Indore 10 EAP2 (build 41463) Release Notes

See also Indore 10 EAP1 (build 40941) Release Notes

- DSL for TeamCity project configuration
- Smart checking for changes interval
- Project Configuration Export
- VCS hosting integrations
- Support for Perforce Jobs
- Refreshed icons in web UI
- Agent limit in pools
- Cloud support
- REST API enhancements
- Bundled Tools updates
- Other Improvements

DSL for TeamCity project configuration

Up until this EAP there were three options to configure project settings in TeamCity: the straightforward way was via the web UI, and the other two were using the REST API or via XML files if the versioned settings feature is enabled.

In this EAP we introduce a new way - the ability to define settings programmatically using DSL based on the Kotlin language.

Kotlin-based DSL can be seen as evolution of the versioned settings: now you can define your configuration in the Kotlin language without the need to use XML files.

Since Kotlin is statically typed, you automatically receive the auto-completion feature in IntelliJ IDEA which makes discovery of available API options a much simpler task.

To get started with Kotlin DSL, navigate to Versioned settings menu option for your project, enable Versioned settings for your project, select kotlin as the format and commit the settings to your version control.
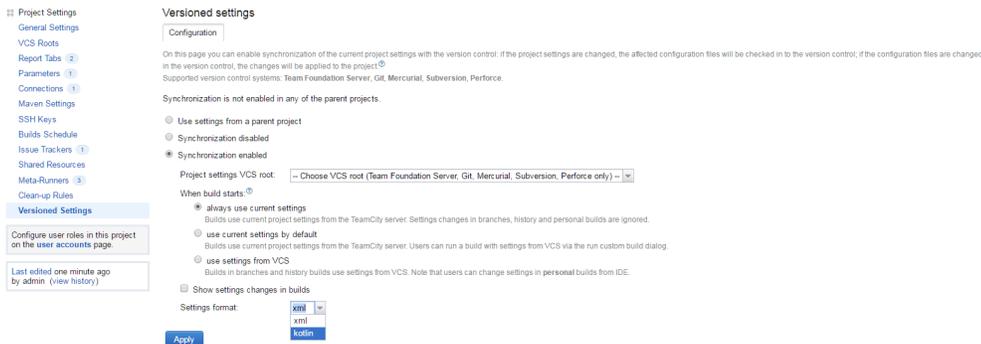
After you commit settings to the VCS, TeamCity will generate necessary Kotlin files for this project and check them in to the specified repository under the `.teamcity` directory. If this repository already had project settings in the XML format, they will be preserved.

TeamCity also adds a pom.xml file under the .teamcity directory. You can open this POM in your IntelliJ IDEA and start working with Kotlin DSL right away. All necessary dependencies will be resolved automatically.

Some important facts:

- each time when you make a commit into `.teamcity`, TeamCity will execute Kotlin DSL files. Since internally TeamCity still operates with XML, the executed DSL files will produce a bunch of XML files. These XML files then will be applied to the existing project effectively changing its configuration. In case of any problems (compilation failures, runtime errors, etc), new changes will not be applied, and the current project settings will be preserved on the server.
- once the project is switched to Kotlin, editing project settings via the web UI will become disabled (except a few pages: Versioned settings, Maven Settings, SSH keys and Meta-runners), because currently there is no way to propagate changes made via the web UI to Kotlin DSL files.
- The Kotlin script is executed on the server, and since this is a potentially dangerous operation, the script is executed in

the sandbox. It cannot modify the file system except the place where it is executed, it cannot run other programs, cannot use reflection, and so on.

- at this point Kotlin DSL is experimental, and the provided API may change significantly in future versions.

Example of a Project defined in Kotlin DSL:

```
object Project : KProject({
    uuid = "my_project_id" // uuid should be some constant, never changing string; it is important for
preserving history, new id means new entity with new history
    extId = "ExampleOfDSL"
    parentId = "_Root" // id of the parent project
    name = "Example of DSL"

    val vcsRoot = vcsRoot {
        uuid = "my_vcs_root_id"
        extId = "ExampleOfDSL_VcsRoot"
        type = "jetbrains.git" // other available types: svn, perforce, tfs, mercurial, starteam, cvs,
vault-vcs
        name = "Example of DSL VCS Root"
        param("url", "<url to my git repository>")
    }

    buildType {
        uuid = "my_build_type_id"
        extId = "ExampleOfDSL_Build"
        name = "Build"

        vcs {
            entry(vcsRoot) // thanks to Kotlin, here we can have static reference to project VCS root
        }

        steps {
            step {
                type = "Maven2"
                param("goals", "clean test")
            }
        }

        options {
            buildNumberPattern = "%build.counter%"
        }

        requirements {
            contains("teamcity.agent.jvm.os.name", "Linux")
        }
    }
})
```

Since this is essentially Kotlin code, you can do whatever you like, add conditions, loops, methods, classes, etc. Note that for your convenience the methods buildType(), vcsRoot() and others not only accept some instance, they also return it as a result. In the example above you can see how the vcsRoot instance can be reused in the build configuration.

