 or higher installed on all the agent machines that you want the build be run on.
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
- Launching Parameters
- Java Parameters
- Build properties
- Code Coverage

Gradle Parameters

Option	Description
Gradle tasks	Specify Gradle task names separated by space. For example: `:myproject:clean :myproject:build` or `clean build`. 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 is not tested.
Incremental building	TeamCity can make use of Gradle :buildDependents feature. If incremental building checkbox is enabled, TeamCity will detect gradle modules affected by changes in build, and start :buildDependents command for them only. This will cause Gradle to fully build and test only modules affected by changes.
Gradle home path	Specify here the path to the Gradle home directory (parent of bin directory). If not specified TeamCity will use Gradle from an agent's `GRADLE_HOME` 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 above fields. In this case, Gradle specified in **Gradle home path** and the one installed on agent, are ignored.

Launching Parameters

Option	Description
Debug	Selecting the **Log debug messages** check box is equivalent to adding `-d` as Gradle command line parameter.
Stacktrace	Selecting the **Print stacktrace** check box is equivalent to adding `-s` as Gradle command line parameter.

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 **JAVA_HOME** environment variable on agent computer, or from **env.JAVA_HOME** property specified in the build agent configuration file (buildAgent.properties). 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, for example, *maximum heap size* or parameters enabling *remote debugging*. These values are passed by the JVM used to run your build. Example: `-Xmx512m -Xms256m`

Build properties

Teamcity build properties are available in build script via "teamcity" property of the project. This property contains map with all defined system properties (see Defining and Using Build Parameters for details). Following example contains task, that will print all available build properties to the build log (it must be executed by buildserver):

```
task printProperties << {
    teamcity.each { key, val ->
      println "##tc-property name='${key}' value='${val}'"
    }
  }
```

Code Coverage

To learn about configuring code coverage options with IDEA code coverage engine, please refer to the corresponding page.

See also:

> **Administrator's Guide**: IntelliJ IDEA Code Coverage

Inspections

The **Inspections** 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 to:

- Find probable bugs
- Locate "dead" code
- Detect performance issues
- Improve code structure and maintainability
- Ensure the code conforms to guidelines, standards and specifications

Refer to IntelliJ IDEA documentation for more details.

> ⚠ In order to run inspections for your project you should have either an IntelliJ IDEA project file (.ipr)/project directory (.idea), or Maven2 pom.xml of your project checked into your version control. Note, that if you want to use `pom.xml`, you need to open it in IntelliJ IDEA and configure inspection profiles, as described in IntelliJ IDEA documentation. IntelliJ IDEA will save your inspection profiles in corresponding folder, make sure you have it checked into your version control. Then specify the paths to inspection profiles while configuring this runner.

This page contains reference information about the **Inspections** Build Runner fields:

- IntelliJ IDEA Project Settings
- Unresolved Project Modules and Path Variables
- Project JDKs
- Java Parameters
- Inspection Parameters

IntelliJ IDEA Project Settings

Option	Description
Project file type	To be able to run IntelliJ IDEA inspections on your code, TeamCity requires either IntelliJ IDEA project file/directory, or Maven2 pom.xml to be specified here.

Path to the project	Depending on what type of project you have selected in the **Project file type**, specify here: • **For IntelliJ IDEA project**: the path to the project file (.ipr) or path to the project directory (root directory of the project that contains `.idea` folder). • **For Maven2 project**: path to the `pom.xml` file. This information is required by this build runner to understand the structure of the project. Specified path should be relative to the checkout directory.
Detect global libraries and module-based JDK in the *.iml files	*This option is available, if you use IntelliJ IDEA project to run the inspections.* In IntelliJ IDEA 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 you ensure 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	Click this button to reparse your Maven2/IntelliJ IDEA project and import build settings right from the project, for example the list of JDKs. If you update your project settings in IntelliJ IDEA or in the `pom.xml` - add new jdks, libraries, don't forget to update build runner settings by clicking **Check/Reparse Project**.
Working directory	Enter a path to a Build Working Directory, if it differs from the Build Checkout Directory. Optional, specify if differs from the checkout directory.

Unresolved Project Modules and Path Variables

This section is displayed, when an IntelliJ IDEA module file (.iml) referenced from IntelliJ IDEA project file:

- cannot be found
- allows you to enter the values of path variables used in the IPR-file.

To refresh values in this section click **Check/Reparse Project**.

Option	Description
<path_variable_name>	This field appears, if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In the **Set value to field**, specify a path to project resources, to be used on different build agents.

Project JDKs

This section provides the list of JDKs detected in the project.

Option	Description
JDK Home	Use this field to specify JDK home for the project. When building with the **Ipr** runner, this JDK will be used to compile the sources of corresponding IDEA modules. For **Inspections** and **Duplicate Finder** builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process itself the JDK specified in the `JAVA_HOME` environment variable will be used.

JDK Jar File Patterns	Click this link to open a text area, where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns.
	The default value is used for Linux and Windows operating systems:
	`jre/lib/*.jar`
	For Mac OS X, use the following lines:
	`lib/*.jar`
	`../Classes/*.jar`

IDEA Home	If your project uses the IDEA JDK, specify the location of IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

⚠ You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a requirement for the corresponding property.

Java Parameters

Option	Description
JDK home path	Use this field to specify the path to your custom JDK which should be used to run the build. If the field is left blank, the path to JDK Home is read either from the `JAVA_HOME` environment variable on agent computer, or from `env.JAVA_HOME` property specified in the build agent configuration file (buildAgent.properties). If these both values are not specified, TeamCity uses Java home of the build agent process itself.
JVM command line parameters	Specify the desired Java Virtual Machine parameters, for example maximum heap size. These settings are passed to the JVM used to run your build. Example: `-Xmx512m -Xms256m`

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. You need to configure the inspection profile in IntelliJ IDEA-based IDE and then specify it in TeamCity.

Please 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 that are 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 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 profile will be loaded from the file. If the loading fails, 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, inspection runner will fail.
- if neither name nor path is specified 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 shared project profile specified with "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 default project profile will be used.

If you need to restrict the sources scope on which to run the Inspections or Duplicates runner, you can specify additional include / exclude patterns.
Include / exclude patterns are newline-delimited set of rules of the form:

```
[+:|-:]pattern
```

Where 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.

Inspections (.NET)

The **Inspections (.NET)** runner allows you to use the benefits of JetBrains ReSharper Code Analysis feature right in TeamCity.
This runner is intended to detect errors and problems in C#, VB.NET, XAML, XML, ASP.NET, JavaScript, CSS and HTML code.

In this section:

Sources to analyze

Option	Description
Solution file path	Path to **.sln** file. Specified path should be relative to the checkout directory.

Projects filter	Specify project name wildcards to analyze only part of the solution. Leave **blank** to analyze the **whole** solution.
	Separate wildcards with new lines.
	Example:
	`JetBrains.CommandLine.*`
	`*.Common`
	`*.Tests.*`

Agent environment requirements

> ⚠ In order to launch inspection analysis, you should have **.NET Framework 3.5** (or higher) installed on an agent where build will run.

Option	Description
Target Frameworks	This option allows you to handle Visual Studio Multi-Targeting feature. Agent requirement will be created for every checked item. ⚠ **.NET Frameworks client profiles** are not supported as target frameworks

Additional options

Option	Description
Custom settings profile path	Path to file wich contains **ReSharper settings**. File should be created via JetBrains Resharper **6.1 or later**. Specified path should be **relative** to the checkout directory. If specified, this settings layer has top priority, so it overrides ReSharper's build-in settings. **By default**, **build-in** ReSharper's settings layers are applied. For additional information about ReSharper settings system please 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 hidden artifact.

Build before analyze

In order to have adequate inspections execution results it might be nesessary to **build your solution before run analysis**.
This pre-step is especially actual when you use (implicitly or explicitly) **code generation** in your project.

IntelliJ IDEA Project

IntelliJ IDEA Project runner allows you to build a project created in IntelliJ IDEA.
TeamCity versions up to 6.0 had Ipr (deprecated) which is now superseded by IntelliJ IDEA Project runner.

Supported IntelliJ IDEA features

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	• 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	• 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	• 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
Flex	⛔	
Android	⛔	
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	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. ⓘ Specified path should be relative to the checkout directory.
Detect global libraries and module-based JDK in the *.iml files	If this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDKs when saved. This helps you ensure all references will be properly resolved. ⛔ **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	Click to reparse the project and import build settings right from the IDEA project, for example the list of JDKs. ⚠ If you update your project settings in IntelliJ IDEA - add new jdks, libraries, don't forget to update build runner settings by clicking **Check/Reparse Project**.
Working directory	Enter a path to a Build Working Directory, if it differs from the Build Checkout Directory. Optional, specify if differs from the checkout directory.

Unresolved Project Modules and Path Variables

This section is displayed, when an IntelliJ IDEA module file (.iml) referenced from IPR-file:

- cannot be found
- allows you to enter the values of path variables used in the IPR-file.

To refresh values in this section click **Check/Reparse Project**.

Option	Description
<path_variable_name>	This field appears, if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In the **Set value to field**, specify a path to project resources, to be used on different build agents.

Project JDKs

This section provides the list of JDKs detected in the project.

Option	Description
JDK Home	Use this field to specify JDK home for the project. ⚠ When building with the **Ipr** runner, this JDK will be used to compile the sources of corresponding IDEA modules. For **Inspections** and **Duplicate Finder** builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process itself the JDK specified in the `JAVA_HOME` environment variable will be used.

JDK Jar File Patterns	Click this link to open a text area, where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns.
	The default value is used for Linux and Windows operating systems:
	`jre/lib/*.jar`
	For Mac OS X, use the following lines:
	`lib/*.jar`
	`../Classes/*.jar`

IDEA Home	If your project uses the IDEA JDK, specify the location of IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

⚠ You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a requirement for the corresponding property.

Java Parameters

Option	Description
JDK home path	Use this field to specify the path to your custom JDK which should be used to run the build. If the field is left blank, the path to JDK Home is read either from the `JAVA_HOME` environment variable on agent computer, or from `env.JAVA_HOME` property specified in the build agent configuration file (buildAgent.properties). If these both values are not specified, TeamCity uses Java home of the build agent process itself.

JVM command line parameters	Specify the desired Java Virtual Machine parameters, such as maximum heap size or parameters that enable remote debugging. These settings are passed to the JVM used to run your build. Example: `-Xmx512m -Xms256m`

Compilation settings

Option	Description
Only compile classes required to build artifacts and execute run configurations	Select whether to compile all classes in the project or only those classes which are required by run configurations or for building artifacts.

Artifacts

Option	Description
Artifacts to Build	Specify here names of the artifacts to be build that are configured in IntelliJ IDEA project.

Run configurations

Option	Description
Run configurations to execute	Specify here names of IntelliJ IDEA run configurations configured in the project to execute inside TeamCity build. Supported configuration types are: JUnit, TestNG and Application. Note, that run configurations specified here should be *shared* 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 new Maven-based build configuration automatically from specified `pom.xml`, and set up a dependency build trigger, if 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 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 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 those managed by TeamCity.
> To avoid this, we recommend not using remote run for build configurations which perform deployment of artifacts.

Below you can find reference information about the **Maven2** Build Runner fields:

- Maven Parameters
- Maven Home
- User Settings
- Java Parameters
- Local Artifact Repository Settings
- Incremental Building
- Code Coverage

Maven Parameters

Option	Description
Goals	In the **Goals** field, specify the sequence of space-separated Maven goals that you want TeamCity to execute. Some Maven goals can use version control systems, and, thus, they may become incompatible with some *VCS checkout modes*. If you want TeamCity to execute such goal: • 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 a goal execution software.
Path to a POM file	Specify path to the POM file relative to the build working directory. By default, the property contains a `pom.xml` 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 `<subdirectory>/pom.xml` is used. Alternatively, click to choose the file using VCS repository browser (Currently VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)
Additional Maven command line parameters	Specify the list of command line parameters. The following parameters are ignored: **-q**, **-f**, **-s** (if **User settings path** is provided)
Working directory	Specify the Build Working Directory, if it differs from the build checkout directory.

Maven Home

In **Maven selection** field choose 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 Maven command line option **-s** or **--settings**. The available options are:

Default	Settings are taken from the default location (chosen by Maven).

User settings path	Enter the path to an alternative user settings file.
Predefined settings	If there are settings filed uploaded to TeamCity server you can select here, which of those files to use. To upload a user settings file to TeamCity, click *Manage settings files*. You can upload Maven user settings files at any time at **Administration** | **Maven Settings** tab. The uploaded files are stored under `<TeamCity Data Directory>/config/_mavenSettings` directory. If necessary, they can be edited right there.

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 as well, TeamCity uses the JDK home on which the build agent process is started.
JVM command line parameters	Specify JVM command line parameters, for example, *maximum heap size* or parameters enabling *remote debugging*. These values are passed by the JVM used to run your build. For example: `-Xmx512m -Xms256m`

Local Artifact Repository Settings

Select **Use own local repository for this build configuration** to isolate this build configuration's artifacts repository from other local repositories.

Incremental Building

Select **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 affected modules and take the result of building the rest of the modules from the previous build we will have the overall result equal to the result of building the whole project from scratch with less effort and faster.

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 affected modules. The preparation phase is to assure there will be no compiler or other errors during the second execution caused by 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.

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 designate custom local repository to each build agent:

- Specify the following property in the `teamcity-agent/conf/buildAgent.properties`:

  ```
  system.maven.repo.local=%system.agent.work.dir%/<subdirectory name>
  ```

  For instance, `%system.agent.work.dir%/m2-repository`
- Run each build agent under different user account.

See also:

Concepts: Build Runner
Administrator's Guide: Maven Artifact Dependency Trigger | Creating Maven Build Configuration

MSBuild

This page contains reference information for the **MSBuild** Build Runner fields.

- General Build Runner Options
- Code Coverage
- Implementation notes

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 2005, 2008 or 2010 solution file please use the Visual Studio (sln) build runner).

General Build Runner Options

Option	Description
Build File Path	Specify the path to the solution to be built, relative to the Build Checkout Directory. For example: `vs-addin\addin\addin.sln` Alternatively, click to choose the file using VCS repository browser (Currently VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)
Working Directory	Specify the path to the build working directory.
MSBuild version	Select the framework you want to run - either .NET Framework (**2.0**, **3.5** or **4.0**) or Mono xbuild.
MSBuild ToolsVersion	Specify here the version of tools that will be used to compile (equivalent to `/toolsversion:` commandline argument). MSBuild supports compilation to older versions, thus you may set **MSBuild version** as 4.0 with **MSBuild ToolsVersion** set to 2.0 to produce .NET 2.0 assemblies while running MSBuild from .NET Framework 4.0. For more information refer to http://msdn.microsoft.com/en-us/library/bb383796(VS.100).aspx
Run platform	From the drop-down list select the desired execution mode on a x64 machine.

Targets	Enter targets separated by spaces. A target is an arbitrary script for your project purposes.
Command line parameters	Specify any additional parameters for `msbuild.exe`
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 <Import> task. If you specified a Visual Studio solution file, it will be called from <MSBuild> task. To disable it, set `teamcity.msbuild.generateWrappingScript` configuration parameter with value `false`.

See also:

> **Concepts**: Build Runner | Build Checkout Directory
> **Administrator's Guide**: NUnit for MSBuild | MSBuild Service Tasks

MSpec

MSpec test runner is designed specifically to run MSpec tests. Note, that to be able 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 `mspec.exe` 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. ℹ 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 .NET assemblies that should excluded from the list of found assemblies to test.
Include specifications	Specify comma or new line 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 an additional commandline parameters for `mspec.exe`.

Code Coverage

To learn about configuring code coverage options, please refer to the Configuring .NET Code Coverage page.

See also:

> **Administrator's Guide**: Configuring .NET Code Coverage

MSTest

This page describes MSTest runner options, for details on MSTest support, please refer to the corresponding page.

Option	Description
Path to MSTest.exe	A path to `MSTest.exe` file. By default Build Agent will autodetect MSTest installation. You may use `%system.MSTest.8.0%`, `%system.MSTest.9.0%` or `%system.MSTest.10.0%` to refer to build agent auto-detected MSTest.exe.
List assembly files	A list of assemblies to run MSTests on. Will be mapped to `/testcontainer:file` argument.
MSTest run configuration file	Specify a MSTest run configuration file to use (`/runconfig:file`).
MSTest metadata	Enter a value for `/testmetadata:file` argument.
Testlist from metadata to run	Every line will be translated into /testlist:line argument.
Test	Names of the tests to run. This option will be translated to the series of `/test:` arguments Check **Add /unique commandline argument** to add `/unique` argument to `MSTest.exe`
Results file	Enter a value for `/resultsfile:file` commandline argument.
Additional commandline parameters	Enter an additional commandline parameters for `MSTest.exe`

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 0.85.

MSBuild Task for NAnt

TeamCity NAnt runner includes a task called `msbuild` that allows NAnt to start MSBuild scripts. TeamCity `msbuild` task for NAnt has the same set of attributes as the NAntContrib package `msbuild` task. The MSBuild build processes started by NAnt will behave exactly as if they were launched by TeamCity MSBuild/SLN2005 build runner (i.e. `NUnit` and/or `NUnitTeamCity` MSBuild tasks will be added to build scripts and logs and error reports will be sent directly to the build server).

> ⚠ `msbuild` task for NAnt makes all build configuration system properties available inside MSBuild script. Note, all property names will have '.' replaced with '_'.
> To disable this, set `false` to `set-teamcity-properties` attribute of the task.

By default, NAnt `msbuild` task checks for current value of NAnt target-framework property to select MSBuild runtime version.
This parameter could be overriden by setting `teamcity_dotnet_tools_version` project property with required .NET Framework version, i.e.

"4.0".

```
...
  <!-- this property enables MSBuild 4.0 -->
  <property name="teamcity_dotnet_tools_version" value="4.0"/>
  <msbuild project="SimpleEcho.v40.proj">
     ...
  </msbuild>
 ...
```

⚠ To pass properties to MSBuild, use the `property` tag instead of explicit properties definition in the command line.

<nunit2> Task for NAnt

To test all of the assemblies without halting on first failed test please use:

```
<target name="build">
       <nunit2 verbose="true" haltonfailure="false" failonerror="true" failonfailureatend="true">
        <formatter type="Plain" />
        <test haltonfailure="false">
          <assemblies>
            <include name="dll1.dll" />
            <include name="dll2.dll" />
          </assemblies>
        </test>
       </nunit2>
</target>
```

⚠ '`failonfailureatend`' attribute is not defined in the original `NUnit2` task from NAnt. Note that this attribute will be ignored if the '`haltonfailure`' attribute is set to '`true`' for either the `nunit2` element or the `test` element.

Below you can find reference information about NAnt Build Runner fields.

General Options

Option	Description
Path to a build file	Specify path relative to the Build Checkout Directory. ✓ Alternatively, click to choose the file using VCS repository browser (Currently VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)
Build file content	Select the option, if you want to use a different build script than the one listed in the settings of the build file. When the option is enabled, you have to type the build script in the text area.
Targets	Specify the names of build targets defined by the build script, separated by spaces.
Working directory	Specify the path to the Build Working Directory. By default, the build working directory is set to the same path as the Build Checkout Directory.
NAnt home	Enter a path to the `NAnt.exe`. ✓ Here you can specify an absolute path to the NAnt.exe file on the agent, or a path relative to the checkout directory. Such relative path allows you to provide particular NAnt.exe file to run a build of the particular build configuration.

Target framework	Sets `-targetframework:` option to `'NAnt'` and generates appropriate agent requirements (*mono-2.0* target framework will require *Mono* system property, *net-2.0 — DotNetFramework2.0* property, and so on). Selecting unsupported in TeamCity framework (*sscli-1.0*, *netcf-1.0*, *netcf-2.0*) won't impose any agent requirements. This option has no effect on framework which used to run `NAnt.exe`. `NAnt.exe` will be launched as ordinary exe file if .NET framework was found, and through mono executable, if not.
Command line parameters	Specify any additional parameters for `NAnt.exe` ✓ TeamCity passes automatically to NAnt all defined system properties, so you do not need to specify all of the properties here via '**-D**' option.
Reduce test failure feedback time	Use following option to instruct TeamCity to run some tests before others.
Run recently failed tests first	If checked, in the first place TeamCity will run tests failed in previous finished or running builds as well as tests having high failure rate (a so called *blinking* tests)

Code Coverage

To learn about configuring code coverage options, please refer to the Configuring .NET Code Coverage page.

See also:

Concepts: Build Runner | Build Checkout Directory | Build Working Directory
Administrator's Guide: Configuring Build Parameters

NuGet

TeamCity integrates with NuGet package manager and allows you to:

- Install and update NuGet packages: NuGet Installer build runner;
- Pack NuGet packages: NuGet Pack build runner;
- Publish packages to a feed of your choice: NuGet Publish build runner. Alternatively you can use TeamCity as NuGet server.
  To start working with NuGet integration, you need to have NuGet.exe Command Line tool installed on build agents: you don't need to do it manually, TeamCity can take care of this task for you.

Installing NuGet to TeamCity agents

You don't need to manually install the required NuGet.exe Command Line tool to all build agents, TeamCity can do this automatically. You need only to choose which NuGet versions you want to be installed on agents at **Administration** | **NuGet Settings** | **NuGet Commandline** tab.
Starting with TeamCity 7.1 you can also upload your own NuGet.CommandLine package instead of downloading it from the public feed.
Note also that installing NuGet on agents results in agents upgrade.

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 native NuGet Server instead of setting up your own repository. To start using TeamCity as NuGet Server you need to enable TeamCity server to be a NuGet server by clicking corresponding button at **Administration** | **NuGet Settings** | **NuGet Server** page. Two different links will be displayed on the same page: for public (with `guestAuth` prefix) and private (with `httpAuth` prefix) feed. If **Public Url** is not available, you need to enable the **Guest user login** in TeamCity at **Administration** | **Global Settings** page.

For an example of set up see blog post: Setting up TeamCity as a native NuGet Server

When you have TeamCity NuGet server enabled:

- You don't need to use NuGet Publish build step (unless you want to publish packages on some public feed).
- You can work with built NuGet packages as with build artifacts: don't forget to specify artifact paths in General Settings of your build configuration.
- You can add TeamCity NuGet server to your repositories in Visual Studio to avoid having to type long URL's in each time you want to

read from a specific package repository (add NuGet repository and specify public URL provided by TeamCity when enabling NuGet server).

> Note that through given links you can get only packages that belong to projects where you have permissions to access artifacts. This relates to both accessing feed from Visual Studio and using console.

Typical Usage Scenarios

- To install packages from a public feed, add NuGet Installer build step. Note, that using this build step, packages are not checked in to your version control.
- To create a package and publish it to a public feed, you need to add NuGet Pack and NuGet Publish build steps.
- To create a package and publish it to internal TeamCity NuGet Server, you need to have it enabled (see above), use NuGet Pack build step and properly configure artifact paths.
- To trigger a new build when a NuGet package is updated use NuGet Dependency Trigger.

See also:

> **Administrator's Guide**: NuGet Installer | NuGet Publish | NuGet Pack | NuGet Dependency Trigger

NuGet Installer

NuGet Installer build runner allows to pull NuGet packages required to build your project **without having to commit them into the version control**, which comes in handy especially when using a DVCS, like Mercurial or Git. Also it can (optionally) automatically update package dependencies to the most recent ones. Learn more about the problem and existing workarounds.

Prerequisites:

- Make sure that sources that you check out from VCS (VCS Settings) include the folder called `packages` from your solution folder.

To configure NuGet Installer:

1. Select NuGet version to use from the **NuGet.exe** drop-down list (if you have installed NuGet beforehand), or specify custom path to `NuGet.exe`.
2. Specify NuGet package sources, for example your own NuGet repository. In case you're using TeamCity as NuGet repository, specify `%teamcity.nuget.feed.server%` here. If you leave this field blank, NuGet will use nuget.org by default to search for your packages.
3. Specify path to your solution file (.sln) where packages should be installed.
4. If needed, select additional options:
   - **Exclude version from package folder names**: Equivalent to `-ExcludeVersion` NuGet commandline argument. If enabled, the destination folder will contain only the package name, not the version number.
   - **Update packages with help of NuGet update command**: Uses NuGet update command to update all packages under solution. Package versions and constraints are taken from `packages.config` files.

     > **Starting with TeamCity 7.1** it is no longer required to have `packages/repository.config` file under solution. TeamCity uses Visual Studio solution file (.sln) to create the full list of NuGet packages to install. It also supports Solution-Wide packages, from `.NuGet/packages.config` file. You can opt to update packages for entire solution of per `packages.config` files.

   - **Perform safe update**: Equivalent to `-Safe` NuGet option, that looks for updates with the highest version available within the same major and minor version as the installed package.

Note, that for installing and updating packages we use `NuGet.exe install` and `NuGet.exe update` commands respectively. See NuGet documentation.

If you have added NuGet Installer to your build configuration, you will probably want to know the exact versions of packages used in builds. For this purpose, there's a **NuGet Packages** tab for every finished build that lists packages used.

See also:

> **Administrator's Guide**: NuGet Pack | NuGet Publish

NuGet Pack

NuGet Pack build runner allows to build a NuGet package from a given spec file. If you want to publish this package, add one more build step called NuGet Publish.

To configure NuGet Pack:

1. Select NuGet version to use from the **NuGet.exe** drop-down list (if you have installed NuGet beforehand), or specify custom path to `NuGet.exe`.
2. Specify your package parameters:
   - Enter path(s) to `csproj` or `nuspec` file(s). You can specify as many specification files here as you need. Wildcards are supported. If you specify here a csproj file, you won't have to redefine version number and copyright information in the spec file.
   - Specify version for the package. You can use TeamCity variable `%build.number%` here.
   - Specify base directory, where the files defined in the nuspec file are located (the directory against which the paths in `<files></files>` from `nuspec` are resolved, usually some *bin* directory). If left blank, TeamCity will use build checkout directory as base directory.
3. In the "Output Directory" field, specify the path where generated NuGet package should be put. Optionally, you can clean this directory before packing. If you're using TeamCity as NuGet repository, select the **Publish created packages to build artifacts** check box to publish packages to TeamCity's NuGet server and be able to use them as regular TeamCity artifacts.

1. Set additional parameters, if needed:
   - Exclude files: specify one or more wildcard patterns to exclude when creating a package. Equavalent to NuGet.exe `-Exclude` argument.
   - Properties: Semicolon or new line separated list of package creation properties. For example to make a release build you define here `Configuration=Release`.
   - Options: Select whether the output files of the project should be in the **tool** folder. Select whether a package containing sources and symbols should be created. When specified with a nuspec, creates a regular NuGet package file and the corresponding symbols package (needed for publishing the sources to Symbolsource)
   - Set "Additional commandline arguments" to be passed to `NuGet.exe`.

See also:

Administrator's Guide: NuGet Installer | NuGet Publish

NuGet Publish

NuGet Publish build runner is intended to publish (`push`) your NuGet packages to a given feed (custom or default).

> ✓ If you're using TeamCity as NuGet server, you don't need to add this build step. However the output of the NuGet Pack build step should be a build artifact: specify path to it in General Settings of your build configuration.

To configure NuGet Publish:

1. Select NuGet version to use from the **NuGet.exe** drop-down list (if you have installed NuGet beforehand), or specify custom path to `NuGet.exe`.
2. In the **Package Sources** field specify the destination to publish the package, by default it's `nuget.org`. If you have your own NuGet server then fill in the address here.
3. Provide your API key. This field is mandatory.
4. List the packages you want to upload: each package individually or using wildcards.
5. If you want to upload your package, but keep it invisible in feed, select the "Only upload package but do not pusblish it to feed" check box. This works for NuGet versions 1.5 and older.

See also:

Administrator's Guide: NuGet Installer | NuGet Pack

NUnit

NUnit build runner is intended to run NUnit tests right on the TeamCity server. However, there are other ways to report NUnit tests results to TeamCity, please refer to the NUnit Support page for the details.

> ℹ Supported NUnit versions: **2.2.10**, **2.4.1**, **2.4.6**, **2.4.7**, **2.4.8**, **2.5.0**, **2.5.2**, **2.5.3**, **2.5.4**, **2.5.5**, **2.5.6**, **2.5.7**, **2.5.8**, **2.5.9**, **2.5.10**, **2.6.0**.

NUnit Test Settings

NUnit runner	Select NUnit version to be used to run the tests.
.NET Runtime	From the **Platform** drop-down select the desired execution mode on a x64 machine. Supported values are: Auto (MSIL) (default), x86 and x64. From the **Version** drop-down select the desired .NET Framework version.
Run tests from	Specify the .NET assemblies, where the NUnit tests to be run are stored. Multiple entries are comma-separated; usage of MSBuild wildcards is enabled. In the following example, TeamCity will search for the tests assemblies in all project directories and run these tests. `**\*.dll` ⚠ All these wildcards are specified relative to path that contains the solution file.
Do not run tests from	Specify .NET assemblies that should excluded from the list of found assemblies to test. Multiple entries are comma-separated; usage of MSBuild wildcards is enabled. In the following example, TeamCity will omit tests specified in this directory. `**\obj\**\*.dll` ⚠ All these wildcards are specified relative to path that contains the solution file.
NUnit categories include	Specify NUnit categories of tests that should be run. Multiple entries are comma-separated.
NUnit categories exclude	Specify NUnit categories that should be excluded from the tests to be run. Multiple entries are comma-separated.
Run a process per assembly	Select this option, if you want to run each assembly in a new process.
Reduce test failure feedback time	Use this option to instruct TeamCity to run some tests before others.

Code Coverage

To learn about configuring code coverage options, please refer to the Configuring .NET Code Coverage page.

See also:

Administrator's Guide: Configuring Unit Testing and Code Coverage | NUnit Support

PowerShell

PowerShell build runner is designed specifically to run PowerShell scripts.

PowerShell Settings

Option	Description

Powershell run mode	Select the desired execution mode on a x64 machine.
Working directory	Specify the path to the build working directory.
Script	Select whether you want to enter the script right in TeamCity, or specify path to a script: • **Script file**: Enter path to Powershell file. It has to be relative to checkout directory. • **Script source**: Enter Powershell script source. Note, that TeamCity references (build parameters and TeamCity system properties) will be replaced in the code.
Script execution mode	Specify powershell script execution mode. If you've selected 'Execute .ps1 script with "-File" argument' your script should be signed or you should make PowerShell allow execution of arbitrary `.ps1` files. If execution policy doesn't allow to run your scripts, select 'Put script into powershell stdin' mode to avoid this issue.
Script arguments	*Available if "Script execution mode" option is set to "Execute .ps1 script with "-File" argument".* Specify here arguments to be passed into PowerShell script.
Additional command line parameters	Specify parameters to be passed to `powershell.exe`.

Development Links

PowerShell support is implemented as an open-source plugin. For development links refer to the plugin's page.

Rake

TeamCity Rake runner supports **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. It is possible to install Ruby interpreter and necessary Ruby gems using Command Line build runner step. If you want to automatically configure agent requirements for this interpreters you need to register its paths in build agent configuration properties and then refer to such property name in 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. Also instead of *gem* command you can install gems using Bundler gem

If you use Ruby 1.9 for Shoulda, Test-Spec and Test::Unit frameworks to operate, the 'test-unit' gem must be installed.

Important Notes

- Ruby's *pending specs* are shown as **Ignored Tests** in the **Overwiew** tab.
- Rake Runner uses its own unit tests runner and loads it using `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 exist for Cucumber tests options.
- When reporting issues, you can look up version of the Rake Runner in `rakeRunnerPluginAgent.zip/version` (agent plugin), and `rakeRunnerPluginServer.jar/version` (server plugin).
- To include HTML reports into the Build Results, you can add corresponding report tab for them.

Rake Runner Settings

Rake Parameters

Option	Description
Path to a Rakefile file	Enter Rakefile path if you don't want to use a default one. Specified path should be relative to the Build Checkout Directory. ✓ Alternatively, click to choose the file using VCS repository browser (Currently VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)
Rakefile content	Type in the Rakefile content instead of using existing Rakefile. The new Rakefile will created dynamically from the specified content before running Rake.
Working directory	Optional. Specify if differs from the Build Checkout Directory.
Rake tasks	Enter tasks names separated by space character if you don't want to use '`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: `ruby rake <Additional Rake command line parameters> <TeamCity Rake Runner options, e.g TESTOPTS> <tasks>`

Ruby Interpreter

Option	Description
Use default Ruby	Use Ruby interpreter settings defined in Ruby environment configurator build feature settings or interpreter will be searched in the `PATH`.
Ruby interpreter path	Path to Ruby interpreter. Path cannot be empty. In this field you can use 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, that 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 Bundler requirements manager and your Rakefile doesn't load bundler setup script, this option will allow you to launch rake tasks using 'bundle exec' command emulation. If you want to execute 'bundle install' command you need to do it in Command Line step before *Rake runner* step. Also don't forget to setup Ruby environment configurator build feature to automatically pass Ruby interpreter to command line runner.
Debug	Check the **Track invoke/execute stages** option to enable showing *Invoke* stage data in the build log.

Tests Reporting

Option	Description
Attached reporters	If you want TeamCity to display test results on the 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 current rvm sdk isn't set or invoke "rvm system" at first.

Additional Runner Options

These options can be configured using system properties in Build Parameters section.

Option	Description
system.teamcity.rake.runner.gem.rake.version	Allows to specify which rake gem to use for launching rake build.
system.teamcity.rake.runner.gem.testunit.version	If your application doesn't use latest installed (in Ruby sdk) test-unit gem version please specify it here. Otherwise Test::Unit test reporter may try to load incorrect gem version and affect runtime behavior. If test-unit gem is installed but you application uses Test::Unit bundled in Ruby 1.8.x SDK please set version value to 'built-in'.
system.teamcity.rake.runner.gem.bundler.version	Launches bundler emulation for 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 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 and 2010 solution files.

- General Build Runner Options

General Build Runner Options

Option	Description

Solution file path	Specify the path to the solution to be built relative to the Build Checkout Directory. For example: `vs-addin\addin\addin.sln` Alternatively, click to choose the file using VCS repository browser (Currently VCS repository browser is only available for Git, Mercurial, Subversion and Perforce.)
Working directory	Specify the Build Working Directory. (optional)
Visual Studio	Select the Visual Studio version: **2005**, **2008**, **2010**.
Targets	Specify the Microsoft Visual Studio targets specific for the previously selected Visual Studio version. The possible options are **Build**, **Rebuild**, **Clean**, **Publish** or a combination of these targets based on your needs. Multiple targets are space-separated.
Configuration	Specify the name of Microsoft Visual Studio solution configuration to build (optional).
Platform	Specify the platform for the solution. You can leave this field blank, and TeamCity will obtain this information from the solution settings (optional).
Command line parameters	Specify additional command line parameters to be passed to the build runner. Instead of explicitly specifying these parameters, it is recommended to define them on the Build Parameters page.

Visual Studio 2003

Visual Studio 2003 Build Runner supports building Microsoft Visual Studio 2003 .NET projects.

- The Visual Studio 2003 build runner uses NAnt instead of MS Visual Studio 2003 to perform the build. As a result the agent is required to have .NET Framework 1.1 installed, however under certain conditions .NET Framework SDK 1.1 might be required. This NAnt solution task may behave differently than MS Visual Studio 2003. See http://nant.sourceforge.net/release/latest/help/tasks/solution.html for details.
- To use this runner you need to configure the NAnt runner.

Option	Description
Solution file path	A path to the solution to be built is relative to the build checkout directory. For example: `vs-addin\addin\addin.sln` Alternatively, click to choose the file using VCS repository browser (Currently 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: `http://localhost:8111=myProjectPath/myProject` where - http://localhost:8111 is a host where the project will be uploaded - `myProjectPath/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	Specify any additional parameters for `NAnt.exe` 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.
Run NUnit tests for	Specify .Net assemblies, where the NUnit tests to be run are stored. Multiple entries are comma-separated; usage of NAnt wildcards is enabled. In the following example, TeamCity will search for the tests assemblies in all project directories and run these tests. `**\*.dll` All these wildcards are specified relative to path that contains the solution file.
Do not run NUnit tests for	Specify .NET assemblies that should be excluded from the list of found assemblies to test. Multiple entries are comma-separated; usage of NAnt wildcards is enabled. In the following example, TeamCity will omit tests specified in this directory. `**\obj\**\*.dll` All these wildcards are specified relative to path that contains the solution file.
Reduce test failure feedback time	Use following option to instruct TeamCity to run some tests before others.
Run recently failed tests first	If checked, in the first place TeamCity will run tests failed in previous finished or running builds as well as tests having high failure rate (a so called *blinking* tests)

See also:

Administrator's Guide: NUnit for MSBuild

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	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. ⓘ Specified path should be relative to the checkout directory.
Detect global libraries and module-based JDK in the *.iml files	If this option is checked, all the module files will be automatically scanned for references to the global libraries and module JDKs when saved. This helps you ensure all references will be properly resolved. ⛔ **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	Click to reparse the project and import build settings right from the IDEA project, for example the list of JDKs. If you update your project settings in IntelliJ IDEA - add new jdks, libraries, don't forget to update build runner settings by clicking **Check/Reparse Project**.
Working directory	Enter a path to a Build Working Directory, if it differs from the Build Checkout Directory. Optional, specify if differs from the checkout directory.

Unresolved Project Modules and Path Variables

This section is displayed, when an IntelliJ IDEA module file (.iml) referenced from IPR-file:

- cannot be found
- allows you to enter the values of path variables used in the IPR-file.

To refresh values in this section click **Check/Reparse Project**.

Option	Description
<path_variable_name>	This field appears, if the project file contains path macros, defined in the Path Variables dialog of IntelliJ IDEA's Settings dialog. In the **Set value to field**, specify a path to project resources, to be used on different build agents.

Project JDKs

This section provides the list of JDKs detected in the project.

Option	Description
JDK Home	Use this field to specify JDK home for the project. When building with the **Ipr** runner, this JDK will be used to compile the sources of corresponding IDEA modules. For **Inspections** and **Duplicate Finder** builds, this JDK will be used internally to resolve the Java API used in your project. To run the build process itself the JDK specified in the `JAVA_HOME` environment variable will be used.
JDK Jar File Patterns	Click this link to open a text area, where you can define templates for the jar files of the project JDK. Use Ant rules to define the jar file patterns. The default value is used for Linux and Windows operating systems: For Mac OS X, use the following lines:
IDEA Home	If your project uses the IDEA JDK, specify the location of IDEA home directory
IDEA Jar Files Patterns	Click this link to open a text area, where you can define templates for the jar files of the IDEA JDK.

You can use references to external properties when defining the values, like `%system.idea_home%` or `%env.JDK_1_3%`. This will add a requirement for the corresponding property.

Java Parameters

Option	Description
JDK home path	Use this field to specify the path to your custom JDK which should be used to run the build. If the field is left blank, the path to JDK Home is read either from the `JAVA_HOME` environment variable on agent computer, or from `env.JAVA_HOME` property specified in the build agent configuration file (buildAgent.properties). If these both values are not specified, TeamCity uses Java home of the build agent process itself.

JVM command line parameters	Specify the desired Java Virtual Machine parameters, such as maximum heap size or parameters that enable remote debugging. These settings are passed to the JVM used to run your build. Example:

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	Click the **Type test patterns** link, and specify the required test patterns in a text area.
	These patterns are used to generate parameters of the `batchtest` JUnit task section. Each pattern generates either `include` or `exclude` 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.
	Each pattern should be placed on a separate line and has the following format:
	`[-]moduleName:[testFileNamePattern]`
	where:
	• **[-]**: If a pattern starts with minus character, the corresponding files will be excluded from the build process.
	• **moduleName** : this name can contain wildcards.
	• **[testFileNamePattern] :** Default value for **testFileNamePattern** is **/*Test.java , i.e. all files ending with Test.java 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:
	`*test:**/*Test.java`
	`-*:**/ui/**/*.java`

Search for tests	In IDEA project, a user can mark a source code folder as either "*sources*" or "*test*" root. This drop-down list allows you to specify directories to look for tests: • **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: - **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	Use following two options to instruct TeamCity to run some tests before others. 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.
Run recently failed tests first	If checked, in the first place TeamCity will run tests failed in previous finished or running builds as well as tests having high failure rate (a so called *blinking* tests)
Run new and modified tests first	If checked, before any other test, TeamCity will run tests added or modified in change lists included in the running build. If both options are enabled at the same time, tests of the **new and modified tests** group will have higher priority, and will be executed in the first place.
Verbose Ant	Check this option, if the generated JUnit task has to produce verbose output in ant terms.

Code Coverage

To learn about configuring code coverage options, please refer to the Configuring Java Code Coverage page.

Adding Build Features

A "build feature" is a piece of functionality that can be added to the build configuration and affect build running or results reporting.
Build features are configured on the Build Steps page of the build configuration settings.

✓ You can disable a build feature temporarily or permanently at any time, even if it is inherited from a build configuration template.

Currently available build features are:

- AssemblyInfo Patcher
- Build Files Cleaner (Swabra)
- Ruby Environment Configurator
- XML Report Processing
- Free disk space
- Performance Monitor

AssemblyInfo Patcher

AssemblyInfo Patcher build feature allows to set a build number to an assembly automatically, without having to patch `AssemblyInfo.cs` files manually. When adding this build feature you need only to specify version format. Note, that you can use TeamCity parameter references here.

When configured, TeamCity searches for all AssemblyInfo files (`.cs`, `.vb` or `.cpp`) in their standard locations under checkout directory and replaces parameter for `AssemblyVersion` and `AssemblyFileVersion` attributes with the build number you have specified in TeamCity web UI.

Note, that this feature will work only for "standard" projects, i.e. created by means of Visual Studio wizard, so that all the AssemblyInfo files and content have standard location.

At the end of the build the files are reverted to initial state.

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 next build start depending on the settings.
Swabra also detects modified or deleted during the build files 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 modified or deleted by previous builds files and init 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.

Checkout directory state is saved into a file in the caches directory named `<checkout_directory_name_hash>.snapshot` using DiskDir format. Checkout directory to snapshot name map is saved into snapshot.map file. The snapshot is used later (t the end of the build or at next build start) to determine which files and folders are newly created, modified or deleted. It is done by actual files' presence, last modification data and size comparison with 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 previous build. If this option is disabled you will only get warnings about modified and deleted files.

Paths to monitor	Specify newline-separated set of `+|-:path` rules to define what files and folders should be involved in files collection process (by default entire checkout directory is monitored). The path can be relative (based from build's checkout directory) or absolute and can include Ant-like wildcards. If no `+:` or `-:` prefix is specified, a rule as treated as "include". Specifying a directory affects it's 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 it's content except `inner` subfolder and it's content. ⚠ Note that after removing some exclude rules it's advisable to run clean checkout.
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 handle.exe 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 agent swabra by default ignores all `.svn`, `.git`, `.hg`, `CVS` folders and their content.
To turn off this behaviour specify 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 **URL for downloading handle.exe** or download it manually and specify path on local machine, click **Continue** and TeamCity will automatically download or upload `handle.exe` and send to Windows agents.

handle.exe is present on agents only after the upgrade process.

You may also download handle.exe, extract it on 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 `swabra.preserve.snapshot` system property to preserve snapshots for debug purposes.

Development links

See plugin page at Swabra.

Ruby Environment Configurator

Ruby environment configurator build feature passes Ruby interpreter to all build steps. The build feature adds selected Ruby interpreter and gems bin directories to system PATH environment variable and configures other necessary environment variables in case of RVM interpreter. E.g. in Command Line build runner you will be able to directly use such commands as *ruby*, *rake*, *gem*, *bundle*, etc. Thus if you want to install gems before launching Rake build runner you need to add Command Line build step which launches custom script like:

```
gem install rake --no-ri --no-rdoc
gem install bundler --no-ri --no-rdoc
```

Ruby Environment Configurator Settings

Option	Description
Ruby interpreter path	Path to Ruby interpreter. If not specified the interpreter will be searched in the `PATH`. In this field you can use 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, that interpreter name cannot be empty. If gemset isn't specified the default one will be used. ⓘ This option can be used if you don't want use `.rvmrc` settings, for instance to run tests on different ruby interpreters instead of hardcoded in `.rvmrc` file.
RVM with .rvmrc file	**Since TeamCity 7.1** Specify here the path to a `.rvmrc` file which should be relative to the checkout directory. If the file is specified, TeamCity will fetch environment variables using rvm-shell and will pass it to all build steps.
Fail build if Ruby interpreter wasn't found	Check the option to fail build if Ruby environment configurator cannot pass Ruby interpreter to step execution environment because the interpreter wasn't found on agent.

See also:

> **Administrator's Guide**: Configuring Build Steps | Command Line | Rake

XML Report Processing

XML Report processing Build Feature allows to specify report files on disk that will be parsed during the build and the results reported as the build results.

The report parsing can also be initiated from within the build via service messages.

XML Report Processing supports the following testing frameworks:

- JUnit Ant task
- Maven Surefire plugin
- NUnit-Console XML reports
- MSTest TRX reports (for MSTest 2005/2008/2010)
- Google Test XML reports

and the following code inspection tools:

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

and the following code duplicates tools:

- PMD Copy/Paste Detector XML reports

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 the reporting is started as soon as first data is available and more data is reported as it is written on disk.

The plugin takes into account only the files updated since the build start (determined by means of the last modification file timestamp).

onfiguring XML Report Processing

Add XML Report Processing as a build feature on the **Build Steps** page and configure its settings:

- Choose report type and specify monitoring rules in the form of `+|-:path` separating them by comma or new line. Paths without `+|:` prefix are treated as including. Ant-style wildcards like `dir/**.xml` (this means all files with extension .xml under directory "dir") are supported here.

  > ⚠ TeamCity loads generated reports once when they are created, make sure your build procedure generates files with unique names for each tests set without report files overwriting.

- Check **Verbose output** option to enable detailed logging to build log.
- For FindBugs report processing it is necessary to specify path to FindBugs installation on 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 the 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

Free disk space

Free disk space build feature allows to ensure the build gets enough disk free space.

Before the build, the agent will check the current available disk free space. If the amount is less then specified, it will try to clean data of other builds before proceeding.

The data cleaned includes:

- checkout directories that were marked for deletion;
- 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: agent's temp directory and build checkout directory.

By default each build has the required free space set to 3Gb.

You can specify custom free disk space value in UI - as the build feature in Build Configuration settings.

If you need to make sure a checkout directory is never deleted while freeing disk space, set `teamcity.build.checkoutDir.expireHours` property to `"never"` value. See more at Build Checkout Directory.

Other ways to set the free disk space value

Also, for compatibility reasons free disk space value can be specified via properties. It is advised to use 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.

Required free space value is defined by the following properties:
`system.teamcity.agent.ensure.free.space` for build checkout directory.
`system.teamcity.agent.ensure.free.temp.space` for agent's 'temp' directory. If teamcity.agent.ensure.free.temp.space is not defined, the value of teamcity.agent.ensure.free.space property is used.

The values of the properties specifies the size of the free disk space to ensure before the build start. The value should be a number followed by kb, mb, gb, kib, mib, or gib suffix. Use no suffix for bytes.
e.g. `system.teamcity.agent.ensure.free.space = 5Gb`

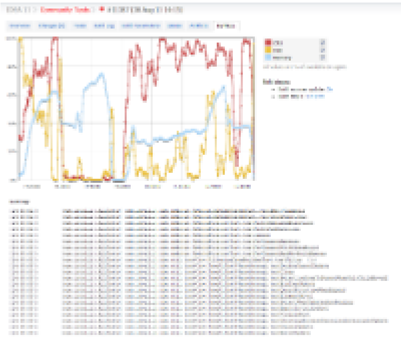
See also:

> **Administrator's Guide**: TeamCity Server Disk Space Watcher

Performance Monitor

TeamCity provides a build feature called **Performance Monitor** that allows to get the statistics on CPU, disk and memory usage during build run on build agent. When enabled, each build has an additional tab called **PerfMon** at the build results page, where this statistics is presented as a graph. You can also click on points in chart to see the corresponding part of the build log.

For example, from the picture below it is clear that at some point CPU and Disk usage is very low. This lasts for about 20 minutes. Seems like 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:



Performance monitor supports Windows, Linux, Solaris and MacOS X operating systems. Note that 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 agent and server installed on the same machine.

Xcode Project

Since TeamCity 7.1 Xcode Project runner comes bundled with TeamCity. If you want to use it with TeamCity 7.0, you can install it as a plugin.
Xcode Project runner supports both Xcode 3 (target-based build) and Xcode 4 (scheme-based build), provides structured build log based on Xcode build stages, detects compilation errors, reports tests from "xcodebuild", adds automatic agent requirements for appropriate version of tools installed (Xcode, SDKs, etc.) and reporting tools via agent properties.
To run an Xcode build you should have one or more build agents running Mac OS X with installed Xcode 3 or 4.

Xcode Project Runner Settings

Setting	Xcode	Description
Path to the project or workspace		Path to the project file (`.xcodeproj`) or workspace file (`.xcworkspace`), should be relative to the checkout directory. For Xcode 3 build only path to the project is allowed.
Working directory		Specify the build working directory.
Xcode		Build settings depend on Xcode version - for Xcode 3 you can specify target-based build and for Xcode 4 you can specify scheme-based build.
Scheme	Xcode 4	Xcode scheme to build. List of available schemes is formed by parsing your project/workspace files in VCS. Press "Check/Reparse Project" button to show/refresh schemes list (make sure your "Path to the project or workspace" setting is correct before). Please note that scheme must be shared to be shown in the list (to check if your scheme is shared you should check if it is located under "xcshareddata" folder and not under "xcuserdata" one, and if "xcshareddata" folder is committed to your VCS; to check the latter you can use VCS tree popup near the "Path to the project or workspace" field). To know more about managing Xcode schemes please see Apple documentation.
Build output directory	Xcode 4	You can override the default path for the files produced in your build. To do it check "Use custom" checkbox and specify the path in the text field. Path should be relative to the checkout directory.
Target	Xcode 3	Xcode target to execute. List of available targets is formed by parsing your project files in VCS. Press "Check/Reparse Project" button to show/refresh targets list (make sure your "Path to the project" setting is correct before).
Configuration	Xcode 3	Xcode configuration. List of available configurations is formed by parsing your project files in VCS. Since configuration depends on the target, you must choose the target first. Press "Check/Reparse Project" button to show/refresh configurations list (make sure your "Path to the project" setting is correct before).
Platform	Xcode 3	You can choose either "iOS" or "Mac OS X" or "Simulator - iOS" or any other platform (if it is provided by agent) to build your project on.
SDK	Xcode 3	You can choose SDK to build your project with (list of available SDKs is formed according to the SDKs those are available on your agents). Since SDK depends on the platform, you must choose the platform first.
Architecture	Xcode 3	You can choose architecture to build your project with (list of available architectures is formed according to the architectures those are available on your agents). Since architecture depends on the platform, you must choose the platform first.
Build action(s)		Xcode build action(s). Default actions are "clean build", but you can specify any of "clean", "build", "test", "archive", "installsrc", "install" in any order.
Run tests		Uncheck this option if you just want to build your project and do not want to run tests.

Additional command line parameters		Any other command line parameters to be passed to "xcodebuild" utility.

See also:

> **Concepts**: Build Runner

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**.

Please, refer to NUnit build runner page.

Alternative approaches

However, if for some reason it is not applicable, TeamCity provides following ways to configure NUnit tests reporting in TeamCity:

- TeamCity supports standard `<nunit2>` NAnt task.
- TeamCity provides `<NUnitTeamCity>` MSBuild task and supports `<NUnit>` MSBuild task from the MSBuild Community tasks.
- TeamCity provides its own NUnit Test Launcher, that can be configured in the MSBuild build script, or launched from command line.
- TeamCity Addin for NUnit is available to turn on reporting on NUnit level without build procedure modifications. TeamCity NUnit Addin supports 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**.
- The bundled XML Test Reporting plugin allows to import any xml report to TeamCity. In this case it is not always possible to track results on the fly. You can add **XML Report Processing** build feature to your build configuration, or use 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 to configure tests reporting manually via service messages.

Comparison matrix:

Approach	Real-Time Reporting	Execution without TeamCity	Tests Reordering	Implicit TeamCity .NET Coverage
NUnit runner	✓	✗	✓	✓
`<nunit2>` NAnt task	✓	✓/✗*	✓	✓
`<NUnit>` MSBuild task	✓	✓/✗*	✓	✓
`<NUnitTeamCity>` MSBuild task	✓	✓/✗*	✓	✓
TeamCity addin for NUnit	✓	✗	✗	✗
TeamCity NUnit Test Launcher	✓	✗	✓	✓
XML Reporting Plugin	✗	only xml	N/A	N/A

***** TeamCity-provided tasks may have different syntax/behavior. Some workarounds may be required to run the script without TeamCity.

In addition to the common test reporting features, TeamCity relieves a headache of running your NUnit tests under x86 process on the x64 machine by introducing an explicit specification of the platform and run-time environment versions. You can define whether to use .NET Framework 1.1, 2.0 or 4.0 started under MSIL, x64 or x86 platform.

This section covers:

- TeamCity NUnit Test Launcher
- NUnit for NAnt Build Runner
- NUnit for MSBuild
- MSBuild Service Tasks
- NUnit Addins Support
- TeamCity Addin for NUnit

See also:

Administrator's Guide: NUnit build runner | MSTest Support | Running Risk Group Tests First | XML Report Processing

TeamCity NUnit Test Launcher

TeamCity provides its own NUnit tests launcher that can be used from command line. The tests are run according to the passed parameters and if the process is run inside TeamCity build agent environment, the results are reported to the TeamCity agent.

- If you need to access the path to TeamCity NUnit launcher from some process, you can add an environment variable `%system.teamcity.dotnet.nunitlauncher%`.
- Values surrounded with "%" within custom scripts in Commandline runner TeamCity treats as TeamCity references.

You can pass to the TeamCity NUnit Test Launcher the following command line options:

```
${teamcity.dotnet.nunitlauncher} <.NET Framework> <platform> <NUnit vers.> [/category-include:<list>]
[/category-exclude:<list>] [/addin:<list>] <assemblies to test>
```

Option	Description
<.NET Framework>	Version of .NET Framework to run tests. Acceptable values are **v1.1**, **v2.0**, **v4.0** or **ANY**.
<platform>	Platform to run tests. Acceptable values are **x86**, **x64** and **MSIL**. For .NET Framework 1.1 only **MSIL** option is available.
<NUnit vers.>	Test framework to use. The value has to be specified in the following format: **NUnit-<version>**. 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**.
/category-include:<list>	The list of categories separated by **';'** (optional).
/category-exclude:<list>	The list of categories separated by **';'** (optional).
/addin:<list>	List of third-party NUnit addins to use (optional).
<assemblies to test>	List of assemblies paths separated by **';'** or space.
/runAssemblies:processPerAssembly	Specify, if you want to run each assembly in a new process.

Examples

The following examples assume that the `teamcity.dotnet.nunitlauncher` property is set as system property on the Build Parameters page of the Build Configuration.

Run tests from an assembly:

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.2.10 Assembly.dll
```

Run tests from an assembly with NUnit categories filter

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.2.10 /category-include:C1 /category-exclude:C2
Assembly.dll
```

Run tests from assemblies:

```
%teamcity.dotnet.nunitlauncher% v2.0 x64 NUnit-2.5.0 /addin:Addin1.dll;Addin2.dll Assembly.dll
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 nunt2 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
teamcity.dotnet.nant.nunit2.failonfailureatend	Run **all tests** regardless of the number of failed ones, and fails if at least one test has failed.
teamcity.dotnet.nant.nunit2.platform	Sets desired runtime execution mode for .NET 2.0 on x64 machines. Supported values are **x86**, **x64** and **ANY**(default).
teamcity.dotnet.nant.nunit2.platformVersion	Sets desired .NET Framework version. Supported values are **v1.1**, **v2.0**, **v4.0**. Default value is equal to NAnt target framework
teamcity.dotnet.nant.nunit2.version	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**.
teamcity.dotnet.nant.nunit2.addins	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 form 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>
```

> ⚠
>
> - Be sure to replace "." with "_" when using Build Parameters in MSBuild scripts. That is you have to 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.

- 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 `NUnit` task from MSBuild Community tasks project.
`ExcludeCategory`	As used in the `NUnit` 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**. For example, `NUnit-2.2.10`.
`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 Build Parameters in MSBuild scripts: use `teamcity_dotnet_nunitlauncher_msbuild_task` instead of `teamcity.dotnet.nunitlauncher.msbuild.task`
- TeamCity also provides Solution Runner for Microsoft Visual Studio 2005 and 2008 solution files. It allows you to use MSBuild-style wildcards for the assemblies to run unit tests on.

Examples

Run NUnit tests using specific NUnit runner version.

```
<Target Name="build_01">
     <!-- start tests for NUnit-2.2.10 -->
     <NUnitTeamCity Assemblies="@(TestAssembly)" NUnitVersion="NUnit-2.2.10"/>

     <!-- start tests for NUnit-2.4.6 -->
     <NUnitTeamCity Assemblies="@(TestAssembly)" NUnitVersion="NUnit-2.4.8"/>
  </Target>
```

Run NUnit tests with custom addins with NUnit 2.4.6:

```
<Target Name="build">
     <NUnitTeamCity Assemblies="@(TestAssembly)" Addins="NUnitExtension.RowTest.AddIn.dll"
NUnitVersion="NUnit-2.4.6"/>
  </Target>
```

Run NUnit tests with custom addins with NUnit 2.4.6 **in per-assembly mode**.

```
<Target Name="build">
     <NUnitTeamCity Assemblies="@(TestAssembly)" Addins="NUnitExtension.RowTest.AddIn.dll"
NUnitVersion="NUnit-2.4.6" RunProcessPerAssembly="True"/>
</Target>
```

To make TeamCity independent build script, consider the following trick:

```
<NUnitTeamCity ... Condition=" '$(TEAMCITY_VERSION)' != '' "/>
```

MSBuild Property TEAMCITY_VERSION is added to msbuild when started from TeamCity.

MSBuild Service Tasks

For MSBuild, TeamCity provides the following service tasks that implement the same options as the service messages:

- TeamCitySetBuildNumber
- TeamCityProgressMessage
- TeamCityPublishArtifacts
- TeamCityReportStatsValue
- 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" />
```

TeamCitySetStatus

`TeamCitySetStatus` is a task to change current status text and/or message

```
<TeamCitySetStatus Status="ERROR" Text="ZZZ" />
```

*`'{build.status.text}'` is substituted with older status text.
Status could have one of the following values: `NOT_CHANGED`, `FAILURE`, `SUCCESS`, `NORMAL`, `ERROR`

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 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 TeamCity server. Alternatively, you can opt to use XML Test Reporting plugin, or manually configure reporting tests by means of service messages.

> TeamCity NUnit Addin supports 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**.

In order to be able reviewing test results in TeamCity, do the following:

1. In your build, set a path to TeamCity Addin to the system property **teamcity.dotnet.nunitaddin** (for MSBuild it would be **teamcity_dotnet_nunitaddin**), and add the version of NUnit to 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 **.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 NUnit console runner with TeamCity Addin for NUnit 2.4.7 (on MSBuild):

```
<ItemGroup>
  <NUnitAddinFiles Include="$(teamcity_dotnet_nunitaddin)-2.4.7.*" />
</ItemGroup>

<Target Name="RunTests">
  <MakeDir Directories="$(NUnitAddinsDir)/addins" />
  <Copy SourceFiles="@(NUnitAddinFiles)" DestinationFolder="$(NUnitAddinsDir)/addins" />
  <Exec Command="$(NUnit) $(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 assembly according to created `AssemblyName` object, however, `'addins'` folder of NUnit 2.4.8 is not included in application probe paths. Thus NUnit 2.4.8 fails to load any addin in console mode.
To solve the problem we suggest you to use any of the following workarounds:

- copy TeamCity addin assembly both to NUnit `bin` and `bin/addins` folders
- patch `NUnit-Console.exe.config` to include addins to application probe paths. Add the following code into config/runtime element:

  ```
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
        <probing privatePath="addins"/>
  </assemblyBinding>
  ```

  See original blog post on this issue http://nunit.com/blogs/?p=56

Environment variables
If you need to configure environment variables for NUnit explicitly, specify environment variable with value reference **%system.teamcity.dotnet.nunitaddin%**.
See Configuring Build Parameters for details.

MSTest Support

TeamCity provides support for MSTest 2005/2008/2010 testing framework via parsing of the MSTest results file (.trx file).
⚠ Due to specifics of MSTest tool, TeamCity does **not** support on-the-fly test reporting for MSTest. All test results are imported **after** tests run has finished.

There are two ways to report test results to TeamCity:

- Add MSTest runner as one of your build steps.
- Configure XML Report Processing via build feature or via service message to parse the .trx reports that are produced by your build procedure.

The easiest way to set up MSTest tests reporting in TeamCity is to add MSTest build runner as one of the steps to your build configuration and specify there all the required parameters.
Please, refer to MSTest build runner page for details.

If the tests are already run within your build script and MSTest generates .trx reports, you can configure service messages to parse the reports.

Autodetection of MSTest

Build agent will add `%system.MSTest.8.0%`, `%system.MSTest.9.0%` or `%system.MSTest.10.0%` system property (see Build Agent Configuration for detail) in cases `MSTest.exe` from Microsoft Visual Studio 2005/2008/2010 is found on the build agent running system.

See also:

Concepts: Testing Frameworks
Administrator's Guide: NUnit Support

Configuring .NET Code Coverage

NUnit Test Launcher bundles support for .NET code coverage using NCover, PartCover and dotCover coverage engines.
Details on configuring code coverage can be found at corresponding pages:

Coverage support limitations

Coverage configuration via TeamCity UI is only supported for the cases when the the tests are run using TeamCity-managed test launchers.
Specifically this covers:
NAnt runner: <nunit2> NAnt task
MSBuild runner: <NUnit> and <NUnitTeamCity> MSBuild tasks
Any runner: TeamCity NUnit Test Launcher

See also:

Administrator's Guide: NUnit Support

NCover

TeamCity supports code coverage with NCover (1.x and 3.x) for NUnit tests run via TeamCity NUnit test runner, which can be configured in one of the following ways: web UI, command line, `NUnitTeamCity` task, `NUnit` task, `nunit2` task.

ⓘ **Important Notes**

- In order to launch coverage, NCover and NCoverExplorer should be installed on an agent where 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 - find the description of the available options below.

Option	Description
Path to NCover	Specify the path to NCover installed on a build agent, or use `%system.ncover.v1.path%` to refer to auto-detected NCover on a 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. ⚠ • Do not enter here arguments, that can be configured in the web UI. • Do not specify here output path for the generated reports. It is configured automatically by TeamCity.
Assemblies to Profile	Specify assembly names (without paths and extensions) separated with new line, 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 coverage statistics. Equivalent to `//ea` NCover.Console option
Report Type	Select the report type. For the details, refer to NCoverExplorer documentation.
Sorting	Select the preferred sorting option. For the details, refer to NCoverExplorer documentation.
Additional NCoverExplorer Arguments	Specify additional arguments to be passed to NCoverExplorer. Do not enter here the output path for the reports, nor specify arguments, for which there are corresponding options in the UI.

Configuring NCover 3.x

Make sure, you use have specified the same platform both for NCover and NUnit.

To configure NCover 3.x:

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)** as a .NET coverage tool.
4. Set up the NCover options - find the description of the available options below.

Option	Description
Path to NCover 3	Specify the path to NCover installation folder. Alternatively use `%system.ncover.v3.x86.path%` or `%system.ncover.v3.x64.path%` to refer to auto-detected NCover 3 on a build agent.
Run NCover under	Select the preferred platform to run coverage under - x86 or x64. Make sure the selected platform agrees with the one used for NUnit tests.
NCover Arguments	Specify NCover arguments, i.e. assemblies to profile and coverage tool specific arguments. Do not enter here arguments, which can be specified in the UI, nor enter here output path for generated reports and NCover process parameters. Use **//ias .*** to get coverage of all assemblies.
NCover Reporting Arguments	Specify additional NCover reporting arguments, except for the output path. Use **//or FullCoverageReport:Html:{teamcity.report.path}** to get a report.

Reporting NCover Results Manually

if .NET code coverage is collected by the build script and need to be reported inside TeamCity (for example, Rake runner, or if you run tests via test launcher other than TeamCity NUnit Test Launcher), there is a way to let TeamCity know about the coverage data. Please read more at Manually Configuring Reporting Coverage.

JetBrains dotCover

TeamCity comes bundled with console runner of JetBrains dotCover. Just by enabling configuration option you can collect code coverage for your .Net project and then view coverage statistics and detailed coverage report inside TeamCity web UI.

If you have a purchased dotCover installed on a developer machine, TeamCity-collected coverage results can be downloaded and viewed inside Visual Studio with the help of TeamCity Visual Studio Add-in.

dotCover Settings

Path to dotCover Home	Leave this field blank to use bundled dotCover. Alternatively, specify path to dotCover installed on a build agent.
Filters	Specify assemblies to profile one per line using following syntax: `+:assembly=*;type=*;method=***`. Use `-:assembly` to exclude an assembly from code coverage. Asterisk wildcard (`*`) is supported here.
Attribute Filters	**Starting with TeamCity 7.1**, if you don't want to know coverage data solution-wide, you can exclude from coverage statistics code marked with an attribute (for example, code marked with `ObsoleteAttribute`). You only need to specify these attribute filters here in a following format: filters should be a new-line separated list; use the `-:attributeName` or `-:module=myassembly;attributeName` syntax to exclude a code marked with attributes from code coverage. Use asterisk * as a wildcard if needed. Supported only for dotCover 2.0 or newer.

✓ Note, that dotCover coverage engine reports statement coverage instead of line coverage.

See also:

> **Administrator's Guide**: Manually Configuring Reporting Coverage

PartCover

TeamCity supports code coverage with PartCover (2.2 and 2.3) for NUnit tests run via 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**
>
> - In order to launch coverage, PartCover should be installed on an agent where 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 PartCover arguments to the TeamCity NUnit test runner.
> - PartCover supports .NET Framework 2.0 and 3.5 started under x86 platform. Make sure you have configured your NUnit tests to run on the .NET Framework version 2.0 or 3.5 and under x86.

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. 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 corresponding system property, if configured.
Additional PartCover Arguments	Specify additional PartCover arguments, except the ones that can be specified using web UI. Moreover, do not specify here the output path for the generated reports, because TeamCity configures it automatically.
Include Assemblies	Explisitly specify the assemblies to profile, or use `[*]*` to include all assemblies.
Exclude Assemblies	Explicitly specify the assemblies to be excluded from coverage statistics. If you have specified `[*]*` to profile all assemblies, type `[JetBrains*]*` here to exclude TeamCity NUnit test runner sources.
Report XSLT	Write xslt transformation rules in the following format (one per line): `file.xslt=>generatedFileName.html`. As `file.xslt` you can use the default PartCover xslt, or your own. Xslt files path is relative to build's checkout directory. Note, that default xslt files bundled with PartCover 2.3 are broken and you need to write your own xslt files to be able to generate reports.

Reporting PartCover Results Manually

if .NET code coverage is collected by the build script and need to be reported inside TeamCity (for example, Rake runner, or if you run tests via 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 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 TeamCity web UI for the specific runner.

For other cases, when the .Net code coverage is collected by the build script and need to be reported inside TeamCity (for example, Rake runner,

or if you run NUnit tests via test launcher other than TeamCity NUnit Test Launcher), there is a way to let TeamCity know about the coverage data.
Communication is done via service messages.
First, the build script need to let TeamCity know details on the coverage engine with "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 HTML report for the coverage.

Configuring Code Coverage Engine

Use the following service message template:

```
##teamcity[dotNetCoverage <key>='<value>' <key1>='<value1>' ...]
```

where **key** is one of the following:
For DotCover (optional):

key	description
`dotcover_home`	Full path to DotCover home folder to override bundled dotCover.

For NCover 3.x:

key	description	sample value
`ncover3_home`	Full path to NCover installation folder.	Path to NCover3 installation directory
`ncover3_reporter_args`	Arguments for NCover report generator.	**//or FullCoverageReport:Html:{teamcity.report.path}**

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 **snapshot** file that is generated by `dotCover.exe cover` command

Java Testing Frameworks Support

TeamCity supports JUnit and TestNG by means of following build runners:

- Ant (when tests are run by junit and testng tasks directly within the script)
- Maven2 (when tests are run by Surefire Maven plugin; on-the-fly reporting is not available.)
- IntelliJ IDEA project: IntelliJ IDEA's JUnit and TestNG run configurations are supported. Note, that such run configurations should be shared and checked in to the version control.

Configuring Java Code Coverage

TeamCity supports Java code coverage based on the IntelliJ IDEA coverage engine and EMMA open-source toolkit.
In this section:

- IntelliJ IDEA
- EMMA

See also:

> **Concepts**: Code Coverage
> **Administrator's Guide**: IntelliJ IDEA | EMMA

IntelliJ IDEA

IntelliJ IDEA coverage engine is the same engine that is used within the IntelliJ IDEA to measure code coverage. This coverage attaches to JVM as a java agent and instruments classes on the fly when they are loaded by JVM. In particular that means that classes are not changed on disk and can be safely used for distribution packages.

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 by means of IntelliJ IDEA engine, follow these steps:

1. While creating/editing Build Configuration, go to the Build Runner page.
2. Select Ant, IntelliJ IDEA Project, Gradle or Maven build runner.
3. Select **IntelliJ IDEA** as a coverage tool in the **Choose coverage** runner drop-down.
4. Set up the coverage options - find the description of the available options below.

Option	Description
Classes to instrument	Specify Java packages for wich code coverage should be gathered. Use patterns (one per line) that start with a valid package name and contain `*`. For example: `org.apache.*`.
Classes to exclude from instrumentation	Use regular expressions (one per line) to specify fully qualified class names which must be excluded from the coverage. Exclude patterns have higher priority than 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), build agent invokes EMMA to instrument compiled classes and to store the location of the source files. As a result, `coverage.em` file is created in the build checkout directory, which contains classes metadata. The collected source paths of java files are used to generate the final HTML report.

   > ⚠ All `coverage.*` files are removed at the beginning of the build, so you have to ensure that full recompilation of sources is performed in the build to have actual `coverage.em` file.

2. Test run. On this stage, actual runtime coverage information is collected. This process results in creation of the file `coverage.ec`. If there are several test tasks, data is appended to `coverage.ec`.
3. Report generation. When build ends, TeamCity generates HTML coverage report, creates a zip file with the report (`coverage.zip`) and

uploads it to the server. It also generates and uploads summary report - `coverage.txt` file, and original `coverage.e(c|m)` files to allow viewing coverage from 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 Runner page.
2. Select Ant, or Ipr build runner.
3. Select **EMMA** as a coverage tool in the **Choose coverage** runner drop-down.
4. Set up the coverage options - find the description of the available options below.

Option	Description
Include Source Files in the Coverage Data	Check this option to include source files into the code coverage report (you'll be able to see sources on the Web). **Warning** Enabling this option can increase the report size and the slow down creation of your builds. To avoid this situation, you can specify some EMMA properties (see http://emma.sourceforge.net/reference_single/reference.html#prop-ref.tables for details).
Coverage Instrumentation Parameters	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.

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 kind of "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 classpath of the java task,
or to ensure that classes used in your java task are not instrumented. Another point, you should run your java task with attribute `fork=true`.

Corresponding `emma.jar` file can be taken from `buildAgent/plugins/coveragePlugin/lib/emma.jar`.
For a typical build, 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 syntax like: `-ix -com.foo.task.*,+com.foo.*,-*Test*`, where package `com.foo.task.*` contains files for your custom task.

EMMA coverage results are unstable

Please make sure that your junit task has attribute `fork=true`. 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

Running Risk Group Tests First

This section covers:

- Reordering Risk Tests for JUnit and TestNG
- 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.

Tests reordering works the following way:

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.

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.

> ⚠ 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
> - if <package/> element is used in the TestNG XML suite

Reordering Risk Tests for NUnit

Supported build runners:

- NAnt, MSBuild, NUnit, Visual Studio 2003 build runners
- NUnit testing framework

Tests reordering only supports reordering of recently failed tests.
⚠ **Test reordering is not supported for parametrized NUnit tests**.

If risk tests reordering option is enabled, the feature for NUnit test runner works in the following way:

1. NUnit runs tests from the "risk" group using test name filter.
2. NUnit runs all other tests using inverse filter.

> ⚠
>
> - Since tests run twice, thus risk test fixtures *Set Up* and *Tear Down* will be performed twice.
> - Tests reordering feature applies to an NUnit task. That is, for NAnt and MSBuild runners, tests reordering feature will be initiated as many times as many NUnit tasks you have in your build script.

See also:

Concepts: Build Runner
Extending TeamCity: Risk Tests Reordering in Custom Test Runner

Build Failure Conditions

In TeamCity you can adjust the conditions when a build should be marked as failed.

Common build failure conditions

To do so, in the **Fail build if** area specify how TeamCity should fail builds:

- **build process exit code is not zero**: Check this option to mark the build as failed if the build process doesn't exit successfully.
- **at least one test failed**: Check this option to mark the build as failed if the build fails at least one test. If this option is not checked, the build can be marked successful even if it fails to pass a number of tests. Regardless of this option, TeamCity will run all build steps.
- **an error message is logged by build runner**: Check this option to mark the build as failed if the build runner reports an error while building.
- **it runs longer than ... minutes**: Check this option and enter a value in minutes to enable time control on the build. If the specified amount of time is exceeded, the build is automatically canceled. This option helps to deal with builds that hang and maintains agent efficiency.
- **an out of memory or crash is detected (Java only)**: Check this option to mark the build as failed if a crash of the JVM is detected, or Java out of memory problems. If possible, TeamCity will upload crash logs and memory dumps as artifacts for such builds.

Advanced build failure conditions

Starting from TeamCity 7.0 you can instruct TeamCity to mark a build as failed if it has become "worse" by some of the metrics generated by the build, for example, code coverage, or artifacts size, etc. For instance, you can mark build as failed if code coverage or code duplicates number is worse than in the previous build. Another advanced build failure condition allows to mark build as fail when a certain line is met in build log.
To add such build failure condition to your build configuration click the *Add build failure condition* link and select it from the list:

- Fail build on metric change
- Fail build on specific text in build log

> You can disable a build failure condition temporarily or permanently at any time, even if it is inherited from a build configuration template.

Fail build on metric change

When using code examining tools in your build, like code coverage, duplicates finders, inspections and so on, your build generates various numeric metrics. For these metrics you can specify a threshold which will fail the build upon exceeding them. For example, you can instruct TeamCity to mark build as failed, if code coverage or code duplicates number is worse than in the previous build.

> In general there are two ways to configure this build fail condition:
>
> - A *build metric* exceeds or is less than the specified threshold. For example: **Fail build if** `build duration (secs)` **is more than** `3000`.
> - A *build metric* has become "worse" comparing to specific build by specified value. For example: **Fail build if** `build duration (secs)` **is more than** `300` **compared to** `Last successful build`. In this case a build will fail if it runs 300 seconds longer than the last successful build.

To compare a *build metric* to, you can select last successful build, last pinned build, last finished build, build with specified build number, last finished build with specified tag.

By default, TeamCity provides the wide range of *build metrics*: artifacts size(bytes), build duration (secs), number of classes, number of code duplicates, number of covered classes, number of covered lines, number of covered methods, number of failed tests, number of ignored tests, number of inspection errors, number of inspection warnings, number of lines of code, number of methods, number of tests, percent of block-level coverage, percent of class-level coverage, percent of line-level coverage, percent of method-level coverage, test duration (secs).