All TeamCity entities - project, build configuration, VCS root and template have so called `uuid`. The uuid is an identifier which can be used to uniquely distinguish this specific entity from others. If at some point the entity receives a new uuid, it is considered a new entity by TeamCity. For instance, if the build configuration uuid changes, the builds history of this build configuration will become empty (if you change it back before cleanup, the history will be restored). The same rule applies to a

project - if its uuid changes, the project will loose its investigations and muted tests. We suggest selecting some naming scheme for uuids in your project and never change them unless you really want to make TeamCity think that these are new entities.

There is also the extId field which is mandatory. The extId is the same as build configuration (or template) ID / vcs root ID, project ID in the web UI. It can be changed at any time. But be aware that some settings use it, for instance the extId can appear in dep. parameter references. If you change the extId, you should find all its occurrences in the current project and change them too.

Smart checking for changes interval

Traditionally TeamCity uses polling for detecting changes in VCS repositories. Polling is a highly reliable approach suitable in the majority of cases. Even if the TeamCity server was stopped for a while, with polling it can easily pickup all the changes made in repositories on the next startup.

But polling has one downside which becomes more and more important, as TeamCity installation grows. If there are many different VCS repositories configured in TeamCity, polling can impose significant load on both TeamCity server and VCS repository servers.

The alternative approach is to use the push model: various post commit hooks and web hooks initiated on the repository side. This approach is more scalable, but it cannot be used alone: if the TeamCity server is stopped, obviously all push notifications will be lost.

To get the best of both worlds we decided to implement a combined approach. Starting with this EAP, if a commit hook initiates the process of checking for changes for some VCS root in TeamCity, TeamCity will automatically increase checking for changes interval for this VCS root, assuming that this commit hook will now come to TeamCity on a regular basis. But, if so happens that TeamCity detects a change in this VCS root during regular polling, then the checking for changes interval will be reset to the initial value specified by the user when the VCS root was created. This is done for the case when a commit hook stopped working for some reason. If the TeamCity server was restarted, it will switch to polling for all of the VCS roots, till commit hooks start informing it about new commits.

Please refer to our documentation on commit hooks configuration in various VCS repositories.

Project Configuration Export

It is now possible to export configuration files for a project with its children as a zip archive to move it to a different TeamCity server. The Project Settings | Settings Export page allows exporting the config files for a project and its subprojects, as well as external dependencies, i.e. build configurations used in snapshot dependencies, templates used as well as vcs roots and all main settings (ssh keys, issue trackers, oauth connections etc...) defined in the parent project.

The settings archive also contains a `report.log` file detailing the reasons for exporting external entities.

VCS hosting integrations

- the Commit status publisher build feature now supports GitLab thanks to an external contribution.
- Bitbucket cloud issue tracker is now supported. The integration can be set up separately, or as a part of TeamCity integration with Bitbucket source code hosting service making it easy to connect TeamCity to Bitbucket issues.
- Git LFS is now supported for checkout on agent

Support for Perforce Jobs

If you use Perforce jobs to label your commits, the changes associated with jobs are now marked with a "wrench" icon 🔧 in the TeamCity UI. Navigating to the icon opens a pop-up with the job information:

Refreshed icons in web UI

Web UI icons have got a refresh in this EAP build. Build status icons, icons for projects and build configurations, icons for investigations have been changed to a more modern look. Some light facelifting have been done to parts of web UI.

Agent limit in pools

It is now possible to set a maximum number of agents in a pool: if the maximum number of agents is reached, TeamCity will not allow adding any new agents to this pool. This includes moving agents from other pools and automatic authorization of cloud agents. New cloud agents will not start if the target pool is full. The limit is not applicable for the Default Pool.

The feature can be useful for those who maintain a pool of agents per project and want to prevent projects from using all of the available agent licenses.

Cloud support

- It is now possible to run a custom script on the launch of an Amazon EC2 instance (applicable to instances cloned from AMI's only). The Amazon website details the script format for Linux and Windows.
- Unique hostnames for Windows vSphere cloud agents on can be specified now: when adding an image, choose a customization spec in the corresponding field. The option is available for Linux VMs as well.

REST API enhancements

With the latest REST API version it is now possible to:

- list the agents compatible with a build configuration and filter agents by compatible build configurations. This does not include non-started cloud agents (images), which are not yet exposed via REST API.
- get the projects and build configurations as well as their order on the Overview page as configured by the specified user
- disable/enable artifact dependencies and agent requirements
- get the build's test occurrences in the order they were run in the build
- get all runs (the number of test invocations) for a test
- get test mutes affecting a specific build configuration
- list users by email, group and user property
- indication of the build configuration settings if it was inherited from a template or a project

Bundled Tools updates

- the bundled Ant is updated to 1.9.7
- the bundled dotCover is updated to 2016.1

Other Improvements

- in case of exit code problem, TeamCity now tries to locate relevant part of a build log and show it right on build results page
- performance of the project and build configuration settings editing has been greatly improved especially for large installations with thousands of projects

- you can now redefine inherited artifact dependencies in build configurations, the same as agent requirements and other settings
- a new option of the Free Disk Space build feature allows you to fail a build if sufficient disk space cannot be freed for the build
- it is possible to make a parameter read-only via parameter specification; such a parameter cannot be changed via custom build dialog
- fixed issues