Moreover you can add your own build metric. To do so, you need to modify TeamCity's configuration file `<TeamCity Data Directory>/config/main-config.xml` and add the following section there:

```
<build-metrics>
<statisticValue key="myMetric" description="build metric for number of files"/>
</build-metrics>
```

So, if your build will publish value `myMetric`, you can use it as a criteria for a build failure.

Fail build on specific text in build log

TeamCity can inspect all lines in build log for some particular text occurrence that indicates build failure.
Lines are matched fair without time and block name prefixes which precede each message in build log representation.

To configure this build failure condition you need to specify:

- a string which presence/absence in build log is an indicator of build failure,
- a failure message to be displayed on build results page when build fails due to this condition.

Note that as well as exact text you can specify a Java Regular Expression regexp pattern to be looked for in build log.

Configuring Build Triggers

Build trigger is an event that initiates a new build. The build is put into the build queue and is run when there are idle agents that can run it.

For each build configuration you can configure the following triggers:

- 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. If corresponding checkbox is enabled, a build is triggered only after successful build of the selected configuration.

  > ✓ Note that if build configuration with "Finish build trigger" has snapshot dependencies to selected build configuration, the trigger will run build on the same revisions and will attach build to the chain. Thus you can have automated promotion of builds in chain.

- Maven Artifact Dependency trigger: the build is triggered if a specified Maven artifact has changed.
- Maven Snapshot Dependency trigger: the build is triggered, when any of the snapshot dependencies is updated in the remote repository.
- **Retry build trigger**: the build is triggered if the previous build failed after specified time delay.
- Branch Remote Run Trigger: personal build is triggered automatically each time TeamCity detects new changes in particular branches of the VCS roots of the build configuration. Supports Git and Mercurial.
- NuGet Dependency Trigger: starts a build if there is a NuGet packages update detected in NuGet repository.

While creating/editing a build configuration you can configure these triggers at the **Build Triggering** page by clicking **Add new trigger**. For configuration details on each trigger, refer to corresponding sections.

> ℹ 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 custom build.

Configuring VCS Triggers

- Trigger on changes in snapshot dependencies
- Per-check-in Triggering
- Quiet Period Settings
- VCS Trigger Rules
  - Trigger Rules Example

VCS triggers automatically start a new build each time TeamCity detects new changes in the configured VCS roots.

TeamCity periodically (according to root's Changes Checking Interval of a VCS root) polls VCS roots of the build configuration for changes. Newly detected changes appear as Pending Changes of a build configuration.

A new VCS trigger with default settings triggers a build each time new changes are detected.

However you can adjust a VCS trigger to your needs by means of Quiet Period Settings and Build Trigger Rules.

Trigger on changes in snapshot dependencies

By default, If you have a build chain (i.e. a number of build configurations interconnected by snapshot dependencies), automatic triggering of builds is performed in one direction only: if the first build in chain is triggered, all the builds it depends on are triggered too, but **not vice versa**. (Learn more about snapshot dependencies) Use this option to change this behaviour.
For example, build A snapshot-depends on B: A--->B. When build A is triggered, build B is triggered too. But if build B is triggered, nothing happens to build A. If you want build A to be triggered on changes in B, enable this option. See also an example at Build Dependencies page.

Per-check-in Triggering

If you have fast builds and enough build agents, you can make TeamCity launch a new build for each check-in ensuring that no other changes get into the same build. To do that, select the **Trigger a build on each check-in** option. Moreover, if you select the **Include several check-ins in build if they are from the same committer** option, and TeamCity will detect a number of pending changes, it will group them by user and start builds having single user changes only.

This helps in sorting out whose change has broken a build, or caused new test failure.

Quiet Period Settings

By specifying the quiet period you can ensure the build is not triggered in the middle of non-atomic check-ins consisting of several VCS check-ins.

Quiet period is a period (in seconds) that TeamCity maintains between the moment the last VCS change is detected and a build is added into the queue. If new VCS change is detected in the Build Configuration within the period, the period starts over from the new change detection time. The build is added into the queue only if there were no new VCS changes detected within the quiet period.

Note that actual quiet period will not be less than maximum checking for changes interval among build configuration VCS roots. Because TeamCity must ensure that changes were collected at least once during the quiet period.

Quiet period can be set to the default value (which can be changed globally at the **Administration** | **Global Settings** page), or custom value can be specified.

> ⚠ Note, that when a build is triggered by a trigger with VCS quiet period set, the build is put into the queue with fixed VCS revisions. This ensures the build will be started with only the specific changes included. Under certain circumstances this build can later become a History Build.

VCS Trigger Rules

If no trigger rules specified, a build is triggered upon any detected change displayed for the build configuration. You can affect the changes detected by changing VCS root settings and specifying Checkout Rules.

To limit the changes that trigger the build, use VCS trigger rules. You can add these rules manually in the text area (one per line), or use the **Add new rule** option to generate them.

Each rule is ether an "include" (starts with "+") or an "exclude" (starts with "-").

The general syntax for a single rule is:

```
+|-:[user=VCS_username;][root=VCS_root_name;][comment=VCS_comment_regexp]:Ant_like_wildcard
```

Where:

- **Ant_like_wildcard** - A wildcard to match the changed file path. Only "*" and "**" patterns are supported, "?" pattern is **not** supported. The file paths in the rule can be relative (resulting paths on the agent will be matched) or absolute (started with '/', VCS paths relative to VCS root are matched).
- **VCS_username** - if specified, limits the rule only to the changes made by a user with corresponding VCS username.
- **VCS_root_name** - if specified, limits the rule only to the changes from the corresponding VCS root.
- **VCS_comment_regexp** - if specified, limits the rule only to the changes that contain specified text in VCS comment. Use Java Regular Expression pattern for matching text in a comment (see examples below).

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.

> 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.

Trigger Rules Example

```
+:.
-:**.html
-:user=techwriter;root=Internal SVN:/misc/doc/*.xml
-:lib/**
-:comment=minor:**
-:comment=^minor$:**
```

Here,

- "`-:**.html`" excludes all `.html` files from triggering a build.
- "`-:user=techwriter;root=Internal SVN:/misc/doc/*.xml`" excludes builds being triggered by `.xml` files checked in by user "techwriter" to the `misc/doc` directory of the VCS root named Internal SVN (as defined in the VCS Settings). Note that the path is absolute (starts with "/"), thus the file path is matched from the VCS root.
- "`-:lib/**`" prevents the build from triggering by updates to the "lib" directory of the build sources (as it appears on the agent). Note that the path is relative, so all files placed into the directory (by processing VCS root checkout rules) will not cause the build to be triggered.
- "`-:comment=minor:**`" prevents the build from triggering, if the changes check in comment contains word "minor".
- "`-:comment=^oops$:**`" no triggering if the comment consists of the only word "oops" (according to Java Regular Expression principles ^ and $ in pattern stand for string beginning and ending)

Branch 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/***

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 2 VCS roots which support branch remote-run and you push changes to both of them, TeamCity will start 2 personal builds with changes from each root.

See also:

> **Administrator's Guide**: Git | Mercurial

Configuring Schedule Triggers

Scheduled trigger allows to set time when a build of the configuration will be run daily or weekly. Besides you can specify more complex 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 corresponding option.

To limit the changes that trigger the build, use VCS trigger rules. You can add these rules manually in the text area (one per line), or use the **Add new rule** option to generate them.

Each rule is ether an "include" (starts with "+") or an "exclude" (starts with "-").

The general syntax for a single rule is:

Where:

- **Ant_like_wildcard** - A wildcard to match the changed file path. Only "*" and "**" patterns are supported, "?" pattern is **not** supported. The file paths in the rule can be relative (resulting paths on the agent will be matched) or absolute (started with '/', VCS paths relative to VCS root are matched).
- **VCS_username** - if specified, limits the rule only to the changes made by a user with corresponding VCS username.
- **VCS_root_name** - if specified, limits the rule only to the changes from the corresponding VCS root.
- **VCS_comment_regexp** - if specified, limits the rule only to the changes that contain specified text in VCS comment. Use Java Regular Expression pattern for matching text in a comment (see examples below).

For each file in a change the most specific rule is found (the rule matching the longest file path). The build is triggered if there is at least one file with a matching "include" rule or a file with no matching rules.

> When entering rules please note that as soon as you enter any "+" rule, TeamCity will remove the default "include all" setting.
> To include all the files, use "+:." rule.

Trigger Rules Example

Here,

- "`-:**.html`" excludes all `.html` files from triggering a build.
- "`-:user=techwriter;root=Internal SVN:/misc/doc/*.xml`" excludes builds being triggered by `.xml` files checked in by user "techwriter" to the `misc/doc` directory of the VCS root named Internal SVN (as defined in the VCS Settings). Note that the path is absolute (starts with "/"), thus the file path is matched from the VCS root.
- "`-:lib/**`" prevents the build from triggering by updates to the "lib" directory of the build sources (as it appears on the agent). Note that the path is relative, so all files placed into the directory (by processing VCS root checkout rules) will not cause the build to be triggered.
- "`-:comment=minor:**`" prevents the build from triggering, if the changes check in comment contains word "minor".
- "`-:comment=^oops$:**`" no triggering if the comment consists of the only word "oops" (according to Java Regular Expression principles ^ and $ in pattern stand for string beginning and ending)

Configuring Maven Triggers

From **Build Triggering** page you can add the following Maven dependency triggers:

- Maven Snapshot Dependency Trigger
- Maven Artifact Dependency Trigger

Maven Snapshot Dependency Trigger

Maven snapshot dependency trigger adds a new build to the queue when any of the snapshot dependencies is updated in the remote repository. 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 first updated artifact.
> To avoid inconsistency, select corresponding 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 in the queue, only if build B is not a part of build chain containing A.

Maven Artifact Dependency Trigger

Maven artifact dependency trigger adds build to the queue when specified Maven artifact changes.
To add such trigger, specify the following parameters in the **Add New Trigger** dialog:

Parameter	Description
Group Id	Specify identifier of a group the desired Maven artifact belongs to.
Artifact Id	Specify the artifact's identifier.
Version or Version range	Specify version or version range of the artifact. The version range syntax is described in the table below.

Type	Define explicitly the type of the specified artifact. By default, the type is `jar`.
Classifier	(Optional) Specify classifier of an artifact.
Maven repository URL	Specify URL to the Maven repository. Note, that this parameter is optional. If the URL is not specified, then: • For a Maven project the repository URL is determined from the POM and the server-side Maven Settings • For a non-Maven project the repository URL is determined from the server-side Maven Settings only
Do not trigger a build if currently running builds can produce this artifact	Select this option to trigger a build only after the build that produces artifacts used here is finished.

For specifying version ranges use the following syntax, as proposed in the Maven documentation:

Range	Meaning
`(,1.0]`	x <= 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 <= x <= 1.3
`[1.0,2.0)`	1.0 <= x < 2.0
`[1.5,)`	x >= 1.5
`(,1.0],[1.2,)`	x <= 1.0 or x >= 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

NuGet Dependency Trigger

NuGet Dependency Trigger allows to start a new build if there is a NuGet packages update detected in NuGet repository.
Prerequisites:

- .NET 4.0 should be installed on TeamCity server

To configure NuGet Dependency Trigger you need to:

- select NuGet version to use from the **NuGet.exe** drop-down list (if you have installed NuGet beforehand), or specify custom path to `NuGet.exe`;
- specify NuGet package source, if it's different from `nuget.org`;
- and enter package Id to check for updates.

Optionally, you can specify package version range to check for. If not specified, TeamCity will check for latest version.
Starting with TeamCity 7.1 you can also opt to trigger build if pre-release package version is detected by selecting corresponding check box.
Note that this is only supported for NuGet version 1.8 or newer.

See also:

Administrator's Guide: NuGet

Configuring 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 the build is started to run, all artifacts the builds depends on will be downloaded and placed in their configured target locations and then can be used by the build.

> The dependencies of the build can later be viewed on the build results page - Dependencies tab. This tab displays also 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 the build A.

See also:

> **Concepts**: Dependent Build
> **Administrator's Guide**: Accessing artifacts via HTTP | Snapshot Dependencies | Artifact Dependencies
> **External Resources**: http://ant.apache.org/ivy/ (additional information on Ivy)

Artifact Dependencies

- Configuring Artifact Dependencies Using Web UI
- Configuring Artifact Dependencies Using Ant Build Script

Configuring Artifact Dependencies Using Web UI

To add dependencies to a build configuration:

1. When creating/editing build configuration, open **Dependencies** page.
2. Click the **Add new artifact dependency** link and specify the following settings:

Option	Description
Depend on	Specify the build configuration that the current build configuration should depend on.
Get artifacts from	Specify the type of build, from which the artifacts should 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 specific build number or last finished build with specified tag. ⚠ • When selecting the build configuration, take your clean-up policy settings into account. Builds are cleaned and deleted on a regular basis, thus the build configuration could become dependent on a non-existent build. When artifacts are taken from a build with a specific number, then the specific build will not be deleted during clean-up. • If both dependency by sources and dependency by artifacts on a last finished build are configured for a build configuration, then artifacts will be taken from the build with the same sources.
Build number	*This field appears, if you have selected **build with specific build number** in the **Get artifacts from** list*. Specify here the exact build number of the artifact.
Build tag	*This field appears, if you have selected **last finished build with specified tag** in the **Get artifacts from** list*. Specify here the tag of the build which artifacts should be used. When resolving dependency, TeamCity will look for the last successful build with given tag and use its artifacts.
Artifacts Rules	Newline-delimited set of rules. Each rule must have following syntax: `[+:|-:]SourcePath[!ArchivePath][=>DestinationPath]` Each rule specifies the files to be downloaded form the "source" build. The *SourcePath* should be relative to the artifacts directory of the "source" build. The path can either identify a specific file, directory, or use wildcards to match multiple files. Ant-like wildcards are supported. Downloaded artifacts will keep the "source" directory structure starting with the first `*` or `?`. *DestinationPath* specifies the destination directory on the agent where downloaded artifacts should be placed. If the path is relative, it will be resolved against the build checkout directory. If needed, the destination directories can be cleaned before downloading artifacts. If destination path is empty, artifacts will be downloaded directly to checkout root.

	Basic examples:
	• Use *`a/b/**=>lib`* to download all files from *`a/b`* directory of the source build to the *`lib`* directory. If there is a *`a/b/c/file.txt`* file in the source build artifacts, it will be downloaded into the file *`lib/c/file.txt`*.
	• At the same time, artifact dependency *`*/.txt=>lib`* will preserve the directories structure: the *`a/b/c/file.txt`* file from source build artifacts will be downloaded to *`lib/a/b/c/file.txt`*.
	
	ArchivePath is used to extract downloaded compressed artifacts. Zip, jar, tar and tar.gz are supported. *ArchivePath* follows general rules for *SourcePath*: ant-like wildcards are allowed, the files matched inside the archive will be placed in the directory corresponding to the first wildcard match (relative to destination path)
	For example: *`release.zip!*.dll`* command will extract all .dll files residing in the root of *`release.zip`* artifact.
	
	Archive processing examples:
	• *`release-`***`.zip!`***`.dll=>dlls`* will extract *.dll from all archives matched *`release-*.zip`* pattern to the *`dlls`* directory.
	• *`a.zip!**=>destination`* will unpack entire archive saving path information.
	• *`a.zip!a/b/c/*/.dll=>dlls`* will extract all .dll files from *`a/b/c`* and its subdirectories, into the *`dlls`* directory, without *`a/b/c`* prefix.
	
	`+:` and **`-:`** can be used to include or exclude specific files from download or unpacking. As **`+:`** prefix can be ommited: rules are inclusive by default, and at least one inclusive rule is required. Order of rules is unimportant. For each artifact, most specific (with longest prefix before first wilcard symbol) rule is applied. When excluding a file, *DestinationPath* is ignored: file wont be downloaded at all. Files can also be excluded from archive unpacking. Set of rules applied to archive content is determined by set of rules matched by archive itself.
	
	Exclusive patterns examples:
	• `*/.txt=>texts`
	`-:bad/exclude.txt`
	Will download all *.txt files from all directories, excluding `exclude.txt` from `bad` directory
	• `+:release-`**`.zip!/`**`.dll=>dlls`
	`-:release-0.0.1.zip!Bad.dll`
	Will download and unpack all dlls from `release-*.zip` files to `dlls` directory. `Bad.dll` file from `release-0.0.1.zip` will be skipped
	• `*/.*=>target`
	`-:excl/*/.*`
	`+:excl/must_have.txt=>target`
	Will download all artifacts to `target` directory. Will not download anything from `excl` directory, but one file, called `must_have.txt`
	
	✓ Click the icon to invoke the Artifact Browser. TeamCity will try to locate artifacts according to specified settings and show then in a tree. Select required in the tree and TeamCity will place the paths to them into the input field.
	
	Artifacts placed under **.teamcity** directory are considered *hidden*. These artifacts are ignored by wildcards by default. If you want to include files from **.teamcity** directory for any purpose, be sure to add 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 - read more.

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.

For handling artifact dependencies between the builds this solution is more complicated then 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 `ivyconf.xml` file that contains some meta information about TeamCity repository. This file should 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 `ivy.xml` file in which define which artifacts should be downloaded 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_CONFIGURATION_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` should be replaced with the name of your organization.
- `YOUR_MODULE should` be replaced with the name of your project or module where artifacts will be used.
- `BUILD_CONFIGURATION_ID` should be replaced with id of the build configuration from where artifacts are downloaded. You can obtain this id from the links in your TeamCity server (you should take value of buildTypeId parameter). e.g. *bt20*
- `BUILD_REVISION` can be either build number or one of the following strings:
  - `latest.lastFinished`
  - `latest.lastSuccessful`
  - `latest.lastPinned`
- `ARTIFACT_FILE_NAME_WITHOUT_EXTENSION` file name or regular expression of the artifact without extension part.
- `ARTIFACT_FILE_NAME_EXTENSION` 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="${basedir}/ivyconf.xml" />
    <ivy:retrieve pattern="${basedir}/[artifact].[ext]"/>
  </target>
```

> ⚠ Please note that among ivy `commons-httpclient`, `commons-logging` and `commons-codec` should be in classpath of Ivy tasks.

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 whose only intended use is artifacts downloading 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 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 the pinned builds.

See also:

Concepts: Dependent Build

Snapshot Dependencies

Setting the dependency by other build's sources you can ensure that a build will start only after the one it depends from is run and finished. See details at Snapshot Dependency.

To add a snapshot dependency, on the **Dependencies** page of the build configuration settings, click the **Add new snapshot dependency** and specify the following options:

Option	Description
Depend on	Specify the build configuration that the current build configuration should 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 appropriate sources revisions exists. See also Suitable Builds below. However, when a 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 dependency will be set to a suitable finished build.
Only use successful builds from suitable ones	A new triggered build will only "use" successfully finished suitable builds as dependencies. If the latest finished "suitable" build is failed, it will be re-run.

Run this build if dependency has failed	If enabled and dependency fails, the build of this build configuration will still be run. The build status will be set to failed. If the option is not set, the build on the build configuration will fail with "failed to start" error without running.
Run build on the same agent	When enabled, and B snapshot-depends on A, then builds of B are run on the same agent, that build of A from the same build chain was run on. If build of B is already in the build queue, then TeamCity will not let any other build run on the agent before the build of B.

Suitable Builds

A "suitable" build in terms of snapshot dependencies is the one which uses the same sources snapshot. If the build configurations have the same VCS settings, this basically means the one with the same sources revisions. If VCS settings are different (VCS roots or checkout rules), then "same sources snapshot" revisions means revisions corresponding one to each other in terms of being current at some moment in time.

Please note that custom builds (those run with custom parameters) or personal builds are not considered "suitable" even when they use the same sources revision.

See also a note on certain VCS settings which might result in builds being not suitable, even though they have no changes.

See also:

> **Concepts**: Dependent Build

Configuring Build Parameters

Build Parameters such as configuration parameters, system properties and environment variables, provide you with flexible means of sharing settings within a single Build Configuration, managing compatible agents based on specific environment (see Configuring Agent Requirements) and a convenient way of passing generic or environment-specific settings into the build script. In this section:

- Configuration Parameters
  - Using references to configuration parameters
  - Example of configuration parameters usage
- System Properties and Environment Variables
  - Where system properties and environment variables are defined
  - What system properties and environment variables are passed to build process

Configuration Parameters

Configuration parameters provide a way to override some settings in a build configuration inherited from a template. They are never passed to a build.
As a rule, a build configuration created based on a template inherits all the settings defined in the template. You cannot edit these settings for the particular build configuration, whereas modifying them in the template will influence all configurations associated with this template.
However, you can bypass this behavior by means of configuration parameters. Please, refer to the Build Configuration Template to see which settings can be redefined by means of configuration parameters.

Using references to configuration parameters

To introduce a configuration parameter use `%ParameterName%` syntax in the template text fields thus providing means to change actual values of such parameters in associated build configuration. Once introduced, such parameter appears on the **Build Parameters** page of the build configuration template with undefined value.

You can either specify parameter's default value or leave it without any value.

Example of configuration parameters usage

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

To do so, perform the following steps:

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

`checkout.rules` parameter right in the "Associate with Template" dialog. Click "Associate".

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

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

System Properties and Environment Variables

System properties and environment variables provide you with a flexible means of sharing settings within a single Build Configuration, managing compatible agents based on specific environment (see Configuring Agent Requirements) and a convenient way of passing generic or environment-specific settings into the build script.
System properties and environment variables are name-value pairs that can be defined by the project administrator or provided by TeamCity as predefined properties.

Where system properties and environment variables are defined

There are several places where you can define system properties and environment variables:

- **Build Parameters** page of Build Configuration settings.
- **Run Custom Build** dialog, when launching a custom build.
- In dedicated `teamcity.default.properties` file, which should be put in the VCS root.
- In Agent Properties, specified in agent's `<Agent home>/conf/buildAgent.properties` file.

> To learn how define system properties and environment variables, refer to Defining and Using Build Parameters in Build Configuration and Project and Agent Level Build Parameters pages.

What system properties and environment variables are passed to build process

When TeamCity starts a build process the following set of environment variables is used:

- environment variables of the Build Agent process itself.
- env.* variables defined in the agent's `buildAgent.properties` file.
- env.* variables defined in the Build Configuration.
- env.* variables redefined in the Run Custom Build dialog.
- pre-defined environment variables.
- env.* variables from `teamcity.default.properties` file.

> Please note that the agent's environment variables can vary depending upon which user the agent process is running. The list of environment variables available on a specific build agent can be found on the **Environment variables** tab of the **Agent Details** page.

System properties passed to the script engine include:

- system.* properties defined in the agent's `buildAgent.properties` file
- system.* properties defined in the Build Configuration
- system.* properties redefined in the Run Custom Build dialog
- pre-defined system properties
- system.* properties that came from `teamcity.default.properties` file

> `env.` and `system.` prefixes are removed from the properties before passing them into environment/build script.

For details on defining and using parameters, refer to: Defining and Using Build Parameters in Build Configuration, 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, please refer to the Configuring Build Parameters page.
In this section:

Defining Build Parameters in Build Configuration

On the **Build Parameters** page of Build Configuration settings you can define 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 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 actual value. Before starting a build, TeamCity resolves all references with 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 just use it's 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. Agent-defined value will be used in the build.

If you want to prevent TeamCity from treating text in the percentage signs as reference to a property you can escape them by using 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 label pattern	`system.build.number` and Server Build Predefined Properties
Artifact dependency settings	only Predefined Server Build Properties

If you reference a build parameter in a build configuration, and it is not defined there, it becomes an agent requirement for the configuration. The build configuration will be run only on agents that have this property defined.

Using Build Parameters in VCS Labeling Pattern and Build Number

In Build number pattern and VCS labeling pattern you can use `%[env|system].property_name%` syntax to reference the properties that are known on the server-side. These are server and reference predefined properties and properties defined in the settings of the build configuration on **Build Parameters** page.
For example, VCS revision number: `%build.vcs.number%`.

Using System Properties in Build Scripts

Any system property (`system.<property name>`) can be referenced in a build script by the property name:

- For Ant, Maven and NAnt use `${<property name>}`
- For Gradle use `teamcity["<property name>"]`
- For MSBuild (Visual Studio 2005/2008 Project Files) use `$(<property name>)`

⚠

- Don't forget to leave the "`system.`" part out of the system properties.
- When using MSBuild, replace "." with "_" when you reference property names.

To get the file names that store the full set of properties, use can use `teamcity.build.properties.file` system property of `TEAMCITY_BUILD_PROPERTIES_FILE` environment variable. see 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

The predefined properties can originate from several scopes:

- Server Build Properties - the properties provided by TeamCity on the server-side in the scope of a particular build. An example of such property is a build number
- Agent Properties - properties provided by an agent. The properties are not specific to any build and characterize agent environment (for example, path to .Net framework).
- Agent Build Properties - properties provided on the agent side in the scope of a particular build. These properties are passed into a build (for example, path to a file with a list of changed files).

There is also a special kind of server-side build properties that can be used in references while defining other properties, but are not passed into the build. See Reference-only Server Properties below for the listing of such properties.

Server Build Properties

System Property Name	Environment Variable Name	Description
teamcity.version	TEAMCITY_VERSION	Version of TeamCity server. This property can be used to determine the build is run within TeamCity.
teamcity.projectName	TEAMCITY_PROJECT_NAME	Name of the project the current build belongs to.
teamcity.buildConfName	TEAMCITY_BUILDCONF_NAME	Name of the Build Configuration the current build belongs to.
build.is.personal	BUILD_IS_PERSONAL	Is set to `true` if the build is a personal one. Is not defined otherwise.
build.number	BUILD_NUMBER	Build number assigned to the build by TeamCity using the build number format. The property is assigned based on the build number format.
teamcity.build.id	none	Internal unique id used by TeamCity to reference builds.
teamcity.auth.userId	none	Generated username that can be used to download artifacts of other build configurations. Valid only during the build.
teamcity.auth.password	none	Generated password that can be used to download artifacts of other build configurations. Valid only during the build.

build.vcs.number.<simplified VCS root name>	BUILD_VCS_NUMBER_<simplified VCS root name>	Latest VCS revision included in the build for the root identified. See below for the *<simplified VCS root name>* description. If there is only a single root in the configuration, `build.vcs.number` property (without the root name) is also provided. ⚠ Please note that this value is a VCS-specific (for example, for SVN the value is a revision number while for CVS it is a timestamp) In versions of TeamCity prior to 4.0, different format for VCS revision number when specified in build number pattern was used: {`build.vcs.number.N`} where `N` is VCS root order number in the build configuration. If you still need this to work, you can launch TeamCity with special internal option: `teamcity.buildVcsNumberCompatibilityMode=true`

`<simplified VCS root name>` is the VCS root name with all non-alphanumeric characters replaced with _ symbol. Please ensure your VCS roots names differ sufficiently and are not mapped into the same simplified name (in this case the value of the property can get any).

Reference-only Server Properties

These are the properties that other properties can reference (only if defined on **Build Parameters** page), but that are not passed to the build themselves.

You can get the full set of reference-only server properties by adding `system.teamcity.debug.dump.parameters` property to the build configuration and examining "Available reference-only server properties" section in the build log.

The following sets of such properties exist:

- Dependencies Properties
- VCS Properties

Dependencies Properties

Properties provided by the builds the current build depends on (via snapshot or artifact dependency).

Dependencies properties have the following format:

```
dep.<btID>.<property name>
```

- `<btID>` — is the internal ID of the build configuration to get the property from. Only the configurations the current one has snapshot or artifact dependencies on are supported. Indirect dependencies configurations are also available (e.g. A depends on B and B depends on C - A will have C's properties available).
- `<property name>` — the name of the server build property of the build configuration with the given ID.

VCS Properties

These are the settings of a VCS roots attached to the build configuration.

VCS properties have the following format:

```
vcsroot.<simplified VCS root name>.<VCS root property name>
```

- `<simplified VCS root name>` — is the VCS root name as described above.
- `<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, `<simplified VCS root name>.` part can be omitted.

Properties marked by VCS support as `secure` (for example, passwords) are not available as reference properties.

Agent Properties

Agent-specific properties are defined on each build agent and vary depending on its environment. Aside from standard properties (for example, `os.name` or `os.arch`, etc. — these are provided by 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 lack of some property) and inside build scripts. For more information on how to reference these properties see Defining and Using Build Parameters in Build Configuration page.

The actual properties defined on agent can be reviewed on the **Agent Details** page.

Predefined Property	Description
agent.name	Name of the agent as specified in the `buildAgent.properties` 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	Path of Agent Work Directory.
agent.home.dir	Path of Agent Home Directory.
os.name	corresponding JVM property (see JDK help for properties description)
os.arch	corresponding JVM property
os.version	corresponding JVM property
user.country	corresponding JVM property
user.home	corresponding JVM property
user.timezone	corresponding JVM property
user.name	corresponding JVM property
user.language	corresponding JVM property
user.variant	corresponding JVM property
file.encoding	corresponding JVM property
file.separator	corresponding JVM property
path.separator	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's 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's 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]	This property is defined if the corresponding version(s) of Visual Studio is installed
VS[2003|2005|2008|2010]_Path	This property's value is the path to the directory that contains `devenv.exe`
teamcity.dotnet.nunitlauncher<version>	This property value is the path to the directory that contains the standalone NUnit test launcher, `NUnitLauncher.exe`. The version number refers to the version of .NET Framework under which the test will run. The version equals the version of .NET Framework and can have a value of 1.1, 2.0, or 2.0vsts.
teamcity.dotnet.nunitlauncher.msbuild.task	The property value is the path to the directory that contains the MSBuild task dll providing the NUnit task for MSBuild, Visual Studio (sln).

⚠

- Make sure to replace "." with "_" when using properties in MSBuild scripts. That is 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 can not be hidden or disabled.

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
teamcity.build.checkoutDir	none	Checkout directory used for the build.
teamcity.build.workingDir	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.
teamcity.build.tempDir	none	Full path of the build temp directory automatically generated by TeamCity. The directory will be cleaned after the build.
teamcity.build.properties.file	TEAMCITY_BUILD_PROPERTIES_FILE	Full name (including path) of the file containing all the `system.*` properties passed to the build. "system." prefix stripped off. The file uses Java properties file format (for example, special symbols are backslash-escaped).
teamcity.build.changedFiles.file	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 project or build agent level.

Project Level Build Parameters

TeamCity allows to define build parameters for **all** of the project's build configurations at one place - **Project Settings** -> **Parameters** tab.

Note that if build parameter P is defined in build configuration and a build parameter with the same name exists on project level, the following heuristics applies:
Case 1: Project A, Build Configuration from project A.
Parameters defined in build configuration have priority over 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 build configuration have priority over parameters with the same name defined in project A, and project-level parameters have priority over parameters with the same name defined in 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 parameters with the same names defined in template.

Agent Level Build Parameters

To define agent-specific properties edit the Build Agent's `buildAgent.properties` file (`<agent home>/conf/buildAgent.properties`). Refer to the Agent-Specific Properties page for more information.

When defining system properties and environment variables in `teamcity.default.properties` or `buildAgent.properties` file use the following format:

```
[env|system].<property_name>=<property_value>
```

For example: `env.CATALINA_HOME=C:\tomcat_6.0.13`

See also:

Administrator's Guide: Configuring Build Parameters | Defining and Using Build Parameters in Build Configuration | Predefined Build Parameters

Typed Parameters

When adding a build parameter (system property, environment variable, or configuration parameter) you can extend its definition with specification that will regulate parameter's control presentation and validation.
This specification is a parameter's "meta" information that is used to display parameter in Run custom build dialog. It allows to make 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 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 should it have, but it can be not obvious to a regular user. With build parameter's specification you can make your parameters more readable in "Run custom build" dialog. Currently you can present parameters in following forms:

Type	Description
Text	The default. Represents a usual text string without any extra handling
Checkbox	True/false otpion represented with a checkbox.
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 TeamCity web UI: it affects settings screens and Run Custom Build dialog where the field is presented with password fields. Also, the value is replaced in the build's **Build Parameters** tab and build log. The value is stored scrambled in the configuration files under TeamCity Data Directory. Please note that build log value hiding is implemented with simple search-and-replace, so if you have a trivial password of "123", all occurrences of "123" will be replaced, potentially exposing the password. 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.

- simple text field with ability to validate its value using regular expression;
- check box;
- select control;
- password field.

To add specification to a build parameter click **Edit** button in the **Spec** area when editing/adding a build parameter. All parameters specifications support a number of common properties, such as:

- **Label**: some text which should be shown near the control.
- **Description**: some text that is shown below the control containing explanatory note of the control use.
- **Display**: If *hidden* is specified, parameter will not be shown in **Run Custom Build** dialog, but will be sent to a build; if *prompt* is specified, TeamCity will always require review of parameter value, even if simply run button is clicked; if *normal* is selected, parameter will be shown as usually.

Depending on the specification's "type", there are additional settings.

Text	**Pattern**: In this field specify Java-style regular expression to validate field value.
Check box	**Check box name**: Title of the check box to be displayed in **Run Custom Build** dialog. **Checked value**/**Unchecked value**: Specify values the parameter should have depending on check box's state.
Select Control (**Enum**)	**Items**: Specify items for the list. Each item on new line. Use following syntax `label => value` or `value`. Click **Allow** if you want to enable multiple selection.

Manually configuring parameter's 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's specification

If you start editing a parameter that has some specification, you can see a link to its raw value in "Edit parameter" dialog. Click it to view the specification in its raw form (in service message format). To use this specification in another build configuration, just copy it from here, and paste in another configuration.

See also:

> **Administrator's Guide**: Configuring Build Parameters | Defining and Using Build Parameters in Build Configuration | Predefined Build Parameters

Configuring Agent Requirements

By specifying Agent Requirements for build configuration you can control on which agents the configuration will be run.

To add a requirement, click corresponding link and specify the following options:

Parameter Type	Specify the type of the parameter: system property, environment variable, or configuration parameter. For details on the types of parameters available in TeamCity, please refer to Configuring Build Parameters section.
Parameter Name	Specify the mandatory property or environment variable name.
Condition	Select condition from the drop-down list. **Some notes on how conditions work:** - **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.
Value	Is shown for some conditions that require value, for example: `equals`

Moreover, you can use the **Frequently used requirements** to quickly add a popular requirement.
Note, that the Agent Requirements page also displays the list of compatible and incompatible build agents for this build configuration, which is updated each time you modify the list of requirements. Possible reasons why build agent may be incompatible with this build configuration are described separately.

See also:

> **Concepts**: Agent Requirements | Build Agent | Build Configuration

Copying and Moving Build Configuration

TeamCity allows administrator to copy/move a build configuration:

- A copy duplicates parameters and 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.

To copy/move a build configuration, open its settings (for example, on the build configuration home page click **Edit Configuration Settings**) and click the **Copy**/**Move** button in the lower right corner of the page.

> ⚠ If the configuration contains non-shared VCS Roots options, and you are copying/moving the configuration into another project, TeamCity suggests to share such roots, or use copies of these VCS roots in new build configuration. In the latter case, TeamCity automatically creates a copy of non-shared VCS root for a project the configuration is copied/moved to. This new root will be also non-shared and will have an automatically generated name which looks like "Copy of <*original VCS Root's name*>".

Triggering a Custom Build

A build configuration usually has build triggers configured in it which automatically start new build each time the conditions are met, like scheduled time, or detection of VCS changes are detected, etc. However TeamCity allows to trigger a build manually whenever you need, and customize this build by adding properties, using specific changes, running the build on specific agent, etc.

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 build results page, go to the **Changes** tab, expand the required change and click the **Run build with this change** and proceed with the options in the **Run Custom Build** dialog.
- Use HTTP request to TeamCity to trigger a build

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 runs 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 following cases:
  - run a build for agent maintenance purposes (e.g. you can create a configuration to check whether agents function properly after 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).

In **General** options you can also specify that this particular build should be run as a personal one. Additionally, you can put the build to the top position in the Build Queue.

Dependencies

This tab is available only for builds that have artifact dependencies on other builds.
You can enforce rebuilding of all dependencies and select particular build from which artifacts should be taken.

Changes

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. Please note, that TeamCity displays only the changes earlier detected by TeamCity for the current build configuration VCS roots. If the VCS root was detached from the build configuration since the change occurred, there is no ability to run the build on such change. A limited number of changes is displayed. If there is an earlier change in TeamCity that you need to run build on, you can locate the change in the Change Log and use **Run build with this change** action.
This section is available only if you have permissions to access VCS roots for the build configuration and if build configuration has pending changes.

- **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.

> ℹ **Since TeamCity 7.1** if you have branches in your build configuration (or in snapshot dependencies of this build configuration), you can choose a branch to be used for the custom build in this dialog.

Properties

This tab show all configuration parameters currently defined for the build configuration. You can add new, edit, and delete additional properties/variables, or edit values of predefined ones.
These settings are available only if you have permissions to change system properties and environment variables for the build configuration.
When adding/editing/deleting properties and variables, note the following:

- For a predefined property/variable you can edit only its value.
- Only newly added properties/variables can be deleted. You cannot delete predefined properties.
- Click the **Reset** link next to the predefined property to reset its value to the default one.

Comment

Add optional comment to the build.

> ⚠ Please note that 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.

To create history builds, TeamCity always uses the VCS roots and settings that are now actual for the particular build configuration. If you delete a VCS root from the build configuration, you can no longer trigger history builds with changes which the root contained.

Triggering a build with custom artifact dependency

When you have artifact dependencies configured, at some point you may need to start a build with custom artifact dependencies. For example, if your build configuration A is configured to take artifacts from the last successful build of configuration B, and you want to include some specific artifacts from B, different from the last successful.

To do that, you can open build results page of build B you want to use artifacts from and click **Actions | Promote**.

See also:

Concepts: Build Queue | Dependent Build | Personal Build
Administrator's Guide: Configuring Build Triggers

Creating and Editing Projects

To create a new project:

1. On the **Administration** | **Projects** page, click the **Create project** button.
2. On the **Create New Project** page, specify the project name and optional description.
3. Click **Create**. An empty project is created.

> ℹ To configure an existing project, select the desired project in the list, and click **Edit**.

4. Create build configurations (select build settings, configure VCS settings, and choose build runners) for the project.
5. Assigning Build Configurations to Specific Build Agents

See also:

Administrator's Guide: Creating and Editing Build Configurations

Archiving Projects

If a project is not in active development state, you can *archive* it from its settings page. When *archived*:

- All project's build configurations are automatically paused.
- Automatic checking of changes in project's VCS roots is disabled.
- 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.

By default, permissions to archive projects are given to project and system administrators.
If you need to unarchive a project, you can do it from the **Administration** | **Projects** | **Archived projects** tab where all archived projects are displayed.

See also:

Concepts: Project | Build Configuration

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 statistics charts, refer to Statistic Charts. This page describes how to modify pre-defined project-level charts, and add custom charts on 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, project's **Statistics** tab shows charts for all build configurations in the current project, which have coverage, duplicates or inspections data. However, you can disable charts of particular type at all, or specify build configurations to be used in the charts.

To modify pre-defined project level charts you need to configure `<TeamCity Data Directory>/config/<project_name>/` **`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 following syntax:

```
<coverage-graph enabled="false"/>
```

In this example, all code coverage charts will be removed from project's Statistics page.

Showing charts only for specific build configurations on project level

To show code coverage chart, which relates only to particular build configuration, use the following syntax:

```
<coverage-graph enabled="true">
    <build-type id="bt234"/>
    <build-type id="bt236"/>
</coverage-graph>
```

where **bt234** and **bt236** values are build configuration identifiers.
However, note that denoted build configurations should contain code coverage data for 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:

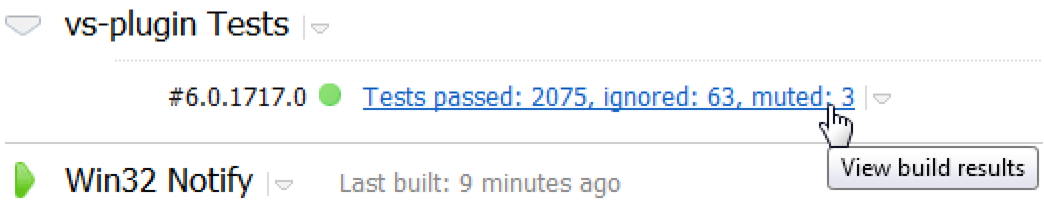
Concepts: Code Coverage | Code Inspection | Code Duplicates
User's Guide: Statistic Charts
Extending TeamCity: Build Script Interaction with TeamCity

Muting Test Failures

TeamCity provides a way to "mute" any of the currently failing tests so they will not affect build status for future builds.

When a test is muted, it is **still run** in the future builds, but its failure does not fail the build (by "at least one test failed" build failure condition). The test can be unmuted manually on specific date or after successful run. Also, tests can be muted only in a single build configuration or in all the build configurations of a specific TeamCity project.

Your build script might need adjustment to make build green when there are failing but muted tests. Please ensure that build does not fail because of other build failure conditions (like "Fail if build process exit code is not zero") in case the only errors encountered were tests failures. See also related issue TW-16784.



vs-plugin Tests

#6.0.1717.0 ● Tests passed: 2075, ignored: 63, muted: 3

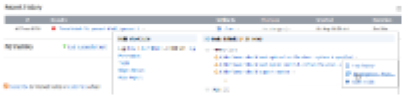
View build results

Win32 Notify Last built: 9 minutes ago

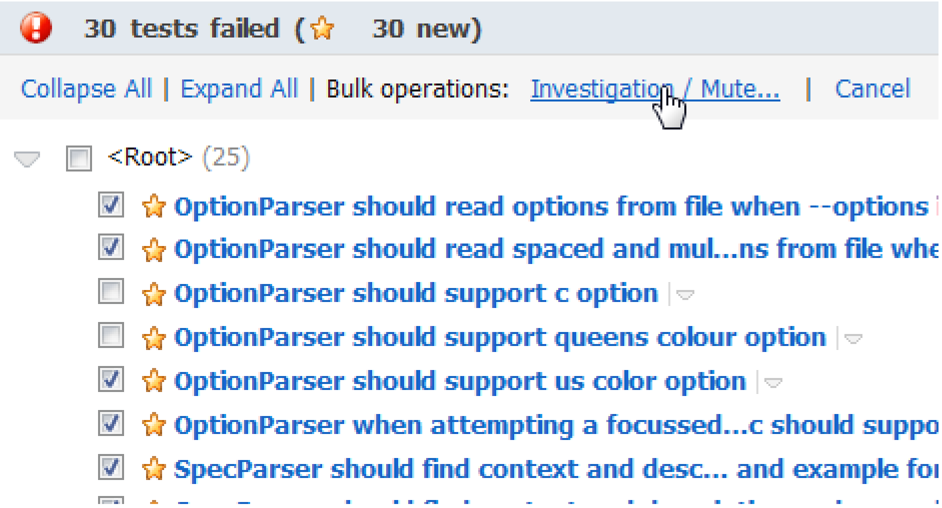
How to mute tests

Only users with the **Mute/unmute problems in project** permission can perform it. By default, these are **Project administrator** and **System administrator**
You can mute a test failure from:

- **Projects** page
- Project overview page
- Build Configuration overview page
- Current Problems tab



On the build results page you can select several test failures (or all) to be muted or unmuted:



30 tests failed (☆ 30 new)

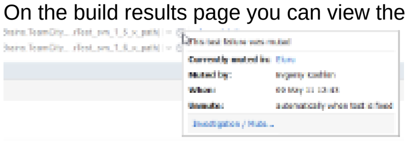
Collapse All | Expand All | Bulk operations: Investigation / Mute... | Cancel

□ <Root> (25)

☑ ☆ OptionParser should read options from file when --options
☑ ☆ OptionParser should read spaced and mul...ns from file whe
□ ☆ OptionParser should support c option
□ ☆ OptionParser should support queens colour option
☑ ☆ OptionParser should support us color option
☑ ☆ OptionParser when attempting a focussed...c should suppo
☑ ☆ SpecParser should find context and desc... and example foi

Note, that you can start investigation of the problem simultaneously with muting the failure. When muting a test failure you can specify conditions when it should be unmuted: on a specified date or when it is fixed. Alternatively, you can unmute it manually.

- On the build results page you can view the list of muted test failures, their stacktraces and details about mute status:



This test failure was muted

Currently muted in: Fluro

Muted by: Evgeny Koshkin

When: 03 May 11 13:03

Unmute: automatically when test is fixed

Investigation / Mute...

- From the Project Home page you can navigate to the **Muted Tests** tab to view all the test failure muted in all build configurations within project.

This feature is useful when some tests fail for some known reason, but it is currently not possible to fix them. For example, responsible developer is on vacation, or you are waiting for the system administrators to fix the environment, or the test is failing intentionally, for example, if required functionality is not yet written (TDD). In these cases you can mute such failure and avoid unnecessary disturbance of other developers.

Build Dependencies Setup

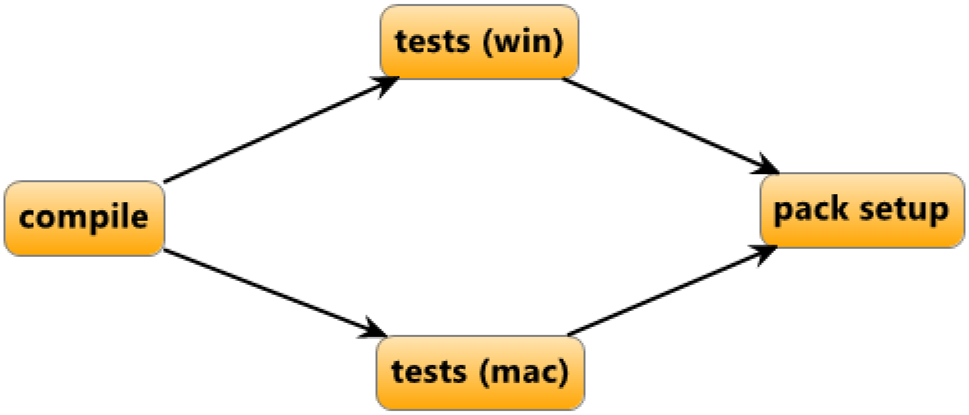
This page is intended to give you the general idea on how dependencies work in TeamCity.

Introduction

In many cases it is convenient to use 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 and, 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:



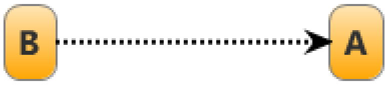
Where *compile*, *tests (win)*, *tests (mac)* and *pack setup* are build configurations, and naturally tests **depend on** compilation, which means they should wait till compilation is ready.

In this section:

        - Trigger on changes in snapshot dependencies
        - Parameters in dependent builds
    - Miscellaneous notes on using dependencies

Basics

Generally known as "build pipeline" in TeamCity 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):

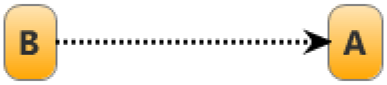
	A build configuration.
	Snapshot dependency between 2 build configurations. Note, that the arrow shows the sequence of triggering build configurations, the build chain flow, meaning that B is executed before A. However the dependencies are configured in the opposite direction (A snaphot-depends on B). The arrows are drawn this way because in 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 output of one build in another, while the second one can trigger builds from several build configurations in specific order, but on the same sources.
These two dependencies are often configured together, because artifact dependency doesn't affect the way builds are triggered, while 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

Artifact dependency allows to reuse the output of one build (or a part of it) in another.



If build configuration **A** has 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 **Build from the same chain** option is selected in artifact dependency settings, TeamCity ensures that artifacts are downloaded from the same-sources build.

Snapshot Dependencies

Snapshot dependency is a dependency between two build configurations that allows to launch builds from both build configurations **in 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 a 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 first run an integration build and after that a release build, if the first one was successful.

Another case is when your tests take too much time to run, so you have to extract them into separate build configuration, but you also need to make sure the same sources snapshot is used.

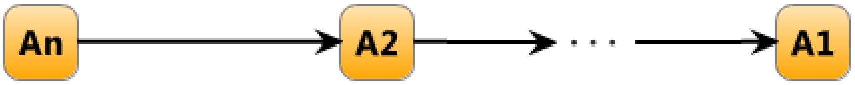
Build Chains in TeamCity UI

Once you have snapshot dependencies defined and at least 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 do snapshot dependencies work?

The easiest way to get an idea of how snapshot dependencies work is to think of module dependencies, because these concepts are similar. However, let's start with the basics.
Assume, we have a build chain:



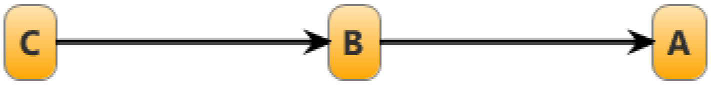
Here are the main rules:

1. If 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 chain will use the same sources snapshot, i.e. with explicit specification of the sources revision, that is calculated at the moment when build chain is added to the queue.

Now let's go into details and examples.

Example 1

Assume we have following build chain with no extra options - plain snapshot dependencies.



What exactly happens when build A is triggered?

1. TeamCity resolves the whole build chain and queues all builds - A, B and C. TeamCity knows that builds are to run in 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 builds added to queue TeamCity starts checking for changes in the entire build chain and synchronizes them - all builds have to start with the same sources snapshot.

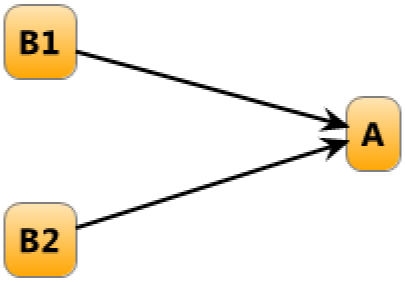
   > Note, that if the build configurations connected with snapshot dependency share the same set of VCS roots, all builds will run on the same sources. Otherwise, if the VCS roots are different, changes in the VCS will correspond to the same moment in time.

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

Instead of enforcing running 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. Matching queued builds will not be run and will be dropped from the queue; instead TeamCity will link the dependency to those 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 build chain modifies system environment, and the next build relies on that system state and thus has to run on the same build agent.

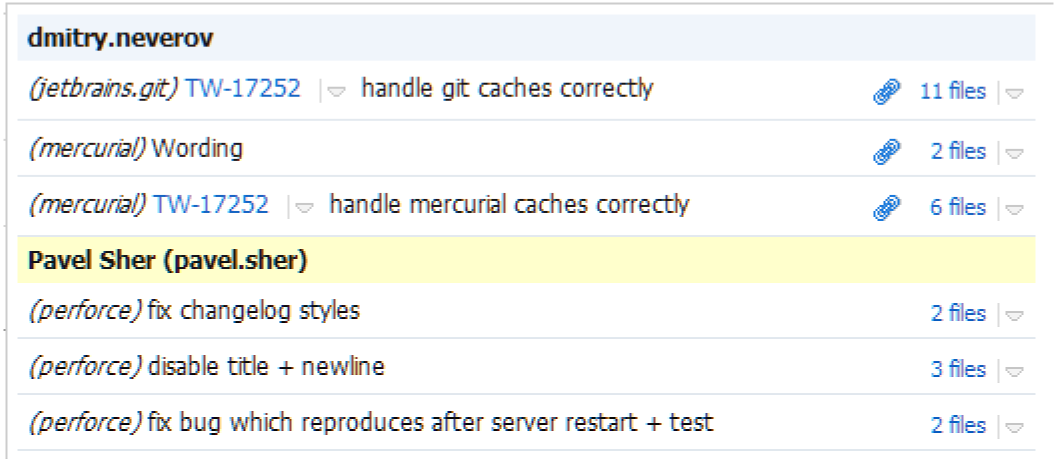
Trigger on changes in snapshot dependencies

Another option that alters triggering behavior within a build chain you can find in 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 anyhow change the order in which builds are executed. 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 of this setting affects pending changes of build configuration, builds changes in builds history, change log and issue log. Changes from dependencies are marked with 🔗. For example:



dmitry.neverov

(jetbrains.git) TW-17252 | handle git caches correctly — 11 files

(mercurial) Wording — 2 files

(mercurial) TW-17252 | handle mercurial caches correctly — 6 files

Pavel Sher (pavel.sher)

(perforce) fix changelog styles — 2 files

(perforce) disable title + newline — 3 files

(perforce) fix bug which reproduces after server restart + test — 2 files

With this setting enabled, "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 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 build chain flow and not vice versa.
For the details on how to use parameters of the previous build in chain, please 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 artifacts downloaded by this configuration from other builds can be cleaned or not. This setting is available on the cleanup policies page.

Changing Build Status Manually

Overview

Starting with TeamCity 7.1 a user with appropriate permissions can change status of a build manually, i.e. make it either failed or successful (issue TW-2529).

The corresponding action is available in the Build Actions popup on the build results page.

Marking build as successful

You may want to make build successful to:

- Change "last successful build" anchor when using the feature "Build failure conditions". I.e. if your last build failed because of the incorrect value of a metric, and this new value is valid, you may mark this build as a successful anchor.
- Allow using an incorrectly failed build with good artifacts in "last successful" dependencies.
- For a running personal build - mark current failures non-relevant to allow pre-tested commit to pass (if user has permission to do this).

"Mark as successful" action is not available for Failed to Start Builds.

Marking build as failed

You may want to mark a build as failed when:

- The build has some problem which didn't affect the final build status.
- There is a known problem with the build, and it should be ignored by your QA team.
- You've mistakenly marked the build as successful manually.

Permissions

By default, the permission to change build status is granted to **Project Administrator**.

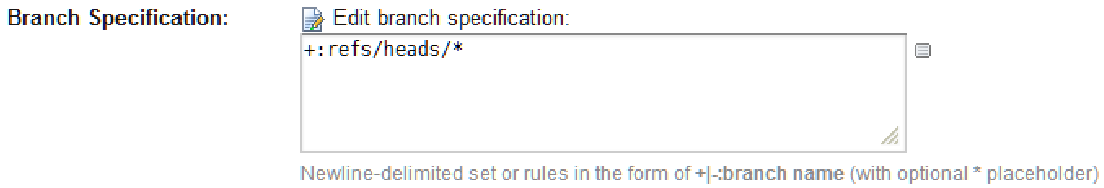
Working with Feature Branches

Feature Branches in distributed version control systems (DVCS) like Git and Mercurial allow you to work on a feature independently from the repository and commit all the changes for the feature onto the branch, merging the changes back when your feature is complete. This approach brings a number of advantages to software development teams, however in continuous integration servers that do not have dedicated support for it, it also causes a number of problems, like constant build configurations duplication, poor visibility and in the end loss of control of the process.

In TeamCity 7.0 feature branches are partially supported by means of the 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. However, TeamCity 7.1 brings feature branches support on a whole new level, allowing to automate the process, and ensuring visibility of branches all over the interface.

Configuring branches

To start working with branches, you need to tell TeamCity which ones of them you want to be monitored. This is done in a build configurations right in the VCS root.
The syntax of branch names specification is similar to checkout rules:



Branch Specification: Edit branch specification:

```
+:refs/heads/*
```

Newline-delimited set or rules in the form of **+|-:branch name** (with optional * placeholder)

For further details on branches specification, please refer to Git and Mercurial VCS roots description.
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 TeamCity interface you'll see `feature1` only as a branch name.
If you want shortened branch labels in builds, you can use extended syntax of branch specification like this:

```
+:refs/heads/release-(7.0)
+:refs/heads/release-(7.1)
```

In this case, TeamCity will use label `7.0` for builds from the `refs/heads/release-7.0` branch and `7.1` for builds from `refs/heads/release-7.1`.

Specification supports comments – lines started with `#`.
In order to use brackets in the branch name you need to escape them. To specify escaping symbol put the following as a first line in the specification:

```
#! escape: \
```

Once you've done branch specification, TeamCity will start to monitor these branches for changes. If your build configuration has VCS trigger and a change is found in some branch, TeamCity will trigger a build in this branch.
From build configuration home page you'll also be able to filter history, change log, pending changes and issue log by branch name. Also branch names will appear in custom build dialog, so you'll be able to manually trigger a custom build on a branch too.

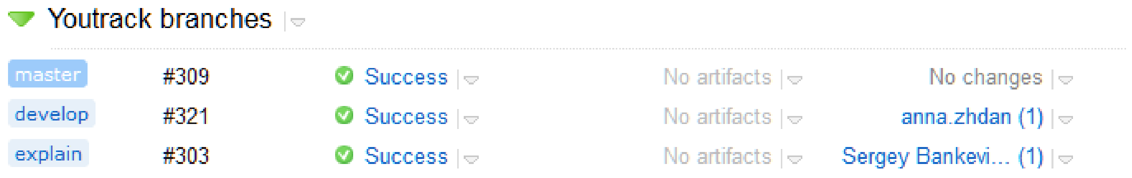
Branch specification & default branch

When configuring Git or Mercurial VCS root, you need to specify "Branch name", which will define so-called "default branch". It is used in situations when branch name was not specified. For example, if someone clicks on a **Run** button TeamCity will create build in the default branch.

Note that you can also use parameter in branch specification.

Builds

Builds from branches are easily recognizable in TeamCity UI, because they are marked with a special label:



Youtrack branches

master	#309	✓ Success	No artifacts	No changes
develop	#321	✓ Success	No artifacts	anna.zhdan (1)
explain	#303	✓ Success	No artifacts	Sergey Bankevi... (1)

You can also filter history by a branch name if you're interested in a particular branch.

TeamCity assigns a branch label to the builds from the default branch too.

Changes

For each build TeamCity shows changes included in it. For builds from branches changes calculation process takes branch into account and presents you with changes relevant to the build branch. Changelog with it's 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.

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, branch filter is available on test details page. So you can see a history of test passes or failures in a single branch.

Triggers

VCS trigger is fully aware of branches and will trigger build once a check-in is detected in a branch. All VCS trigger options like, per-checkin triggering, quiet period, triggering rules are directly available for builds from branches.

Dependencies

If build configuration with branches has snapshot dependencies on other build configurations, when a build in a branch is triggered, all builds from the chain will be marked with this branch too.

It is currently not possible to configure artifact dependencies to retrieve artifacts from a build from a specific branch, artifact dependencies always use builds from default branch. The same applies to finish build trigger. It will only watch for finished builds from default branch.

Notifications

All notification rules except "My changes" will only notify for builds from default branch. At the same time "My changes" rule will work for builds from all available branches.

Build configuration status

Build configuration status is calculated based on builds from default branch only. Consequently per-configuration investigation works for builds from default branch. For example, successful build from non-default branch won't remove per-configuration investigation. But successful build from default branch will.

Multiple VCS roots

If your build configuration uses more than one VCS root and in both VCS roots you have specified branches to monitor, the way how builds are triggered is more complicated.

VCS Trigger groups branches from several VCS roots by the part matched by star. When some root doesn't have branch from the other root, its default branch is used. For example you have 2 VCS roots, both have default branch `refs/heads/master`, first root has branch specification `refs/heads/7.1/*` and changes in branches `refs/heads/7.1/feature1` and `refs/heads/7.1/feature2`, second root has 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
refs/heads/master	refs/heads/master
refs/heads/7.1/feature1	refs/heads/devel/feature1
refs/heads/7.1/feature2	refs/heads/master

Build parameters

For Git & Mercurial TeamCity provides additional build parameters with names of VCS branches known at the moment of build starting. If a build took a revision from the `refs/heads/bugfix` branch, TeamCity will add a configuration parameter with the following name: `teamcity.build.vcs.branch.<simplified VCS root name>`

Where `<simplified VCS root name>` is the name of the VCS root where all non-alpha numeric characters are replaced with `{_}`.

In addition to this parameter, VCS branch is now shown on build changes page.

See also:

Administrator's Guide: Git (JetBrains) | Mercurial

Managing User Accounts, Groups and Permissions

Before creating and managind user accounts, groups and changing user's permission, we recommend you to read following pages in the Concepts section:

- 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.

Managing Users and User Groups

On this page:

- Managing Users
  - Creating a new user
  - Editing user account
  - Assigning roles to users
- Managing User Groups
  - Creating a new group
  - Editing Group settings
  - Adding multiple users to a group

Managing Users

Creating a new user

The **Create user account** link is available on the **Administration** | **Users** page only if the default authentication scheme is used. For NT and LDAP authentication schemes, the list of users coincides with the domain user database.
When creating a user account, only username and password are required. Any new user is automatically added to the All Users group and inherits roles and permissions defined for this group.
If you don't use per-project permissions, you can specify here whether a user should have administrative permissions or not. Otherwise, you can assign roles to new user later.

Editing user account

To edit/delete a user account click its name on the **Users** tab of the **Administration** | **Users** page.

General settings and Version Control Username Settings
Depending on the authentication scheme, username can be editable or read-only. If the default authentication scheme is selected, you can modify user's name. If another authentication schemes is used, the user name can be edited only by the system administrator. If needed, modify here user's full name, email address and password.
The *Version Control Username Settings* pane allows to view default usernames for different VCS used by the current user. The names set here will be used to show builds with changes committed by a user with such VCS username on the My Changes page. Also, such builds can be highlighted on the Projects page, etc.
Groups
Use this tab to review the groups the user belongs to, and add/remove user from groups.
Roles
*This tab is available only if per-project permissions are enabled at **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

> ℹ To be able to grant roles to users on per-project basis, enable per-project permissions on the **Administration**|**Global Settings** page.

There are several ways to assign roles to one or several users:

- To assign a role to a specific user, on the **Users** tab for the user click *View roles* in the corresponding column. In the Roles tab, click **Assign role**.
- To assign a role to multiple users, on the **Users** tab select users and click *More Actions - Assign roles to the selected users*.
- To assign a role to all users in a group, on the **Groups** tab click *View roles* for the group in question, then assign role on a group level. When assigning a role, you can:
- Select whether a role should be granted globally, or in particualr projects.
- Replace existing roles with newly selected. This will remove all roles assigned to user(s)/group and replace them with the selected one instead.

Managing User Groups

Creating a new group

Open the **Administration** | **Groups** page. Click **Create new group**.
When creating a group, you can select 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 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. For a group 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.

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 View roles permissions link.
You can also specify notification rules to be applied to all users in current group. To learn more about notification rules, please refer to Subscribing to Notifications.

Adding multiple users to a group

On the **Users and Groups** page, select the users, click the **Add selected users to groups** link, and check the groups where these users should be added. Note, that all these users will inherit 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 seach 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 existing roles and create new ones on the **Administration** | **Users and Groups** | **Roles** tab.
At this page you can:

- Create new roles by clicking corresponding link.
- Delete roles.
- Add/delete permissions from existing roles.
- Include/exclude one role permissions to another.
  Note, that these settings are global.
  Moreover, roles and permissions are described and can be modified in the `roles-config.xml` file stored in `<TeamCity Data Directory>/config` directory.

See also:

Concepts: Role and Permission

Customizing Notifications

TeamCity provides a wide range of notification possibilities to keep developers informed about the status of their projects. Notifications can be sent by e-mail, Jabber/XMPP instant messages or can be displayed in the IDE (with the help of TeamCity plugins) or the Windows system tray (using TeamCity Windows tray notifier). Each user can select the events to receive notifications. The notification messages can be customized globally on per-server basis.

Notifications can also be received via Atom/RSS syndication feeds, but since feed use "pull" model for receiving notifications instead of "push", some of the approaches are different for the feeds.

Notifications Lifecycle

TeamCity supports a set of events that can generate user notifications (such as build failure, investigation state changes, etc). On event occurrence, for each notificator type, TeamCity processes notification settings for all the users to define users that the notification should be sent to.

When the set of users is determined, TeamCity fills the notification model (the objects relevant to the notification as "build", investigation data,

etc.) and evaluates a notification template that corresponds to the notification event.
The template uses the data model objects to generate output values (e.g. notification message text). The output values are then used by the notificator to send the message. Each notificator supports a specific set of the output values.

Please note that the template is evaluated once for an event which means that notification properties cannot be adjusted on per-user basis.

The output values defined by the template are then used by the notificator to send notification to the selected users.

Customizing Notifications Templates

Notification Templates Location
Each of the bundled notificators has a directory under `<TeamCity data directory>/config/_notifications/` which stores FreeMarker (.ftl) templates. There are also .dist files that store default templates. Each notification type evaluates a template file with corresponding name.
The template files can be modified while the server is running. By default server checks for changes in the files each 60 seconds, but this can be changed by setting `teamcity.notification.template.update.interval` internal property to the desired number of seconds.

If there an error occurs during template evaluation, TeamCity logs the error details into `teamcity-notifications.log`. There can be non-critical errors that result in ignoring part of the template or critical errors that result in inability to send notification at all. Whenever you make changes to the notification templates please ensure the notification can still be sent.

This document doesn't describe the FreeMarker template language, so if you need a guidance on the FreeMarker syntax, please refer to the corresponding template manual at http://freemarker.org/docs/dgui.html.

TeamCity notificators use templates to evaluate output values (global template variables) which are then retrieved by name. The following output values are supported:

Email Notificator

- **subject** - subject of the email message to send
- **body** - plain text of the email message to send
- **bodyHtml** - (optional) HTML text of the email message to send. It will be included together with plain text part of the message
- **headers** - (optional) Raw list of additional headers to include into email. One header per line. For example:

```
<#global headers>
  X-Priority: 1 (Highest)
  Importance: High
</#global>
```

Jabber

- **message** - plain text of the message to send

IDE Notifications and **Windows Tray Notifications**

- **message** - plain text of the message to send
- **link** - URL of the TeamCity page that contains detailed information about the event

The Atom/RSS feeds template differs from the others. For the details, please refer to the dedicated section.

For the template evaluation TeamCity provides the default data model that can be used inside the template. The objects exposed in the model are instances of the corresponding classes from TeamCity server-side open API.
The set of available objects model differs for different events.
You can also add your own objects into the model via plugin. See Extending Notification Templates Model for details.

Here is an an example description of model (the code can be used in IntelliJ IDEA to edit the template with completion):

```
<#-- @ftlvariable name="project" type="jetbrains.buildServer.serverSide.SProject" -->
<#-- @ftlvariable name="buildType" type="jetbrains.buildServer.serverSide.SBuildType" -->
<#-- @ftlvariable name="build" type="jetbrains.buildServer.serverSide.SBuild" -->
<#-- @ftlvariable name="agentName" type="java.lang.String" -->
<#-- @ftlvariable name="buildServer" type="jetbrains.buildServer.serverSide.SBuildServer" -->
<#-- @ftlvariable name="webLinks" type="jetbrains.buildServer.serverSide.WebLinks" -->

<#-- @ftlvariable name="var.buildFailedTestsErrors" type="java.lang.String" -->
<#-- @ftlvariable name="var.buildShortStatusDescription" type="java.lang.String" -->
<#-- @ftlvariable name="var.buildChanges" type="java.lang.String" -->
<#-- @ftlvariable name="var.buildCompilationErrors" type="java.lang.String" -->

<#-- @ftlvariable name="link.editNotificationsLink" type="java.lang.String" -->
<#-- @ftlvariable name="link.buildResultsLink" type="java.lang.String" -->
<#-- @ftlvariable name="link.buildChangesLink" type="java.lang.String" -->
<#-- @ftlvariable name="responsibility"
type="jetbrains.buildServer.responsibility.ResponsibilityEntry" -->
<#-- @ftlvariable name="oldResponsibility"
type="jetbrains.buildServer.responsibility.ResponsibilityEntry" -->
```

TeamCity notification properties

The following properties can be useful to customize the notifications behaviour:
`teamcity.notification.template.update.interval` - how often the templates are reread by system (integer, in seconds, default 60)
`teamcity.notification.includeDebugInfo` - include debug information into the message in case of template processing errors (boolean, default false)
`teamcity.notification.maxChangesNum` - max number of changes to list in e-mail message (integer, default 10)
`teamcity.notification.maxCompilationDataSize` - max size (in bytes) of compilation error data to include in e-mail message (integer, default 20480)
`teamcity.notification.maxFailedTestNum` - max number of failed tests to list in e-mail message (integer, default 50)
`teamcity.notification.maxFailedTestStacktraces` - max number of test stacktraces in e-mail message (integer, default 5)
`teamcity.notification.maxFailedTestDataSize` - max size (in bytes) of failed test output data to include in a single e-mail message (integer, default 10240)

See also TW-8621 on including build log messages into the template.

Syndication Feed Template

The template uses different approach to configuration from other notification engines.

The default template is stored in the file: `<TeamCity data directory>/config/default-feed-item-template.ftl`. This file should never be edited: it is overwritten on every server startup with the default copy. To specify a new template to use, copy the file under the name `feed-item-template.ftl` into the same directory. This file can be edited and will not be overwritten. It will be used by the engine if present.

The template is a FreeMarker template and can be freely edited.

You can use several templates on the single sever. The template name can be passed as a URL parameter of the feed URL.

During feed rendering, the template is evaluated to get the feed content. The resultant content is defined by the global variables defined in the temple.

See the defualt template for an example of available input variables and output variables.

See also:

User's Guide: Subscribing to Notifications

Managing Licenses

In this section:

- Licensing Policy
- Enterprise License Expiration

- Third-Party License Agreements

Licensing Policy

There are two editions of TeamCity: Professional and Enterprise. Please see the section on the differences.
This section describes related licensing questions.

Professional edition does not require any license key and can be used free of charge.
Enterprise edition requires a license key.
Each additional **Build Agent** above bundled 3 requires a license key in both editions.

The Enterprise edition requires one of the following types of licenses:

- **Commercial** — no expiration date
- **Evaluation** — has an expiration date and provides an unlimited number of agents and build configurations. To obtains the evaluation license please use a link on TeamCity download page. 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.
- **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.

Each TeamCity edition comes bundled with 3 agents. More Build Agents can be added with separate licenses. The agent licenses can be used with either TeamCity edition (Enterprise and Professional). For more information about purchasing agent licenses, refer to the product page.
When there are more agents authorized then the agent licenses available, the server stops to start any builds and displays a warning message to all users in the web browser.

TeamCity licenses are perpetual for the TeamCity version they were issued for. This means that you can run given TeamCity version with purchased licenses for unlimited time.
In addition, each TeamCity license (including Enterprise Server and Agent) has a **maintenance period** (generally 1 year). The license key is valid with any TeamCity version released within the maintenance period.

Before you upgrade to newer TeamCity version, please check validity of the existing licenses with the new version.
If new TeamCity server release date is not covered by the maintenance period of some licenses, corresponding licenses will not be valid with the TeamCity version and would need an upgrade.

Please note that TeamCity support covers only the latest TeamCity major version released, so regular upgrades are recommended.

If you have any licensing-related questions, please contact our sales department.

Entering new license keys and reviewing currently used ones (including license issue date and maintenance period) can be done at **Administration** > **Licenses** page of TeamCity web UI. By default, only users with System Administrator role can access 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.

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.

Adding Build Agents

If you require more Build Agents, you can purchase additional Build Agents licenses.

See also:

Concepts: Edition | Build Agent
Licensing: Licensing & Upgrade

Enterprise License Expiration

If build configurations number exceed 20 when the server is working in Professional edition mode, the server stops to start any builds and displays a warning message to all users in the web browser.
Please see TeamCity Editions for details.

See also:

Concepts: TeamCity Editions
Administrator's Guide: Licensing Policy
License Agreement: http://www.jetbrains.com/teamcity/buy/license.html

Third-Party License Agreements

The following is an alphabetical list of third-party libraries distributed with TeamCity:

Product	License
Acegi Security	Apache
Apache Ant	Apache
Apache Commons libraries	Apache
Apache HttpComponents	Apache
Apache Lucene	Apache
Behaviour	BSD
Byteman	GNU LGPL
CassiniDev	Ms-PL
Core4j	Apache 2.0
cglib	Apache
CVS client library	CDDL
CyberNekoHTML Parser	Apache-style
DbUnit	GNU LGPL
DHTML Tip Message	DHTML Tip Message
EhCache	Apache
Expat XML Parser Toolkit	MIT
FormattedDataSet API	BSD

FreeMarker	BSD-style
Ganymed SSH-2 for Java	BSD-style
Google Collections Library	Apache
Highlight.js	BSD-like
HSQLDB	BSD
Ivy	Apache
Jakarta	Apache
Jakarta-ORO	Apache
JAMon	BSD-like
JAXB reference implementation	CDDL v1.0
Java Native Access	LGPL
Java Service Wrapper, version 3.2.3	Java Service Wrapper 3.2.3 License
JCIFS	GNU LGPL
JDom	JDom
Jersey	CDDL v1.0, CDDL v1.1
JFreeChart	GNU LGPL
JHighlight	CDDL
jMock	jMock
JNIWrapper	JNIWrapper
jQuery	MIT
jQuery Flot plugin	MIT
jQuery UFD plugin	MIT
Joda Time	Apache
JSch	BSD-Style
JUnit	CPL
J2SSH Maverick	J2SSH Maverick
Log4j	Apache
Log4net	Apache
Maven	Apache
Microsoft.Web.Infrastructure	MVC-3-EULA
Mono.Cecil	MIT/X11
NanoContainer	NanoContainer
NUnit	NUnit
OData4j	Apache
pack:tag	LGPL
Paul Johnson's MD5	BSD
PicoContainer	PicoContainer
PocketHTTP	Mozilla
PocketXML-RPC	Mozilla

PostgreSEL Data Base Management System	BSD
Prototype	MIT
Raphaël	MIT
Rome	Apache
RouteMagic	MS-PL
Script.aculo.us	MIT License
SilverStripe Unobtrusive Javascript Tree Control	BSD
Shaj	Apache
Slf4j	MIT
Smack	Apache
Spring Framework	Apache
Code snippets from Subclipse	EPL
SVNKit	TMate Open Source
Tomcat	Apache
Tom Wu's jsbn	BSD
Trove4j	GNU LGPL
Underscore.js	MIT
user-agent-utils	BSD
Waffle	EPL
WebActivator	MS-PL
Xerces2 Java Parser	Apache
XML Pull Parser	IU Extreme! Lab
XML-RPC.NET	MIT X11
XStream	XStream

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).
Shaj is Copyright Cenqua Pty Ltd.
SilverStripe Unobtrusive Javascript Tree Control Copyright © 2006-7, SilverStripe Limited - www.silverstripe.com.

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.

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.

> ℹ **Note**
> You can also use the condition **contains**, however, it may include more than one specific build agent (e.g. a build configuration with a requirement **agent.name contains** Agent10, will run on agents named **Agent10**, **Agent10a**, and **Agent10b**).

4. Click the **Add requirement for a property** link and add the required property, condition, and value. For example, if you have several Linux-only builds, you can add the **os.name** property and set the **starts with** condition and the **linux** value.

> ✓ On the **Agent Requirements** page, click the **Frequently used requirements** link to add the regularly used requirements.

See also:

Concepts: Build Agent | Agent Requirements | Run Configuration Policy
Administrator's Guide: Triggering a Custom Build

Patterns For Accessing Build Artifacts

This section covers URL patterns that you may use to download build artifacts from outside of TeamCity.
See Accessing Server by HTTP on basic rules covering HTTP access from scripts.

You may also download artifacts from TeamCity using Ivy dependency manager.
If you need to access the artifacts in your builds, consider using TeamCity's built-in Artifact Dependency feature.

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

Individual files can be downloaded with the help of REST API which provides more rich build selection facilities.

There are also other URLs exposing build's artifact that you can use. These are mostly preserved for backward-compatibility with previous TeamCity versions:

To download artifacts of the latest builds (last finished, successful or pinned), use the following paths:

```
/repository/download/BUILD_TYPE_ID/.lastFinished/ARTIFACT_PATH
/repository/download/BUILD_TYPE_ID/.lastSuccessful/ARTIFACT_PATH
/repository/download/BUILD_TYPE_ID/.lastPinned/ARTIFACT_PATH
```

To download artifacts by build id, use:

```
/repository/download/BUILD_TYPE_ID/BUILD_ID:id/ARTIFACT_PATH
```

To download artifacts by build number, use:

```
/repository/download/BUILD_TYPE_ID/BUILD_NUMBER/ARTIFACT_PATH
```

To download artifacts from last build with specific tag, use:

```
/repository/download/BUILD_TYPE_ID/BUILD_TAG.tcbuildtag/ARTIFACT_PATH
```

To download all artifacts in a .zip archive, use:

```
/repository/downloadAll/BUILD_TYPE_ID/BUILD_SPECIFICATION
```

where

- **BUILD_TYPE_ID** is a build configuration ID.
- **BUILD_SPECIFICATION** can be `.lastFinished`, `.lastSuccessful` or `.lastPinned`, specific `buildNumber` or build id in format `BUILD_ID:id`.
- **ARTIFACT_PATH** is a path to artifact on TeamCity server. This path may contain a **{build.number}** pattern which will be replaced with build number of the build whose artifact is retrieved.
  By default, archive with all artifacts does not include hidden artifact. To include them, add "`?showAll=true`" to the end of the corresponding URL.

Obtaining Artifacts from an Archive

TeamCity allows to obtain a file from an archive from the build artifacts directory by means of the following URL patterns:

```
/repository/download/BUILD_TYPE_ID/BUILD_SPECIFICATION/<archive>!PATH_WITHIN_ARCHIVE
```

- **BUILD_TYPE_ID** is a build configuration ID.
- **BUILD_SPECIFICATION** can be `.lastFinished`, `.lastPinned`, `.lastSuccessful`, specific `buildNumber` or build id in format `BUILD_ID:id`.
- **PATH_WITHIN_ARCHIVE** is a path to a file within an zip/jar/tar.gz archive on TeamCity server.

Following archive types are supported (case insensitive):

- .zip
- .jar
- .war
- .ear
- .nupkg
- .sit
- .apk
- .tar.gz
- .tgz
- .tar.gzip
- .tar

Obtaining Artifacts from a Build Script

It is often required to download artifacts of some build configuration by tools like **wget** or another downloader which does not support HTML login page. TeamCity asks for authentication if you accessing artifacts repository.

To authenticate correctly from a build script, you have to change URLs (add `/httpAuth/` prefix to the URL):

```
/httpAuth/repository/download/BUILD_TYPE_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.

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_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_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_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/bt222/.lastFinished/TeamCity-{build.number}.exe
```

See also:

Concepts: Build Artifact | Authentication Scheme
Administrator's Guide: Retrieving artifacts in builds
Extending TeamCity: Accessing Server by HTTP

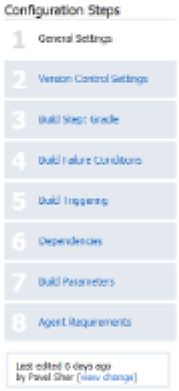
Tracking User Actions

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 needed information faster, filter the log by the activity type, build configurations and/or particular users.

If project or build configuration settings were modified, you can not only see the name of user who made the modification, but also view the change itself by clicking corresponding link.

Since project and build configuration settings are stored on disk in plain xml, the link will just open the usual TeamCity diff window showing changes in these xml files.

You can also see the latest modifications made to a project or build configuration right from project/build configuration settings page.



Last edited 6 days ago
by Pavel Sher (view change)

Audit log also can be retrieved in a text form, see {{logs\teamcity-activities.log }} log file.

Mono Support

Mono framework is an alternative framework for running .NET applications on both Windows and Unix-based platforms.
For more information please refer to the Mono official site.

TeamCity supports running .NET builds using MSBuild and NAnt build runners under Mono framework as well as under .NET Frameworks.

Tests reporting tasks are also supported under Mono.

Mono Platform Detection

When a build agent starts it detects Mono installation automatically.

On each platform Mono detection is compatible with NAnt one. See `NAnt.exe.config` for frameworks detection on NAnt.

Agent Properties

When Mono is detected automatically on agent-side, the following properties are set:

- **Mono** — path to **mono** executable (Mono JIT)
- **MonoVersion** — Mono version
- **MonoX.Z** — set to `MONO_ROOT/lib/mono/X.Z` if exists
- **MonoX.Z_x64** — set to `MONO_ROOT/lib/mono/X.Z` if exists and Mono architecture is x64
- **MonoX.Z_x86** — set to `MONO_ROOT/lib/mono/X.Z` if exists and Mono architecture is x86

If the Mono installation cannot be detected automatically (for example, you have installed Mono framework into custom directory), you can make these properties available for build runners by setting them manually in the agent configuration file.

Windows Specifics

Automatic detection of Mono framework under Windows has the following specifics:

1. Mono version is read from `HKLM\SOFTWARE\Novell\Mono\DefaultCLR`
2. Frameworks paths are extracted from `HKLM\SOFTWARE\Novell\Mono\ %MonoVersion%`
3. Platform architecture is detected by analyzing `mono.exe`

Mac OS X Specifics

1. Framework is detected automatically from /Library/Frameworks/Mono.framework/Versions
2. The highest version is selected
3. Frameworks path are extracted from /Library/Frameworks/Mono.framework/Versions/%MonoVersion%/lib/mono
4. Platform architecture is fixed to x86 as Mono official builds support only X86

Custom Linux/Unix Specifics

Automatic detection of Mono framework under Unix has the following specifics:

1. Mono version is read from "`pkg-config --modversion mono`"
2. Frameworks paths are extracted from "`pkg-config --variable=prefix mono`" and "`pkg-config --variable=libdir mono`"
3. Platform arch is detected by analyzing `*PREFIX*/bin/mono` executable.

You can force Mono to be detected from custom location by adding `*PREFIX*/bin` directory to the beginning of the `PATH` and updating `PKG_CONFIG_PATH` (described in **pkg-config(1)**) with `*PREFIX*/lib/pkgconfig`

Supported Build Runners

Both **NAnt** and **MSBuild** runners support using Mono framework to run a build (MSBuild as xbuild in mono).

See also:

Administrator's Guide: NAnt | MSBuild

Maven Support

Maven Settings Resolution on the Server Side

There are global-level and user-level Maven settings stored in separate XML files.

For the *global-level settings* in the following location:
`${env.M2_HOME}/conf/settings.xml` (or `${system.maven.home}/conf/settings.xml`)

For *user-level settings* TeamCity searches the following locations (listed in order of priority):

1. `~/.BuildServer/system/pluginData/maven/settings.xml`
2. `~/.m2/settings.xml`

where ~ denotes the home directory of the user under which the TeamCity server is running and valid for both MS Windows and Unix-based systems.

For understanding the logic of Maven settings see the Maven product documentation.

See also:

Administrator's Guide: Maven | Maven Artifact Dependency Trigger | Creating Maven Build Configuration

Integrating TeamCity with Other Tools

In this section:

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&lt;a target=&quot;_blank&quot; title=&quot;Click to open this issue a new
window&quot; href=&quot;
       http://www.jetbrains.net/jira/browse/$2&quot;&gt;$2&lt;/a&gt;$3"
       description="JetBrains Jira issue link" />
   </comment-transformation>
...
</server>
```

Search & replace patterns have java.util.regex.Pattern syntax.

External Changes Viewer

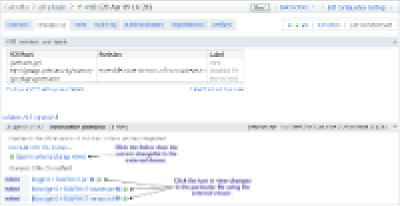
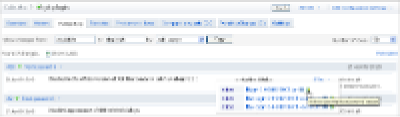
TeamCity supports integration with external changes viewers like Atlassian Fisheye.

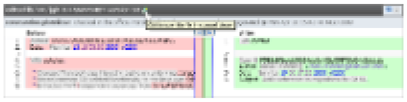
To enable external viewer for changes, you should create and configure the <`TeamCity Data Directory`>`/config/change-viewers.properties` file.

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 tab of the build results page:

- Changes popups (on **Projects** and project home page, **Overview** tab and the **Change Log** tab of the build configuration home page):

- TeamCity file diff page:



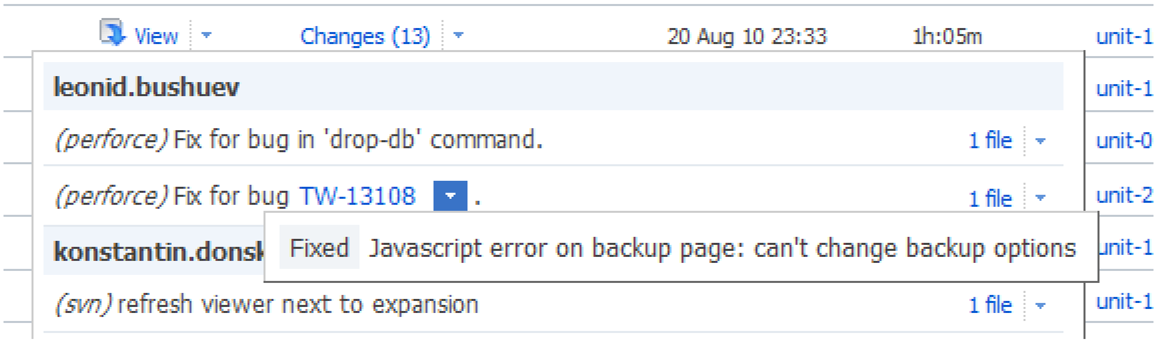
Integrating TeamCity with Issue Tracker

In this section:

- Dedicated Support for Issue Trackers
- Recommendations on Using Issue Tracker Integration
- Enabling Issue Tracker Integration
- Integrating TeamCity with Other Issue Trackers

Dedicated Support for Issue Trackers

TeamCity supports Jira, Bugzilla and YouTrack issue trackers out of the box. Refer to the Supported Platforms and Environments page for the list of supported versions.
When integration is configured, TeamCity automatically transforms an issue ID (=issue key in JIRA) mentioned in VCS comment into a link to the corresponding issue and allows to see basic issue details on hover over an icon displayed near the issue ID.



20 Aug 10 23:33 1h:05m unit-1

leonid.bushuev

(perforce) Fix for bug in 'drop-db' command. 1 file unit-1

(perforce) Fix for bug TW-13108 . 1 file unit-0

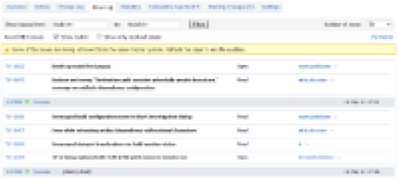
konstantin.donsk Fixed Javascript error on backup page: can't change backup options unit-2

(svn) refresh viewer next to expansion 1 file unit-1

unit-1

Issues fixed in the build can also be viewed on the Issues tab of the build results.

On the build configuration home page, you can review all the issues mapped to the comments at the **Issue Log** tab. You can filter the list to particular range of builds and view which issues where mentioned in comments, and their state.



Recommendations on Using Issue Tracker Integration

To get best results out of issue tracker integration we recommend to:

- When committing changes to your version control **always mention issue id (issue key)** related to the fix in the comment to commit.
- Resolve issues when they are fixed (time of resolve does not really matter).
- Use **Issue Log** of a build configuration to get issues related to builds; turn "Show only resolved issues" option on to show only issues fixed in the builds.

Enabling Issue Tracker Integration

To enable integration, you need to create connection to your issue tracker on the **Administration** | **Issue Tracker** page.
The described below settings are common for all issue trackers:

Connection type	Select type of your issue tracker from the list.
Display name	Symbolic name that will be displayed in TeamCity UI for the issue tracker.
Server URL	Issue tracker URL

Username/Password	Username/password to log in to the issue tracker, if it requires authorizing.

In addition to these general settings you also need to specify which strings should be recognized by TeamCity and converted to links to issues in your issue tracker. For details, please refer to corresponding section:

- YouTrack
- JIRA
- Bugzilla

> ⚠ **Requirements**:
> Information about the issues is retrieved by the TeamCity server using the credentials provided and then is displayed to TeamCity users.
> This has several implications:
>
> - TeamCity server should have direct access to the issue tracker. (Also, TeamCity does not yet support proxy for connections to issue trackers).
> - The user configured in the connection to the issue tracker should have enough permissions to view the issues that can be mentioned in TeamCity. Also, TeamCity users will be able to view the issue details in TeamCity for all the issues that the configured user has access to.

Integrating TeamCity with Other Issue Trackers

To integrate TeamCity with an issue tracker other than Jira/YouTrack/Bugzilla, you can configure TeamCity to turn any issue tracker issue ID references in change comments into links. See mapping external links in comments for details.

Dedicated support for an issue tracker can also be added via a custom issue tracker integration plugin.

See also:

> **Concepts**: Supported Issue Trackers
> **Administrator's Guide**: Mapping External Links in Comments
> **Extending TeamCity**: Issue Tracker Integration Plugin

Bugzilla

Converting Strings into Links to Issues

When enabling issue tracker integration in addition to general settings, you need to specify which strings should be recognized as references to issues in your tracker.
For Bugzilla, you need to specify **Issue Id Pattern**: a Java Regular Expression pattern to find issue ID in the text. The matched text (or the first group if there are groups defined) is used as issue number. Most common case seems to be `#(\d+)` - this will extract 1234 as issue ID from text "Fix for #1234".

Requirements

If username and password are specified, you should have Bugzilla XML-RPC interface switched on. This is not required if you use anonymous access to Bugzilla without username and password.

Known Issues

There are several known issues in Bugzilla regarding XMLs generated for the issues, which makes it hard to communicate to. However it usually can be fixed by tweaking Bugzilla configuration.

- If you see a *path/to/bugzilla.dtd not found* error, this means that the issue XML contains the relative path to the `bugzilla.dtd` file, instead of URL. To fix that you need to set server URL in Bugzilla.
- Sometimes you may see a `SAXParseException` saying that *Open quote is expected for attribute “type_id” associated with an element type “flag”*. This happens because the generated XML does not correspond to the bundled `bugzilla.dtd`, to fix it you need to make the `type_id` attribute `#IMPLIED` (optional) in the bugzilla.dtd file. The issue and the workaround described in detail here.

See also:

Concepts: Supported Issue Trackers
Administrator's Guide: Integrating TeamCity with Issue Tracker

JIRA

Converting Strings into Links to Issues

When enabling issue tracker integration in addition to general settings, you need to specify which strings should be recognized as references to issues in your tracker.
For JIRA, you need to specify space-separated list of **Project keys**.

For example, if a project key is **WEB**, then when an issue key like **WEB-101** is mentioned in a VCS comment, it'll be resolved to a link to corresponding issue.

Requirements

For the Jira integration to work, you should have Jira XML-RPC interface switched on (see instructions in Jira documentation).

See also:

Concepts: Supported Issue Trackers
Administrator's Guide: Integrating TeamCity with Issue Tracker

YouTrack

Converting Strings into Links to Issues

When enabling issue tracker integration in addition to general settings, you need to specify which strings should be recognized as references to issues in your tracker.
For YouTrack, you need to specify space-separated list of **Project Ids**. For example, if a project id is **TW**, then when an issue id like **TW-18802** is mentioned in a VCS comment, it'll be resolved to a link to corresponding issue.

Enhancing Integration with YouTrack

YouTrack provides native TeamCity integration which enhances the set of available features. For example:

- YouTrack is able to fill "Fixed in build" field with a specific build number.
- YouTrack allows you to apply commands to issues by specifying them in a comment to a VCS change commit.

However, to be able to use these features you need to configure YouTrack.

See also:

Concepts: Supported Issue Trackers
Administrator's Guide: Integrating TeamCity with Issue Tracker

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

TeamCity integration provides the following features:

- Remote Run and Pre-Tested (Delayed) Commit,
- customizing parameters for personal builds,
- 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-rununing failed tests from IntelliJ IDEA plugin using JUnit or TestNG,
- opening the patch from the change details web page (for this feature to work you need to have IDEA X installed).

Installing TeamCity plugin

TeamCity IDE plugin version should correspond to the version of the TeamCity server it connects to. Connections to TeamCity servers with different versions are generally not supported.

Installing the Plugin from the Plugin Repository

The plugin repository has a TeamCity plugin from one of the recently released versions. You can install the plugin from repository (e.g. from IntelliJ IDEA Settings > Plugins), then enter the address of your local TeamCity server and let the plugin update itself to the version corresponding to the server.

To install the TeamCity plugin for IntelliJ platform IDE:

1. In IDE, open the **Settings** dialog. To do so either press **Ctrl+Alt+S** or choose **File** > **Settings...** from the main menu.
2. Open **Plugins** section under **IDE Settings** to invoke the **Plugins** dialog.
3. On the **Plugins** dialog, open the **Available** tab to view a list of available plugins.
4. Select **JetBrains TeamCity Plugin**, click the button, and click **OK**.
5. Restart the IDE.
6. Log in into TeamCity server from the plugin
7. Invoke **Update** command tin **TeamCity** menu to install the plugin version matching the server version.

Installing the Plugin Manually

The plugin for IntelliJ platform can be downloaded from the **TeamCity Tool** area on the **My Settings & Tools** page of TeamCity web UI.

To install the TeamCity plugin:

1. In the top right corner of the TeamCity web UI, click the arrow next to your username, and select **My Settings&Tools**.
2. Under the IntelliJ Platform Plugin section in the **TeamCity Tools** area, click the **download** link, and save the archive.
3. Make sure that IDE is not running and unzip the archive into the IDE user plugins directory.

Plugins directory for IntelliJ IDEA is located in:

- Windows: `C:\Documents and Settings\<username>\.IntelliJIdea<vers.>\config\plugins`
- OS X: `$HOME/Library/Application Support/IntelliJIDEA<vers.>`
- Linux/Unix: `$HOME/.IntelliJIdea<vers.>/config/plugins`

Plugins directory for RubyMine is located in:

- Windows: `C:\Documents and Settings\<username>\.RubyMine<vers.>\config\plugins`
- OS X: `$HOME/Library/Application Support/RubyMine<vers.>`
- Linux/Unix: `$HOME/.RubyMine<vers.>/config/plugins`

All additional information on how to work with TeamCity plugin is available in **IDE Help System**.

Configuring IntelliJ IDEA-platform based IDE to Check for Plugin Updates

1. In IntelliJ IDEA , open Settings/ Updates
2. Add "http://<your_teamcity_server_URL>/update/idea-plugins.xml" to the list
3. Set "Check for updates" to "Daily"
4. Press "Apply", then "Check Now"

Eclipse Plugin

Plugin Features

TeamCity integration with Eclipse provides the following features:

- Remote Run and Pre-Tested (Delayed) Commit for Subversion, Perforce, CVS and Git.
- customizing parameters for personal builds,
- monitoring the projects status in the IDE,
- exploring changes introduced in the source code and comparing the local version with the latest version in the project repository,
- navigating from build logs opened in Eclipse to files referenced in build log,
- viewing failed tests of a particular build,
- navigating to TeamCity web interface,
- starting investigation of a build failure,
- viewing server-provided code coverage results run on TeamCity using IDEA or EMMA code coverage engine: "<Main Menu>/TeamCity/Code Coverage Data...",
- comparing personal patch content with workspace resources,
- viewing compilation errors,
- downloading patch to IDE from the TeamCity server,
- shelving changes,
- re-runing tests failed on the TeamCity agent locally,
- support for P4Eclipse up to 2012.1 and Eclipse EGit 0.9 - 0.12

Installing the Plugin

TeamCity Eclipse 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.

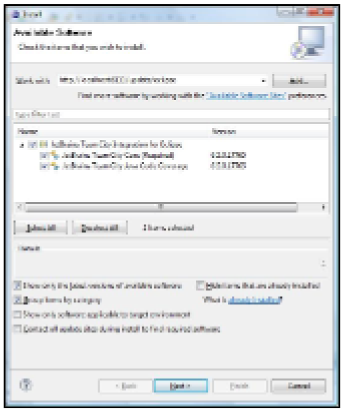
Prerequisites

- **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, 1.8.x update site, 1.6.x update site, 1.4.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 **Eclipse plugin** header, copy the **update site link** URL. For example, in Internet Explorer you can right-click the link and choose **Copy shortcut** on the context menu.
4. In Eclipse, choose **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 required features of the TeamCity Eclipse Plugin.


Install
Available Software
Check the items that you wish to install.
Work with:
Add...
Name
Version
JetBrains TeamCity Integration for Eclipse
JetBrains TeamCity Core (Required)
JetBrains TeamCity Java Code Coverage
Select All
Deselect All
Details
< Back
Next >
Finish
Cancel

1. Click **Next** button and follow the installation instructions.

For detailed instructions on how to work with the plugin, refer to the TeamCity section of Eclipse's help system.

Visual Studio Addin

The TeamCity add-in for Microsoft Visual Studio provides the following features:

- Remote Run for TFS, Subversion and Perforce (for remote run for Mercurial and Git see Branch Remote Run Trigger),
- Pre-Tested (Delayed) Commit for TFS, Subversion and Perforce,
- fetching JetBrains dotCover coverage analysis data from TeamCity server to MS Visual Studio (requires dotCover of supported version installed in Visual Studio),
- viewing recently committed changes and personal builds with their build status in MyChanges tool window,
- opening build failure details in MS Visual Studio from the TeamCity web UI,
- viewing failed tests' details for a build,
- re-runing tests failed in the TeamCity build locally via ReSharper test runner (requires ReSharper 5.0 installed),
- navigation from IDE to build results web page,
- re-applying changes sent in Remote Run or Pre-tested commit to the working directory.

For detailed instructions on how to use the add-in, refer to the TeamCity Help section, embedded in Microsoft Visual Studio's Help System.

> To enable navigation to the failed tests in MS Visual Studio by using "open in IDE" actions in the web UI, make sure that .pdb file generation for the assemblies involved in NUnit/MSTest unit tests is switched on in current Visual Studio project.

Installing the Add-in

TeamCity VS add-in version should correspond to the version of the TeamCity server it connects to. Connections to TeamCity servers with different versions are generally not supported.

Download the add-in available in the **TeamCity Tools** section of the **My Settings&Tools** page (to open the page click the arrow next to your username in the top right corner of the TeamCity web UI and select **My Settings&Tools**) and follow the installation wizard.
Please, close all running instances of Visual Studio before starting add-in installation (initial or upgrade).

Requirements

Please see Supported Platforms and Environments page for the system requirements for integrations with different version control systems or coverage tools.

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.

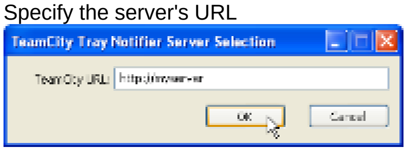
See also:

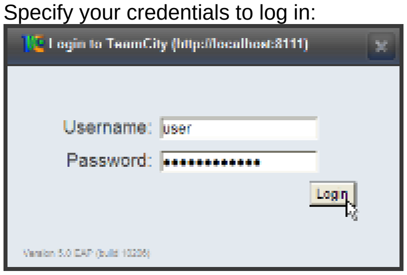
> **User's Guide**: Subscribing to Notifications
> **Administrator's Guide**: Customizing Notifications
> **Installing Tools**: Working with Windows Tray Notifier

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

TeamCity Tray Notifier Server Selection
TeamCity URL: http://myserver

2. Specify your credentials to log in:

Login to TeamCity (http://localhost:8111)
Username: user
Password: •••••••••••••

Version 5.0 EAP (build 10226)

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.

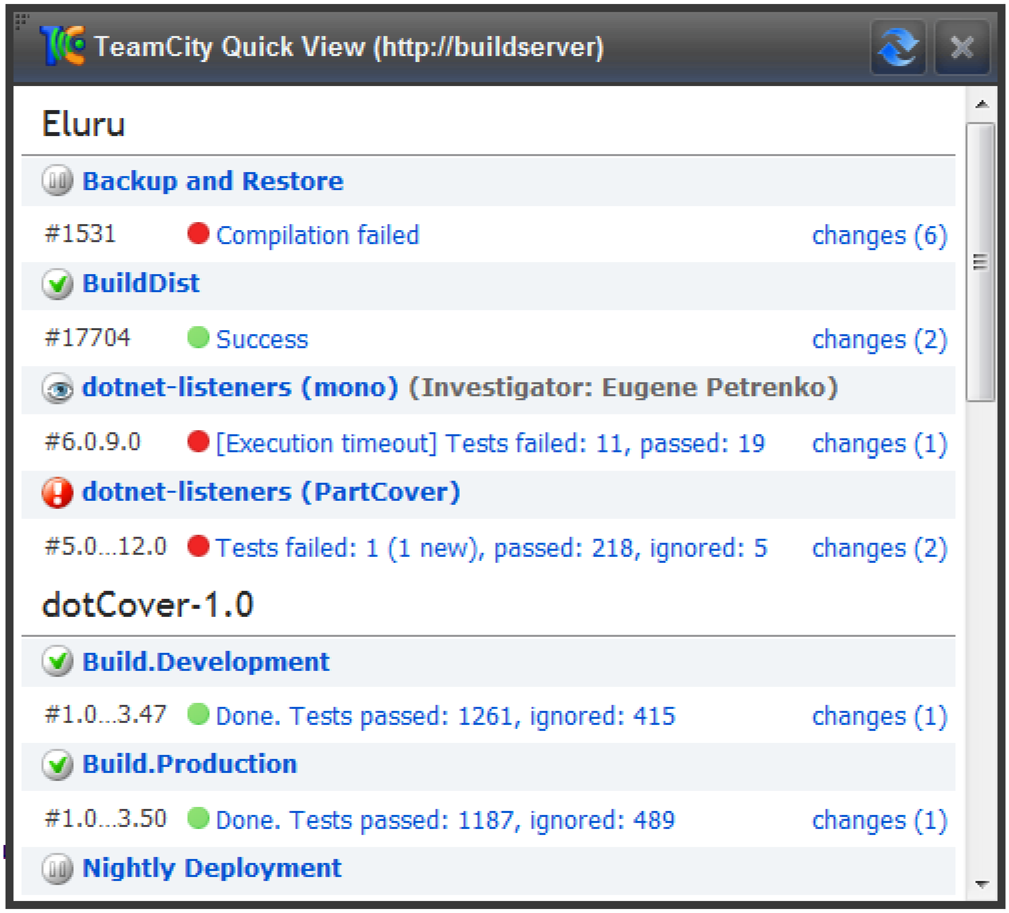
Icon	Meaning
✓	Build is successful
!	Build failed and nobody is investigating the failure
👁	Some team member has started investigation of the build failure

✓	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:



TeamCity Quick View (http://buildserver)

Eluru

⏸ Backup and Restore

#1531 ● Compilation failed changes (6)

✓ BuildDist

#17704 ● Success changes (2)

dotnet-listeners (mono) (Investigator: Eugene Petrenko)

#6.0.9.0 ● [Execution timeout] Tests failed: 11, passed: 19 changes (1)

! dotnet-listeners (PartCover)

#5.0...12.0 ● Tests failed: 1 (1 new), passed: 218, ignored: 5 changes (2)

dotCover-1.0

✓ Build.Development

#1.0...3.47 ● Done. Tests passed: 1261, ignored: 415 changes (1)

✓ Build.Production

#1.0...3.50 ● Done. Tests passed: 1187, ignored: 489 changes (1)

⏸ Nightly Deployment

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.

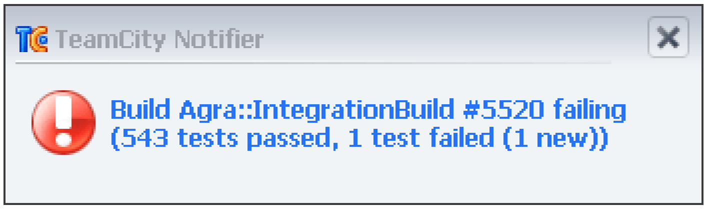
If you right-click the icon, you can access all Windows Tray Notifier features described in table below.

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.



Build Agra::IntegrationBuild #5520 failing (543 tests passed, 1 test failed (1 new))

When a pop-up notification appears, you can click the link in it to go the **Build results** page for more details and investigate the desired issues.

See also:

User's Guide: Subscribing to Notifications
Administrator's Guide: Customizing Notifications
How To: Watching Several TeamCity Servers With Tray Notifier

Syndication Feed

To configure a syndication feed for obtaining information about the builds of certain build configurations:

1. In the top right corner of the TeamCity web UI, click the arrow next to your username, and select **My Settings&Tools**.
2. In the **TeamCity Tools** section, click the **Customize** link.
3. In the Feed URL Generator page specify the build configurations and events (builds and/or changes) you want to be notified about, and define authentication settings.
4. Copy the Feed URL, generated by TeamCity, to your feed reader, or click **Subscribe**.
5. Preview summary of the subscription and click **Subscribe now**.

See also:

User's Guide: Feed URL Generator

Extending TeamCity

TeamCity behavior can be extended in several ways. You can communicate with TeamCity from the build script and report tests, change build number or provide statistics data. Or you can write full-fledged plugin which will provide custom UI, customized notifications and much more.

If you cannot find relevant information here, have questions or want to share your TeamCity plugins experience with other users, welcome to TeamCity Plugins Forum.

Customizing TeamCity without Plugins

- Build Script Interaction with TeamCity
- Accessing Server by HTTP
- Risk Tests Reordering in Custom Test Runner
- Including Third-Party Reports in the Build Results

Plugin Development

- Plugin Types in TeamCity
- Bundled Development Package
- Open API Changes
- Plugins Packaging
- Server-side Object Model
- Agent-side Object Model
- Extensions
- Web UI Extensions
- Plugin Settings
- Development Environment
- Typical Plugins
  - Build Runner Plugin
  - 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
  - Extending Highlighting for Web diff view

Open API Javadoc
Open API Javadoc (ver. 5.0.x)
Open API Javadoc (ver. 4.5.x)

Publicly Available Plugins

- TeamCity Plugins
- Open-source Bundled Plugins

Build Script Interaction with TeamCity

If TeamCity doesn't support your testing framework or build runner out of the box, you can still avail yourself of many TeamCity benefits by customizing your build scripts to interact with the TeamCity server. This makes a wide range of features available to any team regardless of their testing frameworks and runners. Some of these features include displaying real-time test results and customized statistics, changing the build status, and publishing artifacts before the build is finished.
The build script interaction can be implemented by means of:

- service messages in the build script
- `teamcity-info.xml` file

> If you use MSBuild build runner, you can use MSBuild Service Tasks.

In this section:

- Service Messages
  - Common Properties
    - Message Creation Timestamp
    - Message FlowId
  - Blocks of Service Messages
  - Reporting Messages For Build Log
  - Reporting Compilation Messages
  - Reporting Tests
  - Reporting .NET Code Coverage Results
  - Publishing Artifacts while the Build is Still in Progress
  - Reporting Build Progress
  - Reporting Build Status
  - Reporting Build Number
  - Adding or Changing a Build Parameter
  - Reporting Build Statistics
  - Disabling Service Messages Processing
  - Importing XML Reports
- teamcity-info.xml
  - Modifying the Build Status
  - Reporting Custom Statistics
    - Providing data using the teamcity-info.xml file

- Describing custom charts

Service Messages

Service messages are used to pass commands/build information to TeamCity server from the build script. In order to be processed by TeamCity they should be printed into standard output stream of the build (otherwise, if the output is not in the service message syntax, it should appear in the build log). A single service message should not contain a newline character inside it, it should not span across multiple lines.

Service messages support two formats:

- Single attribute message:

  ```
  ##teamcity[message 'value']
  ```

- Multiple attribute message:

  ```
  ##teamcity[message 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 a 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 *WSPpropertyNameOWSP=OWSP'_value*'_ blocks

For escaped values, TeamCity uses a vertical bar "|" as an escape character. In order to have certain characters properly interpreted by the TeamCity server they must be preceded by a vertical bar.
For example, the following message:

```
##teamcity[testStarted name='foo|'s test']
```

will be displayed in TeamCity as 'foo's test'. Please, refer to the table of the escaped values below.

Character	Should be escaped as
' (apostrophe)	|'
\n (line feed)	|n
\r (carriage return)	|r
\uNNNN (unicode symbol with code 0xNNNN)	|0xNNNN
| (vertical bar)	||
[(opening bracket)	|[
] (closing bracket)	|]

Common Properties

Any "message and multiple attribute" message supports the following list of optional attributes: `timestamp`, `flowId`.
In the following examples `<messageName>` is the name of the specific service message.

Message Creation Timestamp

```
##teamcity[<messageName> timestamp='timestamp' ...]
```

Timestamp format is "yyyy-MM-dd'T'HH:mm:ss.SSSZ" or "yyyy-MM-dd'T'HH:mm:ss.SSS" (or "yyyy-MM-dd'T'HH:mm:ss.fffzzz" for .NET DateTime), according to Java SimpleDateFormat syntax e.g.

```
##teamcity[<messageName> timestamp='2008-09-03T14:02:34.487+0400' ...]
##teamcity[<messageName> timestamp='2008-09-03T14:02:34.487' ...]
```

Message FlowId

The flowId is a unique identifier of the messages flow in a build. Flow tracking is necessary for example to distinguish separate processes running in parallel. The identifier is a string that should be unique in the scope of individual build.

```
##teamcity[<messageName> flowId='flowId' ...]
```

Blocks of Service Messages

Blocks are used to group several messages in the build log.

Block opening:

```
 ##teamcity[blockOpened name='<blockName>']
```

Block closing:

```
 ##teamcity[blockClosed name='<blockName>']
```

> ⚠ Please note that when you close the block all inner blocks become closed automatically.

Reporting Messages For Build Log

You can report messages for build log in the following way:

```
##teamcity[message text='<message text>' errorDetails='<error details>' status='<status value>']
```

where:

- The `status` attribute may take following values: `NORMAL`, `WARNING`, `FAILURE`, `ERROR`. The default value is `NORMAL`.
- The `errorDetails` attribute is used only if `status` is `ERROR`, in other cases it is ignored.

This message fails the build in case its status is `ERROR` and "Fail build if an error message is logged by build runner" checkbox is checked on build configuration "Build Failure Conditions" page. For example:

```
##teamcity[message text='Exception text' errorDetails='stack trace' status='ERROR']
```

Reporting Compilation Messages

```
##teamcity[compilationStarted compiler='<compiler name>']
...
##teamcity[message text='compiler output']
##teamcity[message text='compiler output']
##teamcity[message text='compiler error' status='ERROR']
...
##teamcity[compilationFinished compiler='<compiler name>']
```

where:

- `compiler name` is an arbitrary name of the compiler performing compilation, eg, javac, groovyc and so on. Currently it is used as a block name in the build log.
- any message with status `ERROR` reported between `compilationStarted` and `compilationFinished` will be treated as compilation error.

Reporting Tests

To use TeamCity's on-the-fly test reporting, testing framework needs dedicated support for this feature to work (alternatively, XML Report Processing can be used).
If TeamCity doesn't support your testing framework natively, it is possible to modify your build script to report test runs to the TeamCity server using service messages. This makes it possible to display test results in real-time, make test information available on the **Tests** tab of the **Build Results** page.

Here is the list of supported test service messages:

Test suite messages:
Test suites are used to group tests. TeamCity display tests grouped by suite on the Tests tab of the build results and in other places.

```
##teamcity[testSuiteStarted name='suite.name']
<individual test messages go here>
##teamcity[testSuiteFinished name='suite.name']
```

All the individual test messages should appear between `testSuiteStarted` and `testSuiteFinished` (in that order) with the same `name` attributes.
Suites may also be nested.

Test start/stop messages:

```
##teamcity[testStarted name='testname' captureStandardOutput='<true/false>']
<here go all the test service messages with the same name>
##teamcity[testFinished name='testname' duration='<test_duration_in_milliseconds>']
```

Indicates that the test "*testname*" was run. If `testFailed` message is not present, the test is regarded as successful.

duration (optional numeric attribute) - sets the test duration to be reported in TeamCity UI. If omitted, the test duration will be calculated from the messages timestamps. If the timestamps are missing, from the actual time the messages were received on the server.
captureStandardOutput (optional boolean attribute) - if `true`, all the standard output (and standard error) messages received between `testStarted` and `testFinished` messages will be considered test output. The default value is `false` and assumes usage of `testStdOut` and `testStdErr` service messages to report the test output.

⚠

- All the other test messages (except for `testIgnored`) with the same `name` attribute should appear between the `testStarted` and `testFinished` messages (in that order).
- Currently, the test-related service messages cannot be output with Ant's *echo* task until *flowId* attribute is specified.

It is highly recommended to ensure that the pair of test suite + test name is unique within the build.
For advanced TeamCity test-related features to work, test names should not deviate from one build to another (a single test should be reported under the same name in every build). e.g. it's highly unrecommended to include absolute paths in the reported test names.

Ignored tests:

```
##teamcity[testIgnored name='testname' message='ignore comment']
```

Indicates that the test "`testname`" is present but was not run (was ignored) by the testing framework.
As an exception `testIgnored` message can be reported without matching `testStarted` and `testFinished` messages.

Test output:

```
##teamcity[testStarted name='testname']
##teamcity[testStdOut name='testname' out='text']
##teamcity[testStdErr name='testname' out='error text']
##teamcity[testFinished name='testname' duration='50']
```

`testStdOut` and `testStdOrr` service messages report the test's standard and error output to be displayed in TeamCity UI. There should be no more then single `testStdOut` and single `testStdErr` message per test.
Alternative, but less reliable approach is to use `captureStandardOutput` attribute of `testStarted` message.

Test result:

```
##teamcity[testStarted name='test1']
##teamcity[testFailed name='test1' message='failure message' details='message and stack trace']
##teamcity[testFinished name='test1']

##teamcity[testStarted name='test2']
##teamcity[testFailed type='comparisonFailure' name='test2' message='failure message' details='message
and stack trace' expected='expected value' actual='actual value']
##teamcity[testFinished name='test2']
```

Indicates that the test with the name "*testname*" has failed. Only single `testFailed` message should appear for a given test name.
`message` contains a textual representation of the error.
`details` contains detailed information on the test failure, typically a message and an exception stacktrace.
`actual` and `expected` attributes should only be used together with `type='comparisonFailure` and can be used for reporting comparison failure. The values will be used when opening the test in the IDE.

Here is a longer example of test reporting with service messages:

```
##teamcity[testSuiteStarted name='suite.name']
##teamcity[testSuiteStarted name='nested.suite']
##teamcity[testStarted name='package_or_namespace.ClassName.TestName']
##teamcity[testFailed name='package_or_namespace.ClassName.TestName' message='The number should be
20000' details='junit.framework.AssertionFailedError: expected:<20000> but was:<10000>|n|r    at
junit.framework.Assert.fail(Assert.java:47)|n|r    at
junit.framework.Assert.failNotEquals(Assert.java:280)|n|r...']
##teamcity[testFinished name='package_or_namespace.ClassName.TestName']
##teamcity[testSuiteFinished name='nested.suite']
##teamcity[testSuiteFinished name='suite.name']
```

Reporting .NET Code Coverage Results

You can configure .NET coverage processing by means of service messages. To learn more, refer to Manually Configuring Reporting Coverage page.

Publishing Artifacts while the Build is Still in Progress

You can publish build artifacts, while the build is still running, right after the artifacts are built.
For this you need to output the following line:

```
##teamcity[publishArtifacts '<path>']
```

And the files matching the `<path>` will be uploaded and visible as artifacts of the running build. `<path>` should adhere to the same rules as Build Artifact specification on the Build Configuration settings.
The message should be printed after all the files are ready and no file is locked for reading. Artifacts uploading happens in background and can take time. Please make sure the matching files are not deleted till the end of the build. (e.g. you might put them in a directory that is cleaned on next build start, temp directory or use Swabra to clean them after the build.)

> Publishing artifacts process can affect the build because it consumes network traffic and some disk/CPU resources (should be pretty negligible for not large files/directories).

Artifacts that are specified in the build configuration setting will be published as usual.

Reporting Build Progress

You can use special progress messages to mark long-running parts in a build script. These messages will be shown on the projects dashboard for corresponding build and on the build results page.

To log single progress message use:

```
##teamcity[progressMessage '<message>']
```

This progress message will be shown until another progress message occurs or until next target starts (in case of Ant builds).

If you wish to show progress message for a part of a build only you can use:

```
##teamcity[progressStart '<message>']
...some build activity...
##teamcity[progressFinish '<message>']
```

> The same message should be used for both `progressStart` and `progressFinish`. This allows nesting of progress blocks. Also note that in case of Ant builds progress messages will be replaced if Ant target starts.

Reporting Build Status

TeamCity allows user to change the **build status** directly from the build script.

You can also permanently change the **build status text** for your build. Unlike with progress messages, this change persists even after build has finished.

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:

- The `status` attribute may take following values: `FAILURE`, `SUCCESS`.
- `{build.status.text}` is an optional substitution pattern which represents the status, calculated by TeamCity automatically using passed test count, compilation messages and so on.
- neither `status`, nor `text` attributes are mandatory

> The status, which is set using the attribute, will be presented while build is running and will affect final build results.

Reporting Build Number

To set a custom build number directly, specify a `buildNumber` message using the following format:

```
##teamcity[buildNumber '<new build number>']
```

In the <new build number> value, you can use the `{build.number}` substitution to use the current build number automatically generated by TeamCity. For example:

```
##teamcity[buildNumber '1.2.3_{build.number}-ent']
```

Adding or Changing a Build Parameter

By using dedicated service message in your build script, you can dynamically update some build parameters right from a build step, so that following build steps will run with modified set of build parameters.

```
##teamcity[setParameter name='ddd' value='fff']
```

When specifying a build parameter's name, mind the prefix:

- **system** for system properties.
- **env** for environment variables.
- no prefix for configuration parameter.
  Read more about build parameters and their prefixes.

The changed build parameters will also be available in dependent builds as `%dep.*%` properties.

Reporting Build Statistics

In TeamCity, it is possible to configure a build script to report statistical data and then display the charts based on the data. Please refer to the Customizing Statistics Charts#customCharts page for a guide to displaying the charts on the web UI. This section describes how to report the statistical data from the build script via service messages. You can publish the build statics values in two ways:

- Using a service message in a build script directly
- Providing data using the teamcity-info.xml file

To report build statistics using service messages:

- Specify a '`buildStatisticValue`' service message with the following format for each statistics value you want to report:

  ```
  ##teamcity[buildStatisticValue key='<valueTypeKey>' value='<value>']
  ```

The `key` should not be equal to any of predefined keys.
The `value` should be a positive integer value.

Disabling Service Messages Processing

If you need for some reason to disable searching for service messages in output, you can disable service messages search with the messages:

```
##teamcity[enableServiceMessages]
##teamcity[disableServiceMessages]
```

Any messages that appear between these two are not parsed as service messages and are effectively ignored.
For server-side processing of service messages, enable/disable service messages also support flowId attribute and will ignore only the messages with the same flowId.

Importing XML Reports

In addition to UI Build Feature, XML reporting can be configured from within the build script with the help of `importData` service message. Also, the message supports importing of previously collected code coverage and code inspection/duplicates reports.

The service message format is:

```
##teamcity[importData type='typeID' path='<path to the xml file>']
```

where `typeID` can be one of the following (see also XML Report Processing):

`typeID`	Description

Testing frameworks	
`junit`	JUnit Ant task XML reports
`surefire`	Maven Surefire XML reports
`nunit`	NUnit-Console XML reports
`mstest`	MSTest XML reports
`gtest`	Google Test XML reports
Code inspection	
`checkstyle`	Checkstyle inspections XML reports
`findBugs` [2]	FindBugs inspections XML reports
`jslint`	JSLint XML reports
`FxCop` [1]	FxCop inspection XML reports
`pmd`	PMD inspections XML reports
Code duplication	
`pmdCpd`	PMD Copy/Paste Detector (CPD) XML reports
`ReSharperDupFinder` [1]	ReSharper `dupfinder.exe` XML reports
Code coverage	
`dotNetCoverage` [1] [3]	XML reports generated by dotcover, partcover, ncover or ncover3

Notes:
[1] only supports specific file in the "path" attribute
[2] also requires `findBugsHome` attribute specified pointing to the home directory oif installed FindBugs tool.
[3] also requires `tool='<tool name>'` service message attribute, where `<tool name>` is one of: `dotcover`, `partcover`, `ncover` or `ncover3`.

- If not specially noted, the report types support Ant-like wildcards in the path attribute.
- `verbose='true'` attribute will enable detailed logging into the build log.
- `parseOutOfDate='true'` attribute will process all the files matching the path. Otherwise, only those updated during the build (is determined by last modification timestamp) are processed.
- `whenNoDataPublished=<action>` (where <action> is one of the following: `info` (default), `nothing`, `warning`, `error`) will change output level if no reports matching the path specified were found.

(**deprecated, use Build Failure Conditions instead**)
`findBugs`, `pmd` or `checkstyle` importData messages also take optional `errorLimit` and `warningLimit` attributes which specify errors and warnings limits, exceeding which will cause the build failure.

> ⚠
>
> - After `importData` message is received, TeamCity agent starts to monitor specified paths on disk and imports matching report files in the background as soon as the files appear on disk.
> - The parsing only occurs within the build step in which the messages were received. On step finish, the agent ensures all the present reports are processed before beginning of the next step. This behavior is different from that of XML Report Processing build feature, which completes files parsing only at the end of the build.
> - Please ensure the report files are available after the generation process ends (the files are not deleted, nor overwritten by the build scirpt)

To initiate monitoring several directories or parse several types of the report, send the corresponding service messages one after another.

teamcity-info.xml

It is also possible to have the build script collect information, generate an XML file called `teamcity-info.xml` in the root build directory. When the build finishes, this file will automatically be uploaded as a build artifact and processed by the TeamCity server.

Please note that this approach can be discontinued in the future TeamCity versions, so service messages approach is recommended instead. In case service messages does not work for you please let us know the details and describe the case via email.

Modifying the Build Status

TeamCity has the ability to change the build status directly from the build script. You can set the status (build failure or success) and change the text of the build status (for example, note the number of failed tests if the test framework is not supported by TeamCity).

XML schema for teamcity-info.xml

It is possible to set the following information for the build:

- **Build number** — Sets the new number for the finished build. You can reference the TeamCity-provided build number using {`build.number`}.
- **Build status** — Change the build status. Supported values are "FAILURE" and "SUCCESS".
- **Status text** — Modify the text of build status. You can replace the TeamCity-provided status text or add a custom part before or after the standard text. Supported `action` values are "append", "prepend" and "replace".

Example `teamcity-info.xml` file:

```
<build number="1.0.{build.number}">
   <statusInfo status="FAILURE"> <!-- or SUCCESS -->
      <text action="append"> fitnesse: 45</text>
      <text action="append"> coverage: 54%</text>
   </statusInfo>
</build>
```

⚠ It is up to you to figure out how to retrieve test results that are not supported by TeamCity and accurately add them to the `teamcity-info.xml` file.

Reporting Custom Statistics

It is possible to provide custom charts in TeamCity. Your build can provide data for such graphs using `teamcity-info.xml` file.

Providing data using the teamcity-info.xml file

This file should be created by the build in the root directory of the build. You can publish multiple statistics (see the details on the data format below) and create separate charts for each set of values.

The `teamcity-info.xml` file should contain code in the following format (you can combine various data in the `teamcity-info.xml` file):

```
<build>
   <statisticValue key="chart1Key" value="342"/>
   <statisticValue key="chart2Key" value="53"/>
</build>
```

The `key` should not be equal to any of predefined keys.
The `value` should be a positive integer value.

The key here relates to the key of **valueType** tag used when describing the chart.

Describing custom charts

See Customizing Statistics Charts page for detailed description.

Accessing Server by HTTP

In addition to the commands described here, there is a REST API that you can use for certain operations.

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

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

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

To trigger a build, send the **HTTP GET** request for the URL: `http://<server address>/httpAuth/action.html?add2Queue=<build type 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 type Id>
```

Example:

```
http://testuser:testpassword@teamcity.jetbrains.com/httpAuth/action.html?add2Queue=bt10
```

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/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/httpAuth/action.html?add2Queue=bt10&agentId=1&agentId
```

To trigger a build on all enabled and compatible agents, use "allEnabledCompatible" as agent ID:

```
http://testuser:testpassword@teamcity.jetbrains.com/httpAuth/action.html?add2Queue=bt10&agentId=allEnable
```

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/httpAuth/action.html?add2Queue=bt10&modificationId=11
```

`modificationId` — internal TeamCity server 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/httpAuth/action.html?add2Queue=bt10&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.

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 on the build results page.
General flow is like this:

- configure the build script to produce the HTML report (preferably in a zip archive);
- configure build artifacts to publish the report as build artifact to the server: at this point you can check that the archive is available in build's artifacts;
- configure Report Tab in the administration area to make the report from artifacts available as an extra tab on the build or project level.

There are two types of report tabs available:

- **Build-level**: appears on 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 or project settings respectively, 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, which 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.
Start page	Specify relative path to the start page of the generated report. The path should be relative to the root of build's artifacts. To use a file from an archive, use `path-to-archive!relative-path` syntax. See also a the list of supported archives. For example, `javadoc.zip!index.html`

Risk Tests Reordering in Custom Test Runner

In TeamCity, you can instruct the system to run risk group tests before any others.

To implement risk group tests reordering feature for your own custom test runner, TeamCity provides following special properties:

- **teamcity.tests.runRiskGroupTestsFirst** — this property value contains groups of tests to run before others. Accordingly there are two groups: **recentlyFailed** and **newAndModified**. If more than one group specified they are separated with comma. This property is provided only if corresponding settings are selected on build runner page.

- **teamcity.tests.recentlyFailedTests.file** — this property value contains full path to a file with recently failed tests. The property is provided only if **recentlyFailed** group is selected. The file contains tests separated by new line character. For Java like tests full class names are stored in the file (without test method name). In other cases full name of the test will be stored in the file as it was reported by the tests runner.

- **teamcity.build.changedFiles.file** — this property is useful if you want to support running of new and modified tests in your tests runner. This property contains full path to a file with information about changed files included in the build. You can use this file to determine whether any tests were modified and run them before others. The file contains lines separated by new line. Each line corresponds to one file and has the following format:

  ```
  <relative file path>:<change type>:<revision>
  ```

  where:

  - `<relative file path>` is a path to a file relative to the current checkout directory.
  - `<change type>` is a type of modification and can have the following values: `CHANGED`, `ADDED`, `REMOVED`, `NOT_CHANGED`, `DIRECTORY_CHANGED`, `DIRECTORY_ADDED`, `DIRECTORY_REMOVED`
  - `<revision>` is a file revision in repository. If file is a part of change list started via remote run then string `<personal>` will be written instead of file revision.

- **teamcity.build.checkoutDir** — this property contains path to a build checkout directory. It is useful if you need to convert relative paths to modified files to absolute ones.

> ⚠ TeamCity will pass **teamcity.tests.runRiskGroupTestsFirst**, **teamcity.tests.recentlyFailedTests.file** and **teamcity.build.changedFiles.file** properties to the build process, but if process starts additional JVM or other processes, these properties won't be passed to them automatically.
>
> For example, if you are using Ant runner, you will have access to these properties from the Ant build.xml. But if your build.xml starts new JVM (or `<junit/>` task with `fork="yes"` attribute), and you want to access these properties from this JVM, you'll have to modify your build script and pass them explicitly.

Custom Chart

In addition to statistic charts generated automatically by TeamCity, it is possible to configure your own statistical charts based on the set of build metrics provided by TeamCity or values reported from build script. In the latter case you will need to configure your build script to report custom statistical data to TeamCity.

Displaying a custom chart in TeamCity web UI

To make TeamCity display a custom chart in web UI, you need to update dedicated configuration file:

- For Project-level chart: `<TeamCity Data Directory>/config/<Project Name>/plugin-settings.xml`
- For Build Configuration-level chart: `<TeamCity data dir>/config/main-config.xml`

You can edit these files while the server is running, they will be automatically reloaded.

A statistics chart is added using `graph` tag. See the examples below:

Custom project-level charts in `plugin-settings.xml`

```
<settings>
  <custom-graphs> <!-- This tag is required only in plugin-settings.xml -->
   <graph title="Duration comparison" hideFilters="showFailed" seriesTitle="some key"
format="duration">
      <valueType key="BuildDuration" title="duration2" buildTypeId="bt7"/>
      <valueType key="BuildDuration" title="duration1" buildTypeId="bt3"/>
      <valueType key="customKey" title="Custom data" /> <!-- Will use data from build configuration
bt3 -->
    </graph>
  </custom-graphs>
</settings>
```

Custom build configuration-level charts in `main-config.xml`

```
<server ...>
 <!-- Some other stuff -->
 <graph title="Passed Test Count" seriesTitle="Configuration">
   <valueType key="PassedTestCount" title="This configuration" />
   <valueType key="PassedTestCount" title="Passed Test Count" buildTypeId="bt32"/> <!-- This is
explicit reference to build configuration -->
 </graph>
 <graph title="Tests against Coverage">
   <valueType key="PassedTestCount" title="Tests" />
   <valueType key="CodeCoverageL" title="Line coverage" />
 </graph>
 <graph title="Custom data" seriesTitle="Metric name" format="size">
   <valueType key="key1" title="Metric 1" />
   <valueType key="key2" title="Metric 1" />
   <valueType key="BuildDuration" title="Duration" />
 </graph>
</server>
```

Note, that when adding custom charts on the project level intermediate `custom-graphs` tag is required.

Tags reference

<graph> : describes a single chart. It should contain one or more `valueType` subtags, which describe series of data shown in the chart.

Attribute	Description
title	Title above the chart.
seriesTitle	Title above list of series used on the chart (in singular form). Default is "Serie".
defaultFilters	List of comma-separated options, which should be checked by default. Can include the following: • showFailed — include results from failed builds by default. • averaged — by default, show averaged values on the chart.
hideFilters	List of comma-separated filter names that should not be shown next to the chart: • all — hide all filters. • series — hide series filter (you won't be able to show only data from specific valueType specified for the chart. • range — hide date range filter. • showFailed — hide checkbox which allows to include data for failed builds. • averaged — hide checkbox which allows to view averaged values. Defaults — empty (all filters are shown).
format	Format of the y-axis values. Supported formats are: • `duration`, data should be in milliseconds; • `percent`, data should be in percents (from 0 to 100); • `size`, data should be in bytes. If no format is specified, 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 `BuildDuration` or `ArtifactsSize` (see below the complete list of pre-defined build metrics), or you can provide your own data by reporting it from the build script.
title	Series name, shown in the series selector. Defaults to <key>.

buildTypeId	This field allows to explicitly specify build configuration to use data from for given valueType. This field is mandatory for the first valueType used in a chart, if the chart is added at project level. In other cases it is optional. However, note that TeamCity chooses build configuration to take data from according to following rules: 1. if `buildTypeId` is set within `valueType`, data is taken from this build configuration even if it belongs to another project. 2. if `buildTypeId` is not set within current `valueType`, but it is set in `valueType` above current one within the chart, the data from the build configuration referenced above will be taken. See example for `plugin-settings.xml` file above. 3. if `buildTypeId` is not set within current `valueType` and is not set above, the chart will show data for the current build configuration, i.e. this chart will work only for build configurations. Such charts can be configured only in `main-config.xml`.

Build Metrics Provided by TeamCity

The following lists the pre-defined value providers that can be used to configure a custom chart. Using these values doesn't require build script modification.

Value	Description	Unit
ArtifactsSize	Sum of all artifact file sizes in artifact directory.	Bytes
BuildArtifactsPublishingTime	Duration of the artifact publishing step in the build.	Milliseconds
BuildCheckoutTime	Duration of the source checkout step.	Milliseconds
BuildDuration	Build duration, excluding checkout or artifact publishing time.	Milliseconds
CodeCoverageB	Block-level code coverage	%
CodeCoverageC	Class-level code coverage	%
CodeCoverageL	Line-level code coverage	%
CodeCoverageM	Method-level code coverage	%
CodeCoverageAbsLCovered	Number of covered lines	int
CodeCoverageAbsMCovered	Number of covered methods	int
CodeCoverageAbsCCovered	Number of covered classes	int
CodeCoverageAbsLTotal	Total number of lines	int
CodeCoverageAbsMTotal	Total number of methods	int
CodeCoverageAbsCTotal	Total number of classes	int
DuplicatorStats	Number of found code duplicates	int
FailedTestCount	Number of failed tests in the build	int
IgnoredTestCount	Number of ignored tests in the build	int
InspectionStatsE	Number of inspection errors in the build	int
InspectionStatsW	Number of inspection warnings in the build	int
PassedTestCount	Number of successfully passed tests in the build	int
SuccessRate	Indicator whether the build was successful	0 - failed, 1 - successful
TimeSpentInQueue	How much time build was in queue	Milliseconds

Custom Build Metrics

If pre-defined 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 `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

Developing TeamCity Plugins

TeamCity functionality can be significantly extended by a custom plugin. TeamCity plugins are written in Java (Groovy and JRuby can also be used), runs within the TeamCity application and has access to internal entities of the TeamCity server or agent.

Aside from this documentation, please refer to the following sources:

- Open API Javadoc
- bundled sample plugin
- open-source plugins: bundled or third-party

If you cannot find enough information or have a question regarding API please do not hesitate to post your question into TeamCity Plugins forum. Please use search before posting to find out if alike question was already answered in the forums.

Please refer to corresponding section for further details.

- Plugin Types in TeamCity
- Bundled Development Package
- Open API Changes
- Plugins Packaging
- Server-side Object Model
- Agent-side Object Model
- Extensions
- Web UI Extensions
- Plugin Settings
- Development Environment
- Typical Plugins
  - Build Runner Plugin
  - 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
  - Extending Highlighting for Web diff view

Plugin Types in TeamCity

TeamCity mainly consists of two parts:

1. Server which gathers information while builds are running
2. Agents which run builds and send information to server

Plugins can be written for server and for agent. Aside from that plugins are divided by their types:

- Build runners
- VCS plugins
- Notifiers
- User authentication plugins
- Build Triggers
- Extensions

Extensions 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 it or modify build status text, read more.

Plugins can also modify TeamCity web UI. They can provide custom content to the existing pages (again there are several extension points provided for that), or create new pages with their own UI, read more.

Bundled Development Package

TeamCity comes bundled with a Development Package that can be used to start developing TeamCity plugins.

To get the package use `.tar.gz` or `.exe.` distribution.
Upon installation, `<TeamCity Home Directory>` will have a `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 plugin and agent-side plugin.
To develop agent-side plugin you need the following part of 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 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. `runtime` directory of `devPackage` contains some jars that can be used in this case.

`devPackage` also contains some base classes for tests under the `tests` directory.

Sample Plugin

Building and deploying sample plugin

Building plugin with Apache Ant

- Unpack `<TeamCity Home Directory>\devPackage\samplePlugin-src.zip` into a directory of your choice
- Edit `build.properties` file and set value for `path.variable.teamcitydistribution` property to path of `<TeamCity Home Directory>`
- Run `ant dist` in the plugin directory (Ant 1.7+ is recommended). The plugin distribution should be created in `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 path variable, set "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 `build.properties` file to set `path.variable.teamcitydistribution` property or regenerate the build script from IDEA (execute "Generate Ant Build" with settings: single file, all other options unchecked).

If you use Ultimate edition of IntelliJ IDEA, you can start TeamCity's Tomcat right form the IDE:

- Go to "server" run configuration settings and configure Application Server pointing it to `<TeamCity Home Directory>`
- Run "server" run configuration. It will run Ant create distribution task, deploy the plugin into `${user.home}/.BuildServer` directory and run TeamCity server.

If you use 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 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.ResposibilityFunctions
            - 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 whene context is switched to next runner.

`j.b.agent.AgentRunningBuild` provides a context of a build (i.e. shared between all runners). All `addShared*` methods modifies the build context (and thus all build runner contexts).

`j.b.agent.BuildAgentConfiguration` now contains getBuildParameters() and getConfigParameters() methods to access parameters. Configuration parameters here are formed from properties from buildAgent.properties that does not start from 'system.' or 'env.' prefix. All parameters are returned with all references resolved.

`j.b.agent.AgentBuildRunner#createBuildProcess` method signature has been changed to receive `j.b.agent.BuildRunnerContext`.

`j.b.agent.CommandLineBuildService#initialize(...)` method signature has been changed to receive `j.b.agent.BuildRunnerContext`.

`j.b.agent.CommandLineBuildService#getRunnerContext(...)` added

`j.b.agent.CommandLineBuildService#afterProcessSuccessfullyFinished()` added

`j.b.agent.BuildServiceAdapter` is added to simplify as proposed base class for commandline base build runner service.

Changes from 5.0 to 5.1

Web extensions:

- deprecated method removed: `j.b.web.openapi.WebControllerManager.addPageExtension(final WebPlace addTo, final WebExtension extension, Anchor<WebExtension> anchor)`
- deprecated class removed: `j.b.serverSide.Anchor`
- deprecated class removed: `j.b.notification.TemplatePatternProcessor`; `j.b.notification.TemplateProcessor` added instead, see Extending Notification Templates Model
- method removed: `j.b.notification.TemplateMessageBuilder.setPatternProcessor()`
- several methods in `j.b.serverSide.SBuildType` now return `boolean` instead of `void`. You will probably need to recompile your plugins that use the interface.

Changes from 4.5.5 to 5.0

Parameters

`j.b.serverSide.parameters.AbstractBuildParameterReferencesProvider` is renamed to
`j.b.serverSide.parameters.AbstractBuildParametersProvider`
`j.b.serverSide.parameters.BuildParameterReferencesProvider` is renamed into
`j.b.serverSide.parameters.BuildParametersProvider`
`BuildParameterReferencesProvider.getParameters(@NotNull final SBuild build)` changed signature to
`getParameters(@NotNull final SBuild build, final boolean emulationMode)`
`j.b.agent.BuildAgentConfiguration#getCacheDirectory` now receives String as argument
`j.b.serverSide.buildDistribution.StartBuildPrecondition#canStart` second parameters
(`Map<QueuedBuildInfo, BuildAgent>`) may contain null values for some queued builds

Miscellaneous

Added new build server events:
`j.b.serverSide.BuildServerListener.vcsRootRemoved(SVcsRoot)`,
`j.b.serverSide.BuildServerListener.responsibleChanged(SProject, TestNameResponsibilityEntry, TestNameResponsibilityEntry, boolean)`

Added three notification methods:
`j.b.notification.Notificator.notifyResponsibleAssigned(SBuildType, Set<SUser>)`,
`j.b.notification.Notificator.notifyResponsibleChanged(TestNameResponsibilityEntry, TestNameResponsibilityEntry, SProject, Set<SUser>)`,
`j.b.notification.Notificator.notifyResponsibleAssigned(TestNameResponsibilityEntry, TestNameResponsibilityEntry, SProject, Set<SUser>)`

Changes prior to 4.5.5

Not documented

Plugins Packaging

This page describes how plugins should be packaged by plugin developers. See Installing Additional Plugins for instruction on plugin installation.

To write TeamCity plugin it is beneficial to have some knowledge about Spring Framework. Server-side and agent-side TeamCity plugins are initialized via Spring container. This means that every plugin should have a Spring bean definition file describing main plugin services.

There is a convention how bean definition file should be named:

- **build-server-plugin-<plugin name>***.xml — for server side plugins
- **build-agent-plugin-<plugin name>***.xml — for agent side plugins

where an asterisk can be replaced with any text, for example: **build-server-plugin-cvs.xml**. Bean definition files should be placed into the `META-INF` folder of the JAR archive containing plugin classes.

Web resources packaging

In most cases plugin is just a number of classes packed into a JAR file, but if you wish to write custom page for TeamCity, most likely you'll need to place images, CSS, JavaScript, and JSP files somewhere. Files that you want to access via hyperlinks and JSP pages you should place into the **buildServerResources** subfolder. Upon server startup these files will be extracted from the archive and moved into the **<TeamCity home >/webapps/ROOT/plugins/<plugin name>** directory (read more on how to construct paths to your JSP files).

Installation of TeamCity plugins

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, newer version) put into `<TeamCity data directory>/plugins` will override plugins put into `<TeamCity web application>/WEB-INF/plugins` directory.

You can put plugin into `plugins` directory as a separate folder or as a zip archive. Note that server must be restarted to load your plugin.

If you use a *separate folder*:

- TeamCity will use the folder name as plugin name

If you use a *zip file*:

- TeamCity will use name of the zip file as the plugin name
- Plugin zip file will be automatically unpacked on server startup to directory with the same name

Inside plugin directory there should be the following structure:

```
agent
  |
  --> <agent plugin zip>
server
  |
  --> <server plugin jar files>
teamcity-plugin.xml
```

agent directory must contain one file only: <*agent plugin zip*> which should be prepared so that all files and directories are placed into the single root directory. That is, there must be one root folder in the archive (<*plugin top level directory*> in the diagram below), and there should not be other files at the top level. Usually for convenience the name of <*plugin top level directory*> is the same as the plugin name. All jar files required by the plugin on the agent must be placed to the lib subfolder:

```
<plugin top level directory>
  |
  --> lib
       |
       --> <jar files>
```

server folder contains server side part of the plugin, that is, a bunch of jar files.

teamcity-plugin.xml contains some meta information about plugin, like its name and version, see below.

Plugins Loading

On the agent side all plugins and their libraries are always loaded by shared classloader.

On the server side by default plugins are loaded in the shared classloader too, however it is possible to load plugin in separate classloader by specifying special parameters in the `teamcity-plugin.xml` file.

Agent upgrade

When server starts it places all agent plugins into the `<TeamCity home>/webapps/ROOT/update/plugins` folder. After that if content of the folder has changed, agents will receive upgrade command from the server and download updated files automatically. Note that server tracks changes in this folder and can initiate agent upgrade procedure on the fly. In practice it means that if you want to deploy updated agent part of your plugin without server restart you can put your agent plugin to this folder.

After successful upgrade your plugin will be unpacked into the `<Agent home>/plugins/` folder. Note that if agent is busy running build it won't upgrade until the build finishes. Also no new builds will start on the agent if it should be updated.

Plugin Descriptor (`teamcity-plugin.xml`)

`teamcity-plugin.xml` file should be placed to the root of plugin folder. You can refer to 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>
    <version>0.239.42</version>
  </info>
  <deployment use-separate-classloader="true" /> <!-- load server plugin's classes in separate
classloader-->
</teamcity-plugin>
```

Please, note that `use-separate-classloader="true"` parameter is for server-side plugins only, it does not affect agent-side plugins. Until you are using some libraries which clash with TeamCity libraries, it is recommended to leave default behavior, that is, to use shared classloader.

Server side plugins also can obtain **jetbrains.buildServer.web.openapi.PluginDescriptor** implementation with help of SpringFramework Injecting dependencies,
Autowiring collaborators autowiring feature.

Starting from TeamCity 5.0 it is allowed to define plugin parameters in the **teamcity-plugin.xml** file:

```
<?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>
    <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>
```

Plugin parameters can be accessed from **jetbrains.buildServer.web.openapi.PluginDescriptor#getParameterValue(String)** method.

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.
jetbrains.buildServer.serverSide.parameters.BuildParametersProvider	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.
jetbrains.buildServer.serverSide.parameters.ParameterDescriptionProvider	5.0	Provides a human readable description for a parameter, see also **BuildParametersProvider**.
jetbrains.buildServer.messages.serviceMessages.ServiceMessageTranslator	4.0	Translator for specific type of service messages.
jetbrains.buildServer.usageStatistics.UsageStatisticsProvider	6.0	Provides a custom usage statistics.

See also:

Extending TeamCity: Developing TeamCity Plugins | Web UI Extensions

Web UI Extensions

This section covers:

- Developing a Page Extension
- Developing a Custom Tab
- 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

Developing a Page Extension

In TeamCity *page extension* is a plugin written for a specific extension point and extending existing web page functionality. In most cases page extensions only provide some additional information and a simple UI that does not require communication with the server.

TeamCity contains a number of pages that can be extended in such a way. These pages provide one or more *extension points* which are contained in the **PlaceId** class.
There are also two special extension points `ALL_PAGES_HEADER` and `ALL_PAGES_FOOTER`. These extension points allow you to insert your plugin content on every page (except external pages not requiring authentication like login) in the header and/or footer.

To write a page extension:

1. Choose an extension point.
2. Implement interface **jetbrains.buildServer.web.openapi.PageExtension** (or you can use/extend **jetbrains.buildServer.web.openapi.SimplePageExtension** class).
3. Attach the implemented interface to a chosen extension point (ask Spring to provide you **jetbrains.buildServer.web.openapi.PagePlaces** interface):
   `pagePlaces.getPlaceById(PlaceId.SOME_EXTENSION_POINT).addPageExtension(myExtension)`

The **jetbrains.buildServer.web.openapi.PageExtension** interface has the following important methods that require your attention:

- `getIncludeUrl` is a mandatory method. You must specify its `includeUrl` parameter — an URL that is used to retrieve your extension content. This URL must point to a JSP file or to a custom controller.
- `isAvailable(HttpServletRequest)` method is called to determine whether page extension content should be shown or not.
- `fillModel(Map, HttpServletRequest)` method is required when you want to pass some parameters to your JSP. The method will be called before actual JSP is shown and a map of parameters will be passed to it. You can add parameters to this map and then in JSP you will be able to retrieve them by their names from the request scope.

jetbrains.buildServer.web.openapi.SimplePageExtension class requires **jetbrains.buildServer.web.openapi.PagePlaces** interface as parameter, plus **jetbrains.buildServer.web.openapi.PlaceId** specification. Given that `PlaceId` is specified, you can call method `register()` to register this extension. You can also use `SimplePageExtension` class to specify your extension in Spring xml descriptor, because it has setters for all main parameters.

Developing a Custom Tab

A number of extension points in TeamCity are *custom tab extensions*. For example, `PlaceId.BUILD_RESULTS_TAB` identifies custom tab on build results page.

To add your own tab you can extend `SimplePageExtension` and additionally implement **jetbrains.buildServer.web.openapi.CustomTab** interface with two new methods:

- `CustomTab.getTabId()` returns unique identifier of the tab among all of the tabs in this extension point.
- `CustomTab.getTabTitle()` returns title to show in the tab.

Processing of custom tabs has some differences from usual page extensions:

- If **jetbrains.buildServer.web.openapi.PageExtension#isAvailable(javax.servlet.http.HttpServletRequest)** method returns false then tab will not be shown and user will not be able to switch to it. So if your tab should always be visible return true in this method.

- `PageExtension.fillModel(Map, HttpServletRequest)` method is called only if the tab is selected by the user.

In all other respects custom tabs are processed as usual page extensions.

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).

To register your controller:

- use **jetbrains.buildServer.web.openapi.WebControllerManager#registerController(java.lang.String, org.springframework.web.servlet.mvc.Controller)** method with the following arguments:
  - First argument of this method (***String***) is a path to which the controller will be bound. The path must end with ".html" suffix, for example: `/myplugin/mycontroller.html`.
  - Second argument (***Controller***) is the controller itself.

ℹ We are using Spring MVC web framework.

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. Links to such pages you can add by means of page extensions.

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:

```
<img src="${teamcityPluginResourcesPath}your_image.gif" height="16" width="16" border="0">
```

Note: <c:url/> is required to construct correct URL in case if TeamCity is deployed under the non root context.

Classes and interfaces from TeamCity web open API

Class / Interface	Description
jetbrains.buildServer.web.openapi.PlaceId	A list of page place identifiers / extension points
jetbrains.buildServer.web.openapi.PagePlace	A single page place associated with **PlaceId**, allows to add / remove extensions
jetbrains.buildServer.web.openapi.PageExtension	Page extension interface
jetbrains.buildServer.web.openapi.SimplePageExtension	Base class for page extensions
jetbrains.buildServer.web.openapi.CustomTab	Custom tab extension interface
jetbrains.buildServer.web.openapi.PagePlaces	Maintains a collection of page places and allows to locate **PagePlace** by **PlaceId**
jetbrains.buildServer.web.openapi.WebControllerManager	Maintains a collection of custom controllers, allows to register custom controllers
jetbrains.buildServer.controllers.BaseController	Base class for controllers

Plugin Settings

Server-wide settings

A plugin can store server-wide setting in `main-config.xml` file (which is 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 to load and save some data in XML format (via JDOM). Please note, that the plugin will be asked to reinitialize data if the file was changed on the disk while TeamCity is up and running.

Project-wide settings

Per-project settings can be stored in `TEAMCITY_DATA_PATH/config/<project-name>/plugin-settings.xml` directory.
To manage 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 constructor injection). Upon registration, you should specify the name of the XML node to store settings under.

You settings should be serialized to XML format by your implementation of **jetbrains.buildServer.serverSide.settings.ProjectSettings** interface. Methods `readFrom` and `writeTo` should be implemented consistently.

When your code needs 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 **jetbrains.buildServer.serverSide.SProject** #persist() call, or via `ProjectManager#persistAllProjects`. This can be done, for instance, upon some event (see **jetbrains.buildServer.serverSide.BuildServerAdapter** #serverStartup()).

Development Environment

Plugin Reloading

If you make changes to a plugin, you will generally need to shut down the server, update the plugin, and start the server again.

To enable TeamCity development mode, pass the "`teamcity.development.mode=true`" internal property. Using the option you will:

- Enforce application server to quicker recompile changed `.jsp` classes
- Disable JS and CSS resources merging/caching

The following hints can help you eliminate the restart in the certain cases:

- if you do not change code affecting plugin initialization and change only body of the methods, you can attach to the server process with a debugger and use Java hotswap to reload the changed classes from your IDE without web server restart. Note that the standard hotswap does not allow you to change method signatures.
- if you make a change in some resource (jsp, js, images) you can copy the resources to `webapps/ROOT/plugins/<plugin-name>` directory to allow Tomcat to reload them.
- change in build agent part of plugin will initiate build agents upgrade.

If you replace a deployed plugin .zip file with changed class files while TeamCity server is running, this can lead to NoClassDefFound errors.
To avoid this, set "`teamcity.development.shadowCopyClasses=true`" internal property. This will result in:

- creating "`.teamcity_shadow`" directory for each plugin .jar file;
- avoid .jar files update on plugin archive change.

See also:

Extending TeamCity: Developing TeamCity Plugins | Plugins Packaging

Typical Plugins

This section covers:

- Build Runner Plugin
- 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
- Extending Highlighting for Web diff view

Build Runner Plugin

Build runner plugin consists of two parts: agent-side and server-side. Server side part of the plugin provides meta information about build runner, web UI for build runner settings and build runner properties validator. Agent side part launches build.

Build runner can have various settings which must be edited by the user on 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 which source code can be used as a reference:

- http://github.com/orj/teamcity-xcode
- http://www.jetbrains.net/confluence/display/TW/Rake+Runner
- FxCop runner sources

Server side part of the runner

The main entry point for the runner on the server side is **jetbrains.buildServer.serverSide.RunType** . Build runner plugin must provide its' own RunType and register it in the **jetbrains.buildServer.serverSide.RunTypeRegistry** .

RunType has a **type** which should be unique among all build runners and correspond to **type** returned by agent-side part of the runner (see **jetbrains.buildServer.agent.AgentBuildRunnerInfo**).

Methods **getEditRunnerParamsJspFilePath** and **getViewRunnerParamsJspFilePath** shall return paths to JSP files for editing and viewing runner settings. These JSP files should be bundled with plugin in **buildServerResources** subfolder, read more. Paths should be relative to the **buildServerResources** folder.

> Starting from TeamCity 5.1, path to 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. Plugin class may use **PluginDescriptor#getPluginResourcesPath()** method to create a path to a .jsp file from buildServerResources folder of a plugin

> TeamCity 5.0.x and earlier uses the following rule to compute a full path to runner's jsp:
>
> ```
> <context path>/plugins/<runType>/<returned jsp path>
> ```

> Hint: before writing your own JSP for custom build runner take a look at the JSP files of the existing runners bundled with TeamCity.

When user fills in your runner settings and submits form a **jetbrains.buildServer.serverSide.PropertiesProcessor** returned by **getRunnerPropertiesProcessor** method will be called. This processor will be able to verify user settings and indicate which of them are invalid.

Usually 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 help of **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 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 factory interface **CommandLineBuildServiceFactory** and make your class a Spring bean. Factory also provides some meta information about runner via **jetbrains.buildServer.agent.AgentBuildRunnerInfo** .

CommandLineBuildService is an abstract class which simplifies external processes launching and allows to listen for process events (output, finish and so on). Your runner should extend this class. Starting from TeamCity 6.0 we introduced **jetbrains.buildServer.agent.runner.BuildServiceAdapter** class that extends **CommandLineBuildService** and provides could utility methods to access build and runner context parameters.

AgentBuildRunnerInfo has two methods: **getType** which must return the same **type** returned by the server side part of the plugin, and **canRun** which is called to determine whether custom runner can run on the agent (in the agent environment).

If command line build service is not suitable for your needs you can still implement **AgentBuildRunner** interface and defined it in the Spring context. Then it will be loaded automatically.

Logging to build log

Usually build runner starts 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 service message is a specially formatted text with attributes, when such text is logged to the process output it is parsed and associated processing is performed. With help of these messages you can create TeamCity hierarchical build log, report tests, errors and so on.

If 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 external Java process: runtime-util.jar, server-logging.jar. Then you should use **jetbrains.buildServer.agent.ant.LoggerFactory** method to construct logger: LoggerFactory.createBuildProgressLogger(parentClassloader). Since this way is more involved it is recommended to use service messages instead.

If logging of the messages is done in the agent JVM (not from within the external process started by your runner) you can obtain **jetbrains.buildServer.agent.BuildProgressLogger** from the **jetbrains.buildServer.agent.AgentRunningBuild#getBuildLogger** method.

Extending Ant runner

TeamCity Ant runner while being a plugin itself can also be extended with help of **jetbrains.buildServer.agent.ant.AntTaskExtension** . This extension works in the same JVM where Ant is running. With help of 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 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.

Custom Build Trigger

An example of a trigger plugin can be found in Url Build Trigger.

Build Trigger Service

Build trigger is a service whose purpose is to trigger builds (add builds to the queue). Build trigger must extend **jetbrains.buildServer.buildTriggers.BuildTriggerService** abstract class. Build trigger service is uniquely identified by trigger name (see **getName** method). There is no need to register BuildTriggerService, instead plugin should provide a class extending the **jetbrains.buildServer.buildTriggers.BuildTriggerService** defined as a Spring bean.

Build Trigger Settings

Build trigger settings is an object containing build trigger parameters specified by a user via the web UI. Build trigger settings are represented by **jetbrains.buildServer.buildTriggers.BuildTriggerDescriptor** class. The settings are contained within a map of string parameters. More than one instance of **jetbrains.buildServer.buildTriggers.BuildTriggerDescriptor** corresponding to the same trigger service can be added to the build configuration or a template. Instances of the **jetbrains.buildServer.buildTriggers.BuildTriggerDescriptor** with the same trigger name and parameters are considered equal. With help of **jetbrains.buildServer.buildTriggers.BuildTriggerService#isMultipleTriggersPerBuildTypeAllowed()** trigger service can allow or disallow multiple trigger settings per build configuration.

Each trigger service can provide an URL to a jsp or custom controller which will show the trigger web UI. The approach is similar to the one used for VCS roots and build runners.

Triggering Policy

Build trigger service must return a policy (**jetbrains.buildServer.buildTriggers.BuildTriggeringPolicy**) which will be used to add builds to the queue. Currently only one policy is available: **jetbrains.buildServer.buildTriggers.PolledBuildTrigger** . More policies can be added in the future.

Trigger returning **jetbrains.buildServer.buildTriggers.PolledBuildTrigger** policy will be polled by the server with regular intervals. Trigger will receive **jetbrains.buildServer.buildTriggers.PolledTriggerContext** object which contains all information necessary to make a decision whether a build must be triggered or not. Trigger can use **jetbrains.buildServer.serverSide.SBuildType#addToQueue(java.lang.String)** method to add builds to the queue. Note that **jetbrains.buildServer.buildTriggers.PolledTriggerContext** also provides access to the custom data storage. This storage can be used for build trigger state associated with a build configuration and trigger settings. Custom storage will be automatically persisted and restored upon server restart.

Extending Notification Templates Model

You can extend data model passed into notification templates when evaluating.

In your plugin, implement `TemplateProcessor` interface. The following example can be found in our sample plugin:

```
public class SampleTemplateProcessor implements TemplateProcessor {
  public SampleTemplateProcessor() {
  }

  @NotNull
  public Map<String, Object> fillModel(@NotNull NotificationContext context) {
    Map<String, Object> model = new HashMap<String, Object>();
    model.put("users", context.getUsers());
    model.put("event", context.getEventType());
    return model;
  }
}
```

Issue Tracker Integration Plugin

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):

- **SIssueProvider** - represents a single provider
- **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 (`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 (`SIssueProvider` and `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:

- **AbstractIssueProviderFactory**
- **AbstractIssueProvider**
- **AbstractIssueFetcher** - a helper entity which encapsulates fetch-related logic

`AbstractIssueProvider` implements a simple caching provider able to extract the issues from the string based on a regexp. In most cases you just need to derive from it and override few methods. A simple derived provider class can look like this:

```
public class MyIssueProvider extends AbstractIssueProvider {
  // Let's name the provider simple: "myName". The plugin name should be the same.
  public MyIssueProvider(@NotNull IssueFetcher fetcher) {
    super("myName", fetcher);
  }

  // Means that issues are in format "PREFIX-123', like in Jira or YouTrack.
  // The prefix is configured via properties, regexp is invisible for users.
  protected boolean useIdPrefix() {
    return true;
  }
}
```

Providers like Bugzilla might need to override `extractId` method, because the mention of issue id (in comment) and the id itself can differ. For instance, suppose the issues are referenced by a hash with a number, e.g. #1234; the regexp is `"#(\d{4})"` (configurable); but the issues in ITS are represented as plain integers. Then the provider must extract the substrings matching `"#(\d{4})"` and return the first groups only. You should implement it in `extractId` method:

```
@NotNull
protected String extractId(@NotNull String match) {
  Matcher matcher = myPattern.matcher(match);
  matcher.find();
  return matcher.group(1);
}
```

The factory code is very simple as well, for example:

```
public class MyIssueProviderFactory extends AbstractIssueProviderFactory {
  public MyIssueProviderFactory(@NotNull IssueFetcher fetcher) {
    // Type name usually starts with uppercase character because it is displayed in UI, but not
necessarily.
    super(fetcher, "MyName");
  }

  @NotNull
  public IssueProvider createProvider() {
    return new MyIssueProvider(myFetcher);
  }
}
```

IssueFetcher is usually the central class performing plugin-specific logic. You have to implement `getIssue` method, which connects to the ITS remotely (via HTTP, XML-RPC, etc), passes authentication, retrieves the issue data and returns it, or reports an error. Example:

```
public IssueData getIssue(@NotNull String host, @NotNull String id,
                          @Nullable final Credentials credentials) throws Exception {
  final String url = getUrl(host, id);
  return getFromCacheOrFetch(url, new FetchFunction() {
    @NotNull
    public IssueData fetch() throws Exception {
      InputStream body = fetchHttpFile(url, credentials);
      IssueData result = null;
      if (body != null) {
        result = parseXml(body, url);
      }
      if (result == null) {
        throw new RuntimeException("Failed to fetch issue from \"" + url + "\"");
      }
      return result;
    }
  });
}
```

You need to implement how to compose the server URL and how do you parse the data out of XML (HTML). `AbstractIssueFetcher` will take care about caching, errors reporting and everything else.

Plugin UI

The only mandatory JSP required by TeamCity is `editIssueProvider.jsp` (the full path must be `/plugins/myName/admin/editIssueProvider.jsp`, that is, the plugin should have the jsp available `/admin/editIssueProvider.jsp` of its resources). This JSP is requested when the user opens the dialog for editing (or creating) the issue provider. In most cases it just renders the provider properties, or returns the form for filling them.

You can see the example in `/plugins/youtrack/admin/editIssueProvider.jsp`.

Version Control System Plugin

Overview

In TeamCity a plugin for Version Control System (VCS) is seen as a set of interface implementations grouped together by instances of

jetbrains.buildServer.vcs.VcsSupportContext

(server-side part) and

jetbrains.buildServer.agent.vcs.AgentVcsSupportContext

(agent-side part).
The server-side part of a VCS plugin is responsible 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 TeamCity plug-ins 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 Version, Modification, Change, Patch, Checkout Rule, which are explained below.

Basic Terms

A *Version* is unambiguous representation of a particular snapshot within a repository pointed at by a VCS Root. The current version represents the head revision at the moment of obtaining.

The current version is taken by calling

jetbrains.buildServer.vcs.VcsSupportCore#getCurrentVersion(jetbrains.buildServer.vcs.VcsRoot)

. The version here is an arbitrary text. It can be a representation of a transaction number, a revision number, a date, whatever suitable enough for getting a source snapshot in a particular VCS. Usually format of the version depends on a version control system, the only requirement which comes from TeamCity — it should be possible to sort changes by version in order of their happening (see **jetbrains.buildServer.vcs.VcsSupportConfig#getVersionComparator()**).
Version is used in several places:

- for changes collecting
- for patch construction
- when content of the file is retrieved from the repository
- for labeling / tagging

TeamCity does not show Versions in the UI directly. For UI TeamCity converts a Version to its display name using

jetbrains.buildServer.vcs.VcsSupportConfig#getVersionDisplayName(String,jetbrains.buildServer.vcs.VcsRoot)

.

A *Change* is an atomic modification of a single file within a source repository. In other words a Change corresponds to a single increment of a file version.

A *Modification* is a set of Changes made by some user at a certain time interval. It most closely corresponds to a single checkin transaction, 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 ... TODO

A *Checkout Rule* is a way of changing default file layout.

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.

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 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 coresponding to different include rules cannot be aggregated under the same `ModificationData` instance even if they logically relate to the same commit transaction. This means a user will see these changes in separate change lists, which may be confusing. Experience shows that it's not very common situation when a user commits to directories monitored with different include rules. However, if the duplication is extremely undesirable an implementation should not utilize `VcsSupportUtil.collectBuildChanges(...)` and control `CheckoutRules` itself.

Another limitation is complete ignorance of exclude rules. As it said before this doesn't cause showing unneeded information in the UI. So an implementation can safely use `VcsSupportUtil.collectBuildChanges(...)` if this ignorance doesn't lead to significant performance problems. However, If an implementer believes the change collection speed can be significantly improved by taking into account include rules, the implementation must handle exclude rules itself.

All above is applicable to building patches using `VcsSupportUtil.buildPatch(...)` including the path relativity aspect.

Server-Side Caching

By default server caches clean patches created by VCS plugins, because on large repositories clean patch construction can take significant time. If clean patches created by your VCS plugin must not be cached, you should return **true** from the method **VcsSupport#ignoreServerCachesFor(VcsRoot)**.

Agent-Side Part

Agent-Side Checkout

Agent part of VCS plugin is optional, if it is provided then checkout can also be performed on the agent itself. This kind of checkout usually works faster but it may require additional configuration efforts, for example, if VCS plugin uses command line client then this client must be installed on all of the agents.

To enable agent-side checkout, be sure to include **jetbrains.buildServer.agent.vcs.AgentVcsSupportContext** into agent plugin part and also enable agent-side checkout via **jetbrains.buildServer.vcs.VcsSupportConfig#isAgentSideCheckoutAvailable()** .

Version Control System Plugin (old style - prior to 4.5)

In TeamCity a plugin for Version Control System (VCS) is seen as an **jetbrains.buildServer.vcs.VcsSupport** instance. All VCS plugins must extend this class.

VCS plugin has a server side part and an optional agent side part. The server side part of a VCS plugin should support the following mandatory operations:

- collecting changes between versions
- building of a patch from version to version
- get content of a file (for web diff, duplicates finder, and some other places)

There are also optional parts:

- labeling / tagging
- personal builds

Personal builds require corresponding support in IDE. Chances are we will eliminate this dependency in the future.

> You can use source code of the existing VCS plugins as a reference, for example:
>
> - http://www.jetbrains.net/confluence/display/TW/Mercurial
> - http://www.jetbrains.net/confluence/display/TW/AccuRev

Before digging into the VCS plugin development details it's important to understand the basic notions such as Version, Modification, Change, Patch, Checkout Rule, which are explained below.

Version

A *Version* is unambiguous representation of a particular snapshot within a repository pointed at by a VCS Root. The current version represents the head revision at the moment of obtaining.

The current version& is obtained from the **VcsSupport#getCurrentVersion(VcsRoot)**. The version here is arbitrary text. It can be transaction number, revision number, date and so on. Usually format of the version depends on a version control system, the only requirement which comes from TeamCity - it should be possible to sort changes by version in order of their appearance (see **VcsSupport#getVersionComparator()** method).

Version is used in several places:

- for changes collecting

- for patch construction
- when content of the file is retrieved from the repository
- for labeling / tagging

TeamCity does not show Versions in the UI directly. For UI, TeamCity converts a Version to its display name using **VcsSupport#getVersionDisplayName(String, VcsRoot)**.

Collecting Changes

A *Change* is an atomic modification of a single file within a source repository. In other words, a Change corresponds to a single increment of the file version.

A *Modification* is a set of Changes made by some user at a certain moment. It most closely corresponds to a single checkin transaction, when a user commits all his modifications made locally to the central repository. A Modification also contains the Version of the VCS Root right after the corresponding Changes have been applied.

TeamCity server polls VCS for changes on a regular basis. A VCS plugin is responsible for collecting information about Changes (grouped into Modifications) between two versions.

Once a VCS Root is created the first action performed on it is determining the current Version (**VcsSupport#getCurrentVersion(VcsRoot)**). This value is stored and used during the next checking for changes as the "from" Version (**VcsSupport#collectBuildChanges(VcsRoot, String, String, CheckoutRules)**). The current Version is obtained again to be used as the "to" Version. The Modifications collected are then shown as pending changes for corresponding build configurations. After the checking for changes interval passes the server requests for next portion of changes, but this time the "from" Version is replaced with the previous "current" Version. And so on.

Obtaining the current Vesion may be an expensive operation for some version control systems. In this case some optimization can be done by implemeting interface **CurrentVersionIsExpensiveVcsSupport**. Its method **CurrentVersionIsExpensiveVcsSupport#collectBuildChanges(VcsRoot, String, CheckoutRules)** takes only "from" Version assuming that the changes are to be collected for the head snapshot. In this case TeamCity will look for the Modification with the greatest Version in the returned Modifications and take it as the "from" parameter for the next checking cycle. If you implement **CurrentVersionIsExpensiveVcsSupport**, 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 ...
}
```

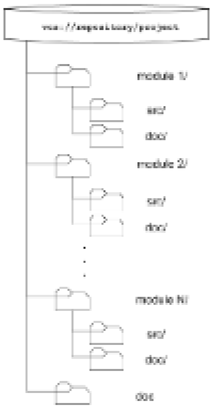
If you want to share data between calls, this approach allows you to do it easily using anonymous classes:

```
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:

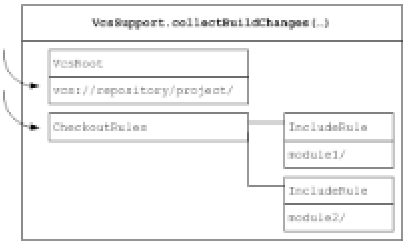


We want to monitor changes only in module1 and module2. Therefore we've configured the following checkout rules:

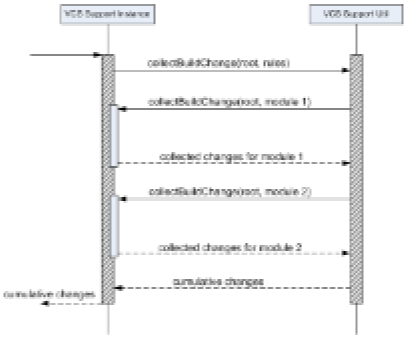
```
\+:module1

\+:module2
```

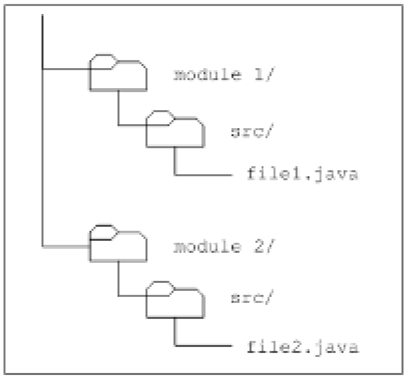
When `collectBuildChanges(...)` is invoked it will receive a `VcsRoot` instance that corresponds to `vcs://repository/project/|| and a {{CheckoutRules` instance with two `IncludeRules` — one for "module1" and the other for "module2".



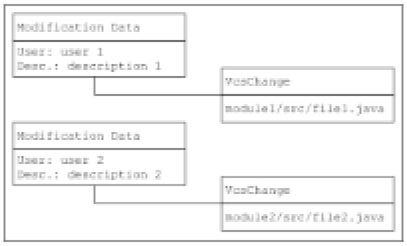
If `collectBuildChanges(...)` utilizes `VcsSupportUtil.collectBuildChanges(...)` it transforms the invocation into two separate calls of `CollectChangesByIncludeRule.collectBuildChange(...)`. If you have implemented `CollectChangesByIncludeRule` in the way described in the listing above you will have the following interaction.


VCS Support Instance
VCS Support Util
collectBuildChange(root, rules)
collectBuildChange(root, module 1)
collected changes for module 1
collectBuildChange(root, module 2)
collected changes for module 2
cumulative changes
cumulative changes

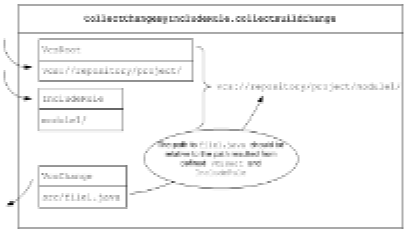
Now let's assume we've got a couple of changes in our sample repository, made by different users.



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.

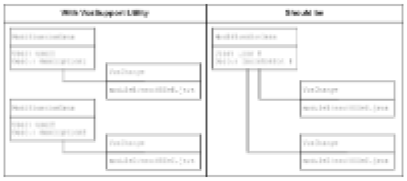

collectChangesByIncludeRule.collectBuildChange
VcsRoot
vcs://repository/project/
IncludeRule
module1/
vcs://repository/project/module1/
The path to file1.java should be relative to the path resulted from defined VcsRoot and IncludeRule
VcsChange
src/file1.java

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 beleives 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

Custom authentication API is based on Sun JAAS API. To provide your own authentication scheme, you should provide a login module class which must implement interface **javax.security.auth.spi.LoginModule** and register it in the **jetbrains.buildServer.serverSide.auth.LoginConfiguration**.

To make the authentication module active the login module class name can then be used during configuring 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 LoginException(t.toString());
}

// 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;
}

return false;
}

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 implements
jetbrains.buildServer.serverSide.auth.LoginModuleDescriptor {
// 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 Class<? extends LoginModule> getLoginModuleClass() {
// return our custom login module class
return CustomLoginModule.class;
}

public Map<String, ?> getOptions() {
// return options which can be passed to our custom login module
// for example, we could store reference to SBuildServer instance here
return null;
}

public String getTextForLoginPage() {
// return text to show on the login page which could describe
// what should be entered in the user name and password fields
return null;
}

public String getDisplayName() {
// presentable name of this plugin
return "My custom authentication plugin";
}
}
```

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 by specifying name of your class in the main-config.xml file, read more.

Custom Notifier

Custom notifier must implement **jetbrains.buildServer.notification.Notificator** interface and register implementation in the **jetbrains.buildServer.notification.NotificatorRegistry** .

When a notifier is registered, it can provide information about additional properties that must be filled in by the user. To obtain values of these properties, use the following code:

```
String value = user.getPropertyValue(new NotificatorPropertyKey(<notifier type>, <property name>));
```

Notifier can also provide custom UI for Notifier rules and **My Settings&Tools** pages. See **PlaceId.NOTIFIER_SETTINGS_FRAGMENT** and **PlaceId.MY_SETTINGS_NOTIFIER_SECTION**.

Notifications are only delivered if there is at least one subscribed user for given event.

Use source code of the existing plugins as a reference:

- http://code.google.com/p/buildbunny/wiki/CreateTeamcityNotifier
- 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 any extension place and so on.

Quick Start

- TODO: how to use a plugin sample project, how to package and deploy a plugin (links?)
- An easy way to add custom statistics is to just insert a jsp fragment into WebPlace using a helper bean:

```
<bean id="myLogoFragment" class="jetbrains.buildServer.web.openapi.SimpleWebExtension"
init-method="register">
    <property name="name" value="myLogoFragment"></property>
    <property name="place" value="PAGE_HEADER"></property>
    <property name="jspPath" value="/myLogoFragment.jsp"/>
  </bean>
```

- To insert statistics chart into a jsp page:

```
<%@taglib prefix="stats" tagdir="/WEB-INF/tags/chart"%>
<stats:buildChart id="g1" valueType="BuildDuration"/>
```

- To add a custom build metric, extend `BuildValueTypeBase` to just define your build metric calculation method, appearance, and key. After that you can reference this metric by its key in statistics chart/report tags.

More Details

BuildType Statistics tab extension point

```
//WebControllerManager webControllerManager
webControllerManager.addPageExtension(WebPlace.BUILD_INFO_FRAGMENT, this);
```

Customizing chart appearance

- width, height — chart image size
- hideFilters — suppress filter controls

Adding custom metrics

1. Implement `jetbrains.buildServer.serverSide.statistics.ValueType`, extend `BuildFinishAwareValueTypeBase` or `CompositeVTB` for convenience
2. Register it using `jetbrains.buildServer.serverSide.statistics.ValueProviderRegistry.registerValueProvider`

Custom build metrics details

1. Implement `jetbrains.buildServer.serverSide.statistics.build.BuildFinishAware` in your `ValueType` to be notified of build finished event.
2. Calculate your metric.
3. Employ `jetbrains.buildServer.serverSide.statistics.build.BuildDataStorage.publishValue` to publish your value.
4. Employ `jetbrains.buildServer.serverSide.statistics.build.BuildDataStorage.getDataSet` to retrieve selected data.

See also:

> **Extending TeamCity**: Build Script Interaction with TeamCity

Extending Highlighting for Web diff view

TeamCity uses JHighlight library to render the code on diff view page. Essentially what JHighlight is doing is it takes plain source code, recognizes the language by extension, parses it, and in case of success renders the HTML output where the tokens are highlighted according to the specified settings. Unfortunately JHighlight supports relatively small subset of languages out-of-the-box (major ones like Java, C++, XML, and several more). Here we'd like to present you a HOWTO on adding the support for more languages.

> ⚠ Please note that in the further versions TeamCity may switch to another highlighting engine, so the changes you make will only work while JHighlight is used by TeamCity.

As an example we are implementing a highlighting for properties files, like this one:

```
# Comment on keys and values
key1=value1
foo = bar
x=y
a b c = foo bar baz

! another comment
! more complex cases:
a\=\fb : x\ty\n\x\uzzzz

 key = multiline value \
still value \
still value
the key
```

The implementation consists of the following steps:

- Step one: Writing a lexer using flex language
- Step two: Generating a lexer on java
- Step three: The renderer class.
- Step four: Running the JHighlight
- Including JHighlight Changes into TeamCity Distribution

Step one: Writing a lexer using flex language

To understand this step you might need to familiarize yourself with a JFlex syntax.

There are several things you need to define in a flex file in order to generate a lexer. First of all, token types or, in our case, styles.

```
public static final byte PLAIN_STYLE = 1;
public static final byte NAME_STYLE = 2;
public static final byte VALUE_STYLE = 3;
public static final byte COMMENT_STYLE = 4;
```

These constants will be mapped to the lexems in a source code and to the CSS classes, so at this moment you should decide which tokens are to be highlighted.
We will highlight names, values of properties, comments and plain text, which is just '=' character.

Then you need to specify the states and actual parsing rules:

```
WhiteSpace = [ \t\f]

%state IN_VALUE

%%

/* Rules for YYINITIAL state */
<YYINITIAL> {
  "\n"                           { return PLAIN_STYLE; }

  {WhiteSpace}                   { return PLAIN_STYLE; }

  [Extending Highlighting for Web diff view^=\n\t\f ]+                   { return NAME_STYLE; }

  "="                            { yybegin(IN_VALUE); return PLAIN_STYLE; }

  [#!] [Extending Highlighting for Web diff view^\n]* \n                { return COMMENT_STYLE; }
}

/* Rules for IN_VALUE state */
<IN_VALUE> {
  "\\\n"                         { return VALUE_STYLE; }

  "\n"                           { yybegin(YYINITIAL); return PLAIN_STYLE; }

  [Extending Highlighting for Web diff view^\\\n]+                       { return VALUE_STYLE; }
}

/* error fallback */
.|\n                             { return PLAIN_STYLE; }
```

Our simple lexer has two states: initial (YYINITIAL - it is predefined) and IN_VALUE. In each of these states we try to handle the next character (or a group of characters) using regexp rules.
The rules are applied from the top to the bottom, the first one that matches non-empty string is used. Each rule is associated with the action to be performed on runtime. Here we have only simple actions that return the token constant and sometimes change the state.

To end the composition of a lexer add the common part to be inserted to the Java file. It's unlikely that you need to modify it.
Here's the full result code:

```
package com.uwyn.jhighlight.highlighter;

import java.io.Reader;
import java.io.IOException;

%%

%class PropertiesHighlighter
%implements ExplicitStateHighlighter

%unicode
%pack

%buffer 128

%public

%int

%{
        /* styles */

        public static final byte PLAIN_STYLE = 1;
        public static final byte NAME_STYLE = 2;
        public static final byte VALUE_STYLE = 3;
        public static final byte COMMENT_STYLE = 4;
        /* Highlighter implementation */

        public byte getStartState() {
                return YYINITIAL+1;
        }

        public byte getCurrentState() {
                return (byte) (yystate()+1);
        }

        public void setState(byte newState) {
                yybegin(newState-1);
        }

        public byte getNextToken() throws IOException {
                return (byte) yylex();
        }

        public int getTokenLength() {
                return yylength();
        }

        public void setReader(Reader r) {
                this.zzReader = r;
        }

        public PropertiesHighlighter() {
        }
%}

WhiteSpace = [ \t\f]

%state IN_VALUE

%%

/* Rules for YYINITIAL state */
<YYINITIAL> {
  "\n"                           { yybegin(YYINITIAL); return PLAIN_STYLE; }

  {WhiteSpace}                   { return PLAIN_STYLE; }

  [Extending Highlighting for Web diff view^=\n\t\f ]+                    { return NAME_STYLE; }

  "="                            { yybegin(IN_VALUE);  return PLAIN_STYLE; }

  [#!] [Extending Highlighting for Web diff view^\n]* \n                 { return COMMENT_STYLE; }
}

/* Rules for IN_VALUE state */
<IN_VALUE> {
  "\\\n"                         { return VALUE_STYLE; }

  "\n"                           { yybegin(YYINITIAL); return PLAIN_STYLE; }

  [Extending Highlighting for Web diff view^\\\n]+                       { return VALUE_STYLE; }
}
```

```
/* error fallback */
.|\n                             { return PLAIN_STYLE; }
```

That's it: the lexer is ready. Download the latest JHighlighter sources from the repository (version 1.0) and put this file to the `src/com/uwyn/jhighlight/highlighter` directory of JHighlight distribution.

Step two: Generating a lexer on java

You can compile the code above using a JFlex tool, or amend the build.xml file adding the following task to the "flex" target:

```
<jflex file="${src.dir}/com/uwyn/jhighlight/highlighter/PropertiesHighlighter.flex"
       destdir="${src.dir}"
       verbose="on"
       nobak="on"/>
```

After the compilation we'll have a java class PropertiesHighlighter implementing ExplicitStateHighlighter interface. If the previous steps are done right, you won't need to modify this file by hand.

Step three: The renderer class.

The only JHighlight class left is the renderer corresponding to the generated lexer. This class should extend a XhtmlRenderer class and provide CSS classes correspondence along with default CSS map:

```
package com.uwyn.jhighlight.renderer;

import com.uwyn.jhighlight.highlighter.ExplicitStateHighlighter;
import com.uwyn.jhighlight.highlighter.PropertiesHighlighter;
import com.uwyn.jhighlight.renderer.XhtmlRenderer;
import java.util.HashMap;
import java.util.Map;

public class PropertiesXhtmlRenderer extends XhtmlRenderer {
        // Contains the default CSS styles.
 public final static HashMap DEFAULT_CSS = new HashMap() {{
  put(".properties_plain",
      "color: rgb(0,0,0);");

  put(".properties_name",
      "color: rgb(0,0,128); " +
      "font-weight: bold;");

  put(".properties_value",
      "color: rgb(0,128,0); " +
      "font-weight: bold;");

  put(".properties_comment",
      "color: rgb(128,128,128); " +
      "background-color: rgb(247,247,247);");
 }};

 protected Map getDefaultCssStyles() {
  return DEFAULT_CSS;
 }

        // Maps the token type with the CSS class. E.g. each token of a 'PLAIN_STYLE' type will be
rendered with 'properties_plain' style (see above).
 protected String getCssClass(int style) {
  switch (style) {
   case PropertiesHighlighter.PLAIN_STYLE:
    return "properties_plain";
   case PropertiesHighlighter.NAME_STYLE:
    return "properties_name";
   case PropertiesHighlighter.VALUE_STYLE:
    return "properties_value";
   case PropertiesHighlighter.COMMENT_STYLE:
    return "properties_comment";
  }

  return null;
 }

 protected ExplicitStateHighlighter getHighlighter() {
  return new PropertiesHighlighter();
 }
}
```

You can leave DEFAULT_CSS empty, but in this case the styles should always be present in jhighlight.properties file. But it is essential that PropertiesHighlighter token constants are mapped to the CSS styles.

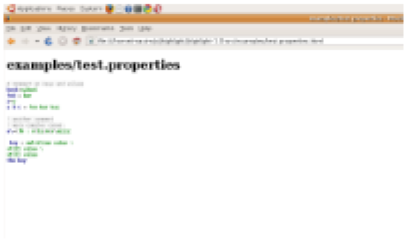
Also we need to tell the factory class that a new renderer exists: for this XhtmlRendererFactory class should be updated. We don't provide the code here as it is very simple (in fact, two lines should be added).

Step four: Running the JHighlight

JHighlight patch is ready, let's check it out in action. Put the properties file to the 'examples' directory and run the commands from JHighlight home directory:

```
ant
java -cp build/classes/ com.uwyn.jhighlight.JHighlight examples/
firefox examples/test.properties.html
```

Voilà! Our properties file is highlighted:



examples/test.properties

Including JHighlight Changes into TeamCity Distribution

TeamCity uses only public JHighlight API, that's why if your patched JHighlight successfully generates the HTML, you have to do just few steps to integrate it to TeamCity:

- repack `jhighlight.jar` (call `ant jar`)
- replace `/WEB-INF/lib/jhighlight-njcms-patch.jar` with it
- restart TeamCity server

Good luck!

REST API

See REST API plugin page at REST API Plugin.

How To...

In this section:

- Integrate with an Issue Tracker
- Install Multiple Agents on the Same Machine
- Watch Several TeamCity Servers with Windows Tray Notifier
- Move TeamCity Installation to a New Machine
- Move TeamCity Agent
- Share the build number for builds in a chain build
- Use an external tool that my build relies on
- Change Server Port
- Make temporary build files erased between the builds
- Retrieve Administrator Password
- How do I clear build queue if it got too many builds due to a configuration error
- Estimate hardware requirements for TeamCity
- Setup TeamCity in Replication/Clustering Environment
- Move TeamCity projects from one server to another
- Automatically create or change TeamCity build configuration settings
- Attach Cucumber reporter to Ant build
- Get last successful build number
- Create a copy of TeamCity server with all data
- Test-drive newer TeamCity version before upgrade
- How do I choose OS/platform for TeamCity server
- How do I set up deployment for my application in TeamCity
- Integrating with Reporting/Metric Tools
- TeamCity Security Notes
- Restore Just Deleted Project
- Set Up TeamCity behind a proxying server
- Transfer 3 default agents to another server

Integrate with an Issue Tracker

TeamCity comes with dedicated support for YouTrack, Jira and Bugzilla.
For any other tracker, you can turn any issue tracker issue ID references in change comments into links. Please see Mapping External Links in

Comments for configuration instructions.

Install Multiple Agents on the Same Machine

See the corresponding section under agent installation documentation.

Watch Several TeamCity Servers with Windows Tray Notifier

TeamCity Tray Notifier is used normally to watch builds and receive notifications from a single TeamCity server. However, if you have more than one TeamCity server and want to monitor them with Windows Tray Notifier simultaneously, you need to start a separate instance of Tray Notifier for each of the servers from the command line with the `/allowMultiple` option:

- From the TeamCity Tray Notifier installation folder (by default, it's `C:\Program Files\JetBrains\TeamCity` run the following command:

  ```
  JetBrains.TrayNotifier.exe /allowMultiple
  ```

  Optionally, for each of the Tray Notifier instances you can explicitly specify the URL of the server to connect using the `/server` option. Otherwise, for each further tray notifier instance you will need to log out and change server's URL via UI.

  ```
  JetBrains.TrayNotifier.exe /allowMultiple /server:http://myTeamCityServer
  ```

See also details in the issue tracker.

Move TeamCity Installation to a New Machine

If you need to move existing TeamCity installation to a new hardware or clean OS, it is recommended to follow instructions on copying the server from one machine to another and then switch from the old server to a new one. If you are sure you do not need the old server data you can probably perform move operations instead of copying.

You can use existing license keys when you move the server from one machine to another (as long as there are no two servers running at the same time). As license keys are stored under <TeamCity Data Directory>, you transfer the license keys with all the other TeamCity settings data.

A usual advice is not to combine TeamCity update with any other actions like environment or hardware changes and perform the changes one at a time so that if something goes wrong the cause can be easily tracked.

Switching from one server to another
Please note that TeamCity Data Directory and database should be used by a single TeamCity instance at any given moment. If you configured new TeamCity instance to use the same data, please ensure you shutdown and disable old TeamCity instance before starting a new one.

Generally it is recommended to use a domain name to access the server (in agent configuration and when users access TeamCity web UI). This way you can update the DNS entry to make the address resolve to the IP address of the new server and after all cached DNS results expire, all clients will be automatically using the new server.

However, if you need to use another server domain address, you will need:

- Switch agents to new URL (requires updating `serverUrl` property in buildAgent.properties on each agent).
- Upon new server startup do not forget to update Server URL on **Administration** | **Global Settings** page.
- Notify all TeamCity users to use the new address

Move TeamCity Agent

Apart from the binaries, TeamCity agent stores it's 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

Suppose you have build configurations A and B that you want to build in sync: use same sources and take the same build number.
Solution:

1. Create a 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 internal ID of the build configuration C. Please refer to the Build Configuration page for description of how to determine build configuration ID.

This reference is also available if you use artifact dependencies instead of snapshot.
Read more about dependency properties.

We plan to provide more option on build number sharing. Please watch/comment on TW-7745.

Use an external tool that my build relies on

If you need to use specific external tool to be installed on a build agent to run your builds, you have the following options:

- Check in the tool into the version control and use relative paths.
- Create a separate build configuration with a single "fake" build which would contain required files as artifacts, then use artifact dependencies to send files to the target build.
- Install and register the tool in TeamCity:
  1. Install the tool on all the agents that will run the build.
  2. Add `env.` or `system.` property into `buildAgent.properties` file (or add environment variable to the system).
  3. Add agent requirement for the property in the build configuration.
  4. Use the property in the build script.
- Add environment preparation stage into the build script to get the tool form elsewhere.

Change Server Port

See corresponding section in server installation instructions.

Make temporary build files erased between the builds

Update your build script to use path stored in `${teamcity.build.tempDir}` (Ant's style name) property as the temp directory. TeamCity agent creates the directory before the build and deletes it right after the build.

Retrieve Administrator Password

On the first start TeamCity displays Administrator Setup page. TeamCity installation should always have a user with System Administrator role in the current authentication scheme.
See also notes when switching from one authentication scheme to another.

If there is no user account with System Administrator role in the current authentication scheme, you can use `http://<your_TeamCity_server>/setupAdmin.html` URL to setup administrator account.
If there is an administrator account in the current authentication scheme, the page is not available and you need to remember the administrator account credentials.

If you forgot Administrator password and use internal database, you can reset the password using the instructions.
Otherwise you can use REST API to add System Administrator role to any existing user.
And here is an instruction to patch roles directly in the database provided by a user.

Related feature requests in our tracker: TW-1964, TW-4524, TW-1681, TW-2456.

How do I clear build queue if it got 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 form the queue.
Also there is an ability to delete many builds from the build queue in a single dialog.

Estimate hardware requirements for TeamCity

The hardware requirements differ for the server and the agents.

The **agent** hardware requirements are basically determined by the builds that are run. Running TeamCity agent software introduces requirement for additional CPU time (but it can usually be neglected comparing to the build process CPU requirements) and additional memory: about 500Mb. 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.

The **server** hardware requirements depend on the server load, which in its turn depends significantly on the type of the builds and server usage. Consider the following general guidelines.

- If you decide to run external database at the same machine with the server, consider hardware requirements with database engine requirements in mind.
- If you face some TeamCity-related Performance issues, they should probably be investigated and addressed individually. e.g. if builds generate too much data, server disk system might need upgrade both by size and speed characteristics.

Database Note:
When using the server extensively, database performance starts to play greater role.
For reliability and performance reasons you should use external database.
Please see notes on choosing external database.
Database size requirements naturally vary based on the amount of data stored. An active server database usage can be estimated at several gigabytes of data. Say, 2 Gb per year.

Overview on the TeamCity hardware resources usage:

- CPU: TeamCity utilizes multiple cores of the CPU, so increasing number of cores makes sense. It is probably not necessary to dedicate more than 8 cores to TeamCity server.
- Memory: See a note on memory usage. Consider also that required memory may depend on the JVM used (32 bit or 64 bit). You will probably not need to dedicate more than 4G of memory to TeamCity server.
- HDD/disk usage: This sums up 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 generated by the builds (size of the build log, number and output size of unit tests, number of inspections and duplicates hits 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.

Based on our experience, a modest hardware like 3.2 dual core CPU, 3.2Gb memory under Windows, 1Gb network adapter 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, main TeamCity process has `-Xmx1100m -XX:MaxPermSize=120m` 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 and add more projects to the server 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.

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 form the fact that our (mostly Java) builds depend on HDD performance in the first place. But YMMV.

TeamCity can work well with up to 200 build agents (200 concurrently running builds). If you need more agents/parallel builds, it is recommended to setup several separate TeamCity instances and distribute the projects between them.

See also a related blog post.

Setup TeamCity in Replication/Clustering Environment

TeamCity does not provide specific support for any of replication/high availability or clustering solutions; however you can replicate the data that TeamCity server uses and prepare to start a new server using the same data if existing server malfunctions.

When setting up TeamCity in a replication environment please note that TeamCity uses both database and file storage to save data. You can browse through TeamCity Data Backup and TeamCity Data Directory pages in to get more information on TeamCity data storing.

Basically, both TeamCity data directory on disk and database that TeamCity uses should remain in a consistent state and thus should be replicated together.

Only single TeamCity server instance should use database and data directory at any time.

Please also ensure that the distribution of the backup server is of exactly the same version as the main server.

See also information on switching from one server to another.

Move TeamCity projects from one server to another

Generally, moving projects to a server that already have projects/build configurations configured is not supported. For addressing simple cases manually, please see a comment.

Automatically create or change TeamCity build configuration settings

If you need a level of automation and web administration UI does not suite your needs, there are two possibilities:

- change configuration files directly on disk (see more at TeamCity Data Directory)
- write a TeamCity Java plugin that will perform the tasks using open API.

Attach Cucumber reporter to Ant build

If you use Cucumber for Java applications testing you should run cucumber with --expand and special --format options. More over you should specify RUBYLIB environment variable pointing on necessary TeamCity Rake Runner ruby scripts:

```
<target name="features">
    <java classname="org.jruby.Main" fork="true" failonerror="true">
      <classpath>
        <pathelement path="${jruby.home}/lib/jruby.jar"/>
        <pathelement path="${jruby.home}/lib/ruby/gems/1.8/gems/jvyaml-0.0.1/lib/jvyamlb.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="${agent.home.dir}/plugins/rake-runner/lib/rb/patch/common;${agent.home.dir}/plugins/rake-runner/li
<env key="TEAMCITY_RAKE_RUNNER_MODE" value="buildserver"/>
    </java>
  </target>
```

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:<internal ID of build
configuration>/builds/status:SUCCESS/number
```

The build number will be returned as a plain-text response.
For `<internal ID of build configuration>`, see Build Configuration#Build Configuration ID.
This functionality is provided by REST API

Create a copy of TeamCity server with all data

One of the ways to create a copy of the server is to create a backup, then install a new TeamCity server of the same version that you already run, ensure you have appropriate environment configured, ensure that the server uses own TeamCity Data Directory and own database and then restore the backup.
This way the new server won't get build artifacts and some other less important data. If you need them, you will need to copy appropriate directories (e.g. "artifacts") from .BuildServer/system from the original to the copied server.

If you do not want to use bundled backup functionality or need manual control over the process, here is a description of the general steps one would need to perform to manually create copy of the server:

1. create a backup so that you can restore it if anything goes wrong,
2. ensure the server is not running,
3. either perform clean installation or copy TeamCity binaries (TeamCity home directory) into the new place (`temp` and `work` subdirectories can be omitted during copying). ⚠ Use exactly the same TeamCity version. If you plan to upgrade after copying, perform the upgrade only after you have existing version up and running.
4. transfer relevant environment if it was specially modified for existing TeamCity installation. This might include:
    - if you run TeamCity with OS startup (e.g. Windows service), make sure all the same configuration is performed on the new machine
    - use the same TeamCity process launching options
    - use appropriate OS user account for running TeamCity server process with appropriately configured settings, global and file system permissions
    - transfer OS security settings if required
    - ensure any files/settings that were configured in TeamCity web UI are accessible; put necessary libraries/files inside TeamCity installation if they were put there earlier)
5. copy TeamCity Data Directory. If you do not need the full copy, refer to the items below for optional items.
    - `.BuildServer/config` to preserve projects and build configurations settings
    - `.BuildServer/lib` and `.BuildServer/plugins` if you have them

    - files from the root of `.BuildServer/system` if you use internal database and you do not want to perform database move.
    - `.BuildServer/system/messages` (optional) if you want build logs (including tests failure details) preserved on the new server
    - `.BuildServer/system/artifacts` (optional) if you want build artifacts preserved on the new server
    - `.BuildServer/system/changes` (optional) if you want personal changes preserved on the new server
    - `.BuildServer/system/pluginData` (optional) if you want to preserve state of various plugins and build triggers
    - `.BuildServer/system/caches` and `.BuildServer/system/caches` (optional) are not necessary to copy to the new server, they will be recreated on startup, but can take some time to be rebuilt (expect some slow down).
6. create copy of the database that your TeamCity installation is using in new schema or new database server
7. configure new TeamCity installation to use proper TeamCity Data Directory and database (`.BuildServer/config/database.properties` points to a copy of the database)

Note: if you want to do a quick check and do not want to preserve builds history on the new server you can skip step 6 (cloning database) and all items of the step 5 marked as optional.

1. ensure the new server is configured to use another data directory and the database then the original server
   At this point you should be ready to run the copy TeamCity server.
2. run new TeamCity server
3. upon new server startup do not forget to update Server URL on **Administration** | **Global Settings** page. You will also probably need to disable Email and Jabber notifiers or change their settings to prevent new server from sending out notifications
4. if you need the services on the copied server check that email, jabber and VCS servers are accessible from the new installation.
5. install new agents (or select some form the existing ones) and configure them to connect to the new server (using new server URL)

See also the notes on moving the server from one machine to another.

Licensing issues
You cannot use a single TeamCity license on two running servers at the same time, so to run a copy of TeamCity server you will need another license.
You can get TeamCity evaluation license 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.

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.

How do I choose OS/platform for TeamCity server

Once the server/OS fulfills the requirements, TeamCity can run on any system. Please also review the requirements for the integrations you plan to use (e.g. integration with Microsoft TFS and VSS will work only under MS Windows)

If you have no preference, Linux platforms may be more preferable due to more effective file system operations and the level of required general OS maintenance.

Final Operating System choice should probably depend more on the available resources and established practices in your organization.

If you choose to install 64 bit OS, TeamCity can run under 64 bit JDK (both server and agent).
However, unless you need to provide more than 1Gb memory for TeamCity, the recommended approach is to use 32 bit JVM even under 64 bit OS. Our experience suggests that using 64 bit JVM does not increase performance a great deal. At the same time it does increase memory requirements to almost the scale of 2. See a note on memory configuration.

How do I set up deployment for my application in TeamCity

1. Write a build script that will perform the deployment task for the binary files available on the disk. (e.g. use Ant or MSBuild for this)
2. Create a build configuration in TeamCity that will execute the build script.
3. In this build configuration configure artifact dependency on a build configuration that produces binaries that need to be deployed
4. Configure one of the available triggers if you need the deployment to be triggered automatically (e.g. to deploy last successful of last pinned build), or use "Promote" action in the build that produced the binaries that need to be deployed. Consider using snapshot dependencies in addition to artifact ones and check Build Chains tab to get the overview of the builds.
5. 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.
6. 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.

Integrating with Reporting/Metric Tools

If you have a tool that generates some report or provides code metrics, you may want to display the data in TeamCity.

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.
For an advanced integration a custom TeamCity plugin can be developed to ease tool configuration and running. See XML Test Reporting and FXCop plugin (see a link on Open-source Bundled Plugins) as an example.

Presenting data in TeamCity
For a report, the most simple approach is 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.

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).

For advanced integration, a custom plugin will be necessary to store and present the data as required. See Developing TeamCity Plugins for more information on plugin development.

TeamCity Security Notes

These notes are provided only for your reference and are not meant to be complete or accurate in their entirety.
TeamCity is developed with security concerns in mind and reasonable efforts are made to make the system not vulnerable to different types of attacks.
However, the general assumption and recommended setup is to deploy TeamCity in a trusted environment with no possibility to be accessed by malicious users.
Here are some notes on different security-related aspects:

- man-in-the middle concerns
  - between TeamCity server and user's web browser: It is advised to use HTTPS for the TeamCity server. During login, TeamCity transmits user login password in an encrypted form with moderate encryption level.
  - between TeamCity agent and TeamCity server: see the section.
  - between TeamCity server and other external servers (version control, issue tracker, etc.): the general rules apply as for a client (TeamCity server in the case) connecting to the external server, see guidelines for the server in question.
- user that has access to TeamCity web UI: the specific information accessible to the user is defined via TeamCity user roles.
- users who can change code that is used in the builds run by TeamCity: the users have the same permissions as the system user under which TeamCity agent is running. Can access and change source code of other projects built on the same agent, modify TeamCity agent code, etc. It is advised to run TeamCity agents under users with only necessary set of permissions and use agent pools feature to insure that projects requiring different set of access are not built on the same agents.
- users with System Administrator TeamCity role: It is assumed that the users also have access to the computer on which TeamCity server is running under the user account used to run the server process. Thus, some operations like server file system browsing can be accessible by the users.
- TeamCity server computer administrators: have full access to TeamCity stored data and can affect TeamCity executed processes. Passwords that are necessary to authenticate in external systems (like VCS, issue trackers, etc.) are stored scrambled under TeamCity Data Directory and can also be stored in the database. However, the values are only scrambled, which means they can be retrieved by the users who have access to the server file system or database.
- TeamCity agent computer administrators: same as "users who can change code that is used in the builds run by TeamCity".
- Other:
  - TeamCity web application vulnerabilities: TeamCity development team makes reasonable effort to fix any significant vulnerabilities (like cross-site scripting possibilities) once they are uncovered. Please note that any user that can affect build files ("users who can change code that is used in the builds run by TeamCity" or "TeamCity agent computer administrators") can make a malicious file available as build artifact that will then exploit cross-site scripting vulnerability.
  - TeamCity agent is fully controlled by the TeamCity server: since TeamCity agents support automatic updates download from the server, agents should only connect to a trusted server. An administrator of the server computer can force execution of arbitrary code on a connected agent.

Restore Just Deleted Project

TeamCity moves settings files of deleted projects under `TeamCity Data Directory/config/_trash` directory.
To restore project you should find its directory on the server and move it one level up. Also you should remove suffix _projectN from the directory name.
You can do this while server is running, it should pick up restored project automatically.

Please note that TeamCity preserves builds history and other data for deleted projects/build configurations for 24 hours since the deletion time. The data id removed during the next cleanup after 24 hours timeout elapses.

Set Up TeamCity behind a proxying server

Internal TeamCity server should work under the same context as it is visible from outside by external address.

Provided:
TeamCity server is installed at URL: teamcity.local:8111
It is visible to the outside world as URL: teamcity.public:400

Then use:
for Apache

> *ProxyPass /tc http://teamcity.local:8111/tc*
> *ProxyPassReverse /tc http://teamcity.local:8111/tc*

for Nginx

```
server {
        listen       400;
        server_name  teamcity.public;

        location /tc {
            proxy_pass http://teamcity.local:8111/tc;
        }
    }
```

Alternative approach is to setup a proxying server to redirect requests to TeamCity server to a dedicated port and edit <TeamCity home>\conf\server.xml to change existing or add 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"
               />
```

(You will need to use secure="true" and scheme="https" for HTTPS)

This latter approach is also described in the comment.

Transfer 3 default agents to another server

This is not possible.

Each TeamCity server (Professional and Enterprise) allows to use 3 agents without any licenses.
It's a "function" of a server: users do not pay for these agents, there is no license key for them, nor are they bound to the server license key.

So, these 3 agents cannot be transferred to another server as they are "bound" to the server instance.

Each TeamCity server allows to connect agents up to number of agent license keys +3

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 Known Issues and Common Problems sections, collect relevant information using 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 did not 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, this 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 relevant data related to the problem and post a new issue into the tracker. Be sure to include TeamCity build number, describe where exactly you see the problem, what were the previous actions if relevant and please also describe your environment (OS, Web Server, TeamCity distribution used, how TeamCity is set up, etc.)
- Contact us to report an issue or ask a question using the general guidelines described.
- If you own Enterprise TeamCity license and need to submit information that is not meant to be public, you can also contact the development team via e-mail.

See also:

Troubleshooting: Known Issues | Reporting Issues

Common Problems

- Build fails or behaves differently in TeamCity but not locally
- Build is slow under TeamCity
- Agent does not upgrade with "Agent has unregistered (will upgrade)"
- Artifacts of a build are not cleaned
- "out of memory" error with internal (HSQLDB) database
- The transaction... log is full
- Slow download from TeamCity server

Build fails or behaves differently in TeamCity but not locally

If the build fails or otherwise misbehave in TeamCity but you believe it should not, please:

- check that the build runs fine on the same machine as TeamCity agent and under the same user that the agent is running, with the same environment variables and the same working directory.
- if TeamCity build agent is installed as a Windows service, try running TeamCity agent via console. 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 does not relate to TeamCity directly. Also, you can setup TeamCity agent to be run from console all the time (e.g. configure automatic user logon and run the agent on user logon).

The following approach can be used for this test:
For a configured build in TeamCity which is failing:

- run the build 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 TeamCity agent runs under
- stop the agent
- in a command 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)
- if the build fails - investigate the reason as the issue is probably not TeamCity specific 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 `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 Run custom build dialog
- when finished, enable the agent

You can also find the command that TeamCity used to launch the build in `logs\teamcity-agent.log` agent log file.

If the build succeeds from console but still fails in TeamCity, please create a new issue in our tracker detailing the case. Please attach the build log, all agent logs covering the build, the command you used in console to run the build and the full console output of the build.

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 console on the same machine as detailed above to see if there is any difference between the build from console and build in TeamCity.

If the slowness is spread over all the operations, 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), 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 build log, full agent logs and other data collected.

Agent does not upgrade with "Agent has unregistered (will upgrade)"

This means the agent is downloading the new agent version and should become connected shortly. The time necessary to upgrade mostly depends on the network link between the agent and the server. After downloading the files upgrade can take 1-3 minutes.

If the agent stays in the state for more then 10 minutes and you have fast network connection between the agent and the server, please look into agent logs.
If you cannot find the cause of the delayed agent upgrade in the logs, contact us and attach the agent logs and the server logs.

Artifacts of a build are not cleaned

If you encounter a case when artifacts of a build are not cleaned, 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 using build configuration, check whether "Prevent dependency artifacts clean-up" is turned ON (this is default value). If it does, then the build's artifacts are not cleaned because of the setting.
  Read more on cleanup settings.

"out of memory" error with internal (HSQLDB) database

If while TeamCity server statup you encounter errors like:

- `"error in script file line: ... out of memory"`
- `"java.sql.SQLException: out of memory"`

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 corruption based on notes on HSQLDB documentation.

A way to attempt a manual database restore:

- stop the TeamCity server
- remove <TeamCity Data Directory>/system/buildserver.data file and replace it with zero-size file of the same name. Please backup the removed file beforehand.
- start the TeamCity server

However, if the database does not recover automatically, chances that it can be fixed manually are minimal.

Unfortunately, internal (HSQL) database is not stable enough for production use and we highly recommend using external database for TeamCity non-evaluation usage.
If you encountered database corruption, you can restore 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 during cleanup, when the external database is MS SQL or Sybase. 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.

Slow download from TeamCity server

If you experience slow seeps when downloading artifacts from TeamCity, try checking the speed on the server machine, downloading from localhost.
If the seeps is OK for localhost, the issue can be in the network configuration or OS/hardware settings when combined with TeamCity(Tomcat) settings.

You can try the following approach:
in <TeamCity home>\conf\server.xml, change default part (port number may be different)

```
<Connector port="8111" protocol="HTTP/1.1"
               connectionTimeout="60000"
               redirectPort="8543"
               useBodyEncodingForURI="true"
    />
```

to:

```
<Connector port="8111" protocol="org.apache.coyote.http11.Http11NioProtocol"
                connectionTimeout="60000"
                redirectPort="8543"
                useBodyEncodingForURI="true"
                socket.txBufSize="64000"
                socket.rxBufSize="64000"
                tcpNoDelay="1"
                                />
```

and restart TeamCity server.

If this helps, please let us know via email.
If it does not, please revert the settings.
To investigate the case you might need to find an administrator with appropriate network-related issues investigation skills to look into the case.

Known Issues

This page contains a list of workarounds for known issues in TeamCity.

Agent running as Windows Service Limitations

When TeamCity build agent is installed as a Windows service, service limitations can affect the builds that the agent is running. At the same time the build can be sensitive to the user it runs under.

Common indicators of the issue are various "Permission denied" or "Access denied" errors during the build process.

The most common service-related issues are:

- user the service runs under. The user should have enough permissions to perform the build;
- access to network shares and mapped drives. Services have Windows-imposed restrictions to access the resources;
- interaction with desktop, headless mode, etc.

Solution

If you run TeamCity agent service under "System", try specifying usual user account with necessary permissions granted and restart the service.

If that does not help, try running TeamCity agent via console.

For more investigation steps, see Common Problems.

Back to top

Clearing Browser Cahes

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 antiviruses) can significantly slow down build agent operations like sources checkout, artifact publishing or even build running.

Certain antivirus software like Kaspersky Internet Security can result in Java process crashes or other misbehavior like inability to access files. Please try running with antivirus software uninstalled before reporting the issue to JetBrains. e.g. see the issue.
ESET antivirus can also slow down Ant/IntelliJ IDEA project builds a great deal (slowing down TCP connections to localhost on agent).

Please disable various indexing services. e.g. there might be problems with Windows Indexing Service. See issue for more details. Windows System Restore Feature might also need disabling.

Please also do not install software with background indexing like WinCVS, TortoiseCVS, TortoiseSVN and other Tortoise* products. This applies to server and also to agents if you use agent-side checkout.

Skype software is known to:
1. use port 80 on the system so you might not be able to use TeamCity server using default 80 port.
2. corrupt layout of pages displayed in Internet Explorer. Internet Explorer Skype plugin is to blame. (TW-13052).

Back to top

Subversion issues

Subversion Repositories With NTLM Authorization

If TeamCity has problems connecting to SVN repository, while using NTLM Authentication, add an option to the server JVM options and to the agent options (if you use agent-side checkout):
`-Dsvnkit.http.methods=Basic,NTLM`
See a related forum post on this issue.

TeamCity default setting for **svnkit.http.methods** property is **Digest,Basic,Negotiate,NTLM**, so this problem should not arise regularly. However, this can also mean that more secure NTLM authentication is not used by default. To enforce using NTLM, add JVM option `-Dsvnkit.http.methods=NTLM,Negotiate,Digest,Basic`.

> ⚠ Enforced NTLM authentication with TeamCity may be unstable. For instance, you may notice authentication errors after each 5 days of running. One of the ways to workaround this, according to our users, is to use local machine account between TeamCity and VisualSVN server (instead of domain account)

See also a related issue on HTTP Negotiate configuration.

Very slow checkout on agent for Subversion in TeamCity 7.0.x

There are two known cases:

1. Using 1.7 working copy for checkout - please upgrade to TeamCity 7.0.2, where the problem should be fixed

2. Using pre-1.7 working copy - please start your build agent under Java 1.7. More details in this issue

svn: E175002: Received fatal alert: bad_record_mac

Please add system property -Dsvnkit.http.sslProtocols=SSLv3 on the build server (see Configuring TeamCity Server Startup Properties).
If you use checkout on agent, add this property on build agent as well.

Subversion-related JVM Crashes

If JVM crashes while executing SVN-related code (e.g. under org.tmatesoft.svn package), you can try to disable it by either:

- Passing `-Dsvnkit.useJNA=false` JVM option to the crashing process (server or agent), or
- Making NTLM support less prioritative by passing `-Dsvnkit.http.methods=Basic,Digest,NTLM` JVM option.

Anyway, upgrading the JVM used to the latest available version is recommended.

Back to top

NUnit 2.4.6 Performance

Due to an issue in NUnit 2.4.6, its performance may be slower than NUnit 2.4.1. For additional information, please refer to the corresponding issue in our issue tracker: TW-4709

Back to top

StarTeam Performance

Using StarTeam SDK 9.0 instead of StarTeam SDK 9.3 on the TeamCity server can significantly improve VCS performance when there is a slow connection between TeamCity and StarTeam servers.

Back to top

Perforce 2009.2 Performance on Windows

If you run Perforce 2009.2 on Windows you may experience significant slow down. This is an issue with P4 server running on Windows. Please refer to corresponding section in Perforce documentation.

Wrong times for build scheduled triggering (Timezone issues)

Please make sure you use the latest JDK available for your platform (e.g. Oracle JDK download).

There were fixes in JDK 1.5 and 1.6 to address various wrong timezone reporting issues.

Back to top

Upgrading IntelliJ IDEA May Affect Active Pre-Tested Commits

Before you upgrade to IntelliJ IDEA X (or other IntelliJ X platform products) please make sure you do not have active pre-tested commits, otherwise they will not be able to be committed after upgrade.
This is only relevant if you use directory-based IDEA project (project files are stored under `.idea` directory).

Other Java Applications Running on the Same Server

If other web applications are available via the same hostname, a session cookie conflict can occur. This usually is visible via random user logouts or losing session-level data. (e.g. TW-12654). To resolve this, you can use different host names when accessing the applications.

Back to top

Reporting Issues

If you experience problems running TeamCity and believe it's related to the software, please contact us with detailed description of the issue.

To fix a problem, we may need a wide range of information about your system as well as various logs. The section below explains how to collect such information for different issues.

In this section:

Slowness, Hangings and Low Performance

If TeamCity is running slower then 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 what process is slow
If you experience slow TeamCity web UI response, checking for changes process, server-side sources checkout or other slow server activity, your target should be machine where TeamCity server is installed.
If the issue is related only to a single build, you will need to also investigate TeamCity agent machine which is running the build.
Please investigate the system resources (CPU, memory, IO) load. If there is a high load, determine the process to blame. If it is not TeamCity-related process, that might need addressing outside of the TeamCity scope.

If it is TeamCity server which is loading CPU/IO or there is no substantial CPU/IO load and all runs just fine except for TeamCity, then this is to be investigated further.

If you have a substantial TeamCity installation, please check you have appropriate memory settings as a first step.

Collect Data
Take several thread dumps of the slow process (see below for thread dump taking approaches). 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 the detailed description of the issue to us accompanied with the thread dumps and full server (or agent) logs covering the issue.
Unless not possible for some reasons, the preferred way is to file an issue into our issue tracker . Please include all the relevant details of investigation, including CPU/IO load information, what specifically is slow and what is not, etc.

Server thread dump

You can take a thread dump of the TeamCity server right from the web UI (if the hanging is local and you can still open 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 `<TeamCity home>/logs` directory (where you can later download the files from "Server Logs"). A small hint: if web UI is not responsive try direct URL substituting your server URL.

If the server UI is not usable please use approaches described below.
Please note that if you can take a thread dump without using TeamCity UI, that can be preferable since the thread dump taken from UI can lack some diagnostics information.

Agent thread dump

You can take a thread dump of the TeamCity agent process right from the web UI: Use `Dump threads on agent` action on the Agent page.

Please note that if you can take a thread dump by other means, that can be preferable since the thread dump taken from UI can lack some diagnostics information
If you will take agent thread dump manually, please note that TeamCity agent consists of two `java` processes: launcher and agent itself. Agent is run by the launcher process. You will usually be interested in the agent (more nested one) and not the launcher process.

Thread dump taking approaches

These can help if the approach to take the thread dump for the server from Administration / Diagnostics page is not applicable.

To take a thread dump:
Under Windows
You have several options:

- if you need server thread dump and the server is run from console, press **Ctrl+Break** in the console window (this will not work for agent, since it's console belongs to launcher process).
- if JDK 1.6 is used, use `jstack <pid_of_java_process>` command (jstack is located in the "bin" directory of JDK installation). If it does not work, ensure you use the tools from the same version of JDK as is used to run the process you take the dump from (same x86/x64 and same version).
- you can also use TeamCity-bundled thread dump tool (can be found in agent's plugins). Run the command:

```
<TeamCity agent>\plugins\stacktracesPlugin\bin\x86\JetBrains.TeamCity.Injector.exe
<pid_of_java_process>
```

If the hanging process is run as a service, taking the thread dump is more complicated and under some environments does not have a reliable approach. Recommended approach is to run the agent or server process from a console and use `Ctrl+Break` approach above.

Alternatively, you can try to run a thread dumping tool using Microsoft PsExec.
Use the following command line to get the dump:

```
psexec -u <user> -p <password> <full path to the util> <process pid> stacktrace <output file>
```

where:
<user> and **<password>** – correspond to the same account that was used to run the hanging process as a service. If service uses "System", use "**-s**" switch instead of `-u` and `-p`.
<full path to the util> – full path to the thread dumping util
<process pid> – the process id of the hanging process
<output file> – a full path to a file to save output to (should be accessible for the specified user).

Under Linux

- run `jstack <pid_of_java_process>` or `kill -3 <pid_of_java_process>`. In the latter case output will appear in `<TeamCity server home>/logs/catalina.out` or `<TeamCity agent home>/logs/error.log`.

You can also use third-party GUI tool: AdaptJ StackTrace Utility.
It supports Windows, Linux or Mac OS X. Choose "Launch" on the page (if you have Java installed). When the application is started, select the process id via "Process > Select", then dump its thread dump with "Process > Thread Dump".

See also Server Performance section below.

Database-related slowdowns

When the server is slow it makes sense to check if the problem is caused by database operations.
It is recommended to use database-specific tools.

You can also use `debug-sql` server logging preset. Upon enabling, all the queries which take longer 1 second will be logged into teamcity-sql.log file. The time can be changed by setting "teamcity.sqlLog.slowQuery.threshold" internal property. The value should be set in milliseconds and is 1000 by default.

MySQL

Along with server thread dump, please attach output of "show processlist;" SQL command executed in MySQL console. Much like the thread dumps it makes sense to execute the command, if slowness occurred several times 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 (OutOfMemory errors), please do the following:

- Determine what process encounters the error (the actual building process, the TeamCity server, or the TeamCity agent)
- If server is to blame, please check you have increased memory settings from default ones for using the server in production (see the section).
- If you use x64 JVM, please consider using 32 bit JVM, as it will require less memory (instructions for server).
- Try to increase the memory for the process via `'-Xmx'` JVM option, like **-Xmx1200m**. If the error message is "java.lang.OutOfMemoryError: PermGen space", increase the value in `-XX:MaxPermSize=270m` JVM option.
  The option needs to be passed to the process with problems:
  - if it is the building process itself, use "JVM Command Line Parameters" settings in the build runner. e.g. Inspections builds may specifically need to increase the parameter;
  - refer to the corresponding documentation for TeamCity server or agent. See also Installing and Configuring the TeamCity Server#memory;
- If increasing memory size does not help, please take several server thread dumps as described above, get the memory dump, archive it and send it to us for further analysis. Please reduce the Xmx setting to normal 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 JVM option (works for JDK 1.5.0_07+): `-XX:+HeapDumpOnOutOfMemoryError`. When OOM error occurs next time, `java_xxx.hprof` file will be created in the process startup directory (`<TeamCity home>/bin` or `<TeamCity Agent home>/bin`);
  - you can also take memory dump manually when the memory usage is at its peak. Go to the **Administration** | **Server Administration** | **Diagnostics** page of your TeamCity web UI and click **Dump Memory Snapshot**.
  - another approach to take the memory dump manually is to run TeamCity server with JDK 1.6+ and use `jmap` standard JVM util. e.g. `jmap -dump:file=<file_on_disk_to_save_dump_into>.hprof <pid_of_your_TeamCity_server_process>`

See how to change JVM options for the server and for agents.

Back to top

"Too many open files" Error

1. Determine what computer it occurs on
2. Determine the process locking the files (on Linux use `lsof`, on Windows you can use handle or TCPView for listing sockets) and the files list
3. If the number is not large, check the OS and process limits on the file handles (on Linux use `ulimit`) 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.

If the number of files is large and is looking suspicions and the locking process is a TeamCity one (TeamCity agent or server with no other web applications running), grab the list of open handles several times with several minutes interval and send the result to us for investigation together with relevant details.

Please note that you wil most probably need to reboot the machine with the error after the investigation to restore normal functioning of the applications.

Logging Events

TeamCity server and agent create logs that can be used to investigate issues.

> **How to enable DEBUG logging on server?**
> Before reproducing the problem it makes sense to enable 'DEBUG' log level for TeamCity classes.
> On the server side, go to the **Administration** | **Server Administration** | **Diagnostics** page and select debug logging level. 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

In general, to debug VCS problems we need information for jetbrains.buildServer.VCS Log4j category.

TeamCity server performs checking for changes and if server-side checkout is used also the checkout.

TeamCity agent performs the checkout if agent-side checkout is used.

For the server, you can switch logging preset to "debug-vcs" in administration web UI and then retrieve `logs/teamcity-vcs.log` log file.

For agent and the server you can change the Log4j configuration manually in <TeamCity home>\conf\teamcity-server-log4j.xml or <BuildAgent home>\conf\teamcity-agent-log4j.xml files to have fragment:

```
<category name="jetbrains.buildServer.VCS" additivity="false">
    <appender-ref ref="ROLL.VCS"/>
    <appender-ref ref="CONSOLE-ERROR"/>
    <priority value="DEBUG"/>
</category>

<category name="jetbrains.buildServer.buildTriggers.vcs" additivity="false">
    <appender-ref ref="ROLL.VCS"/>
    <appender-ref ref="CONSOLE-ERROR"/>
    <priority value="DEBUG"/>
</category>
```

If there are separate logging options for specific version controls, they are described below.

Subversion Integration Debug Logging

For server-side logging use "debug-vcs" logging preset in administration web UI and then retrieve `logs/teamcity-svn.log` log file.

Alternative manual approach that is also necessary for agent-side logging:

First, please enable generic VCS debug logging, as described above.

Uncomment SVN-related parts (`SVN.LOG` appender and `javasvn.output` category) of Log4j configuration file on server and on agent (if agent-side checkout is used). The log will be saved to the `logs/teamcity-svn.log` file. Generic VCS log should be also taken from `logs/teamcity-vcs.log`

ClearCase

Uncomment Clearcase-related lines in the `<TeamCity home>\conf\teamcity-server-log4j.xml` file.
The log will be saved to `logs\teamcity-clearcase.log` directory.

Patch Application Problems

In case server-side checkout is used, the "patch" that is passed from server to the agent can be retrieved by:

- add property `system.agent.save.patch=true` to the build configuration.
- trigger the build.

Build log and agent log will contain the line "Patch is saved to file ${file.name}"
Get the file and provide it with the problem description.

Back to top

Remote Run Problems

The changes that are sent form the IDE to the server on a remote run can be retrieved from server's `.BuildServer\system\changes` directory. Locate the `<change_number>.changes` file that corresponds to your change (you can pick the latest number available or deduce the from the URL of the change form the web UI).
The file contains the patch in the binary form. Please provide it with the problem description.

Back to top

Server Performance

If you experience degraded server performance and TeamCity server process is producing large CPU load, please take the CPU profiling snapshot and send it to us accompanied with the detailed description of what you were doing and what is your system setup.
You can take the CPU profiling and memory snapshots by installing the server profiling plugin and following the instructions provided on the plugin page.

Here are some hints to get the best results from CPU profiling:

- after server startup 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 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 the snapshot
- archive the snapshot and send it to us including description of the actions that cause CPU load

Back to top

Logging in IntelliJ IDEA/Platform-based IDEs

To enable debug logging for IntelliJ Platform based IDE plugin you can manually change the Log4j configuration in `<IDE home>\bin\log.xml` file to have fragment:

```
<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 please restart the IDE. TeamCity plugin debug logs will appear in `<IDE data directory>/system/log` directory (`idea-teamcity*` files).

Open in IDE functionality logging

(Applicable to IntelliJ IDEA and Eclipse)
Add the following JVM option before starting IDE:
`-Dteamcity.activation.debug=true`
Logging related to open in IDE functionality will appear in the IDE *console*.

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. Name of each property corresponds to the plugin module and 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/perfomance/vcs = true
jetbrains.teamcity.core/perfomance/teamcity = true
```

Read more about Eclipse Debug mode Gathering Information About Your Plug-in and built-in Eclipse help.

Back to top

TeamCity Visual Studio Addin issues

TeamCity Addin logging

To capture logs from TeamCity Visual Studio Addin please do the following:

1. Close all instances of Microsoft Visual Studio.
2. Open `<username-profile-folder>\Local Settings\Temp\JetLogs` folder.
3. Delete all of the files in this folder.
4. Restart Microsoft Visual Studio.
5. Open a solution.
6. Try to log in to TeamCity.
7. Close Microsoft Visual Studio.
8. Navigate back to the `<username-profile-folder>\Local Settings\Temp\JetLogs` folder. All of the files that were created are logs.

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

JVM Crashes

On a rare occasion of TeamCity server or agent process terminating unexpectedly with no reason, it can happen that this is caused by Java runtime crash.
If this happens, Oracle JVM creates a file named `hs_err_pid*.log` in the working directory of the process. Working directory can be `<TeamCity server or agent home>\bin` or other like `C:\Windows\SysWOW64`. You can search the disk for the recent files with "hs_err_pid" in the name.
See also related description.

Please send this file to us for investigation and consider updating JVM for the server (or for agents) to the latest version available.

Build Log Issues

While investigating issues related to build log we might need raw binary build log as stored by TeamCity.
It can be downloaded via Web UI from the Build Log build's tab: select "Verbose" log detail and use "raw messages file" link at the top-right.

Sending Information to the Developers

Files under 5Mb in size can be attached right into the tracker issue (if you do not want the attachments to be publicly accessible, limit viewing the attachment to "teamcity-developers" user group only).
You can also send small files (up to 2 Mb) via email: teamcity-feedback@jetbrains.com Please do not forget to mention your TeamCity version and environment.

If the file is over 5 Mb, you can upload the archived files to ftp://ftp.intellij.net/.uploads and let us know the exact file name. If you receive permission denied error on 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>".

Back to top

Applying Patches

Microsoft Visual Source Safe Integration

To apply patch for `vss-native.exe`:

1. Shut down TeamCity server
2. Open `<TeamCity Home>/webapps/root/WEB-INF/plugins/vss/` or `<TeamCity Home>/webapps/root/WEB-INF/lib/` folder
3. Back up `vss-support.jar` file
4. Inside `vss-support.jar` file, replace `/bin/vss-native.exe` with the new one
5. Start the server

To apply full VSS plugin patch:

1. Shut down TeamCity server
2. Open `<TeamCity Home>/webapps/root/WEB-INF/plugins/vss/` or `<TeamCity Home>/webapps/root/WEB-INF/lib/`
3. Back up `vss-support.jar`
4. Replace `vss-support.jar` with the new one
5. Start the server

Capturing Logs From VSS-native

Each time TeamCity starts, it creates a new instance of the `vss-native.exe` file and places it to the `<TeamCity home>/temp` folder. The name of the copy is generated automatically and uses the following template: `TC-VSS-NATIVE-<some digits>.exe`

To manually enable detailed logging (for debugging purposes) for VSS Native:

1. Copy the `<TeamCity Home>/temp/TC-VSS-NATIVE-<some digits>.exe` file to any folder.
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.

Enabling Detailed Logging for .NET Runners

To enable debug logging for investigating process launch issues for .Net-related runners enable debugging as described below.
The detailed logging will then be printed into build log for the following builds.

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, please do not forget to roll back the change.
The change in `teamcity-log4net.xml` will be removed on build agent autoupgrade.

Alternative approach that might also work:
set 'teamcity.agent.dotnet.debug' configuration parameter to 'true' in a build configuration or on agent.

Visual C Build Issues

If you experience any problems running Visual C++ build on a build agent, you can try to workaround these issues with the following steps, sequentially:

⚠ Any of these steps may solve your issue. Please feel free to leave feedback of you experience.

- Make sure you do not use mapped network drives.
- Make sure build user have enough right to access necessary network paths
- Log on to the build agent machine under the same user as for build and try running the following command:

```
msbuild.exe <path to solution.sln> /p:Configuration:Release /t:Rebuild
```

- Build Agent service runs under the user with local administrative privileges
- Make sure Microsoft Visual Studio is installed on the build agent
- You have to start Visual Studio 2005 or Visual Studio 2008 under build user once
http://www.jetbrains.net/devnet/message/5233781#5233781
- If **Error spawning cmd.exe** appears, you should put the following lines exactly into the list in **Tools -> Options -> Projects and Solutions -> VC++ Directories**:

```
--$(SystemRoot)\System32
--$(SystemRoot)
--$(SystemRoot)\System32\wbem
```

http://www.jetbrains.net/devnet/message/5217957#5217957

- You need to add all environment variables from `...\Microsoft Visual Studio 9.0\VC\vcvarsall.bat` to environment or to `buildAgent.properties` file
- Try using **devenv.exe** with Command Line Runner instead of Visual Studio(sln) build runner
- Ensure all paths to sources do not contain spaces
- Set `VCBuildUserEnvironment=true` in runner properties
- Specify 'VCBuildAdditionalOptions' property with value '/useenv' in the build configuration settings to instruct msbuild to add '/useenv' commandline argument for spawned vcbuild processes.

See also:

Administrator's Guide: .NET Testing Frameworks Support | NUnit Support