QuickBuild 1.0.1 User's Guide

Table of Contents

1. Introduction 1
    - Background 1
    - Basic concepts 1
        - Configuration 1
        - Build 1
        - Repository 1
        - Builder 2
        - Notifier 2
        - Step 2
        - Login mappings 3
        - Schedule 3
        - Triggering instance 3
        - Build queue 3
        - Configuration's working directory 3
        - Configuration's checkouts directory 3
        - Configuration's publish directory 3
        - Build's publish directory 4
        - Build's artifacts directory 4
        - Build's JUNIT html report directory 4
        - Build's Clover html report directory 4
2. User interfaces 5
    - Dashboard 5
    - Configurations 7
    - Build queues 9
    - Find builds 10
    - Administration 11
        - Manage groups 12
        - Manage users 13
3. Configure repositories 15
    - Configure Base Clearcase 15
    - Configure Clearcase UCM repository 16
    - Configure CVS repository 17
    - Configure File system repository 18
    - Configure Perforce repository 18
    - Configure Subversion repository 19
    - Configure Visual Sourcesafe repository 22
    - Configure StarTeam repository 25
    - Configure Accurev repository 27
    - Configure QuickBuild repository 27
4. OGNL expressions 29
5. Velocity templates 32
6. Security 34
    - User authentication 34
    - User authorization 34
        - Set queue permissions 34
        - Set configuration permissions 34
7. Remote and parallel build support 36
    - Mechanism 36
    - Deadlock prevention 36
8. Data management 37
    - Backup and restore database 37

    Data migration between different databases 37
    Migrate data from QuickBuild PREVIEW2 and PREVIEW3 37
    Migrate data from Luntbuild 1.2.x 37
9. REST API 39
10. Use cases 40
    General use cases 40
        Working with your first build 40
        Working with Maven 42
        Working with project dependencies 43
        Working with multiple branches 44
        Working with build promotion 44
        Sharing working directories 45
        Sharing build versions 46
        Using date and iteration as part of build version 46
        Managing major, minor, and iteration part of a version string 46
        Specifying label to build against 47
        Updating information of many projects 47
        Working with build queues 47
        Working with public configurations 48
    Remote and parallel use cases 48
        Building multi-platform products 48
        Working with parallel builds 49
        Performing automation/smoking tests on a machine other than build machine 50
    REST API use cases 52
        Set up real-time continuous integration build 52
        Trigger other builds after build of particular project 53

List of Tables

3.1. Date/Time format characters 23
6.1. Configuration permissions 35

Chapter 1. Introduction

Background

QuickBuild is a build automation and management tool working with *Apache Ant* [http://ant.apache.org], *NAnt*, *Maven* [http://maven.apache.org], or any other build tools with command line interface. With QuickBuild, daily builds and continuous integration builds can be set easily. Refer to the following articles for benefits of daily builds and continuous integration builds, if you are not familiar with them:

- *Continuous Integration* [http://www.martinfowler.com/articles/continuousIntegration.html]
- *Daily Builds Are Your Friend* [http://www.joelonsoftware.com/articles/fog0000000023.html]

Basic unit of work in QuickBuild is a *build*. Build execution is triggered by a schedule, or by the user manually. Typically, a build in QuickBuild performs following steps:

1. Checks out source code from the repository (CVS, Subversion for instance).
2. Runs an Ant/NAnt/Maven/Command build script in the source tree.
3. Labels the current source code based on the current build version.
4. Publishes the build log and other build artifacts.

Beyond the ability to automate your builds, QuickBuild put extra emphasis on build management, with which you can manage all kinds of builds (such as QA/release builds) simply and efficiently. Build configuration, monitoring, and access to the build artifacts are all done using an intuitive web interface. Your development and testing team will have a central area to access the build information.

Basic concepts

Configuration

Configuration holds all configuration information in order to generate builds. These information includes what to build, how to build, and when to build, etc. Configurations are organized in a tree structure. To refer to a configuration, name of all its ancestor configurations should be put together and connected with period, plus its own name, for example: *root.department1.project1.nightly*. Child configurations can inherit build settings from parent, and can selectively override some of them. This behavior is referred as *inherit and override* rule, which makes build management of many projects very easy.

Build

Build always associates with a version, and is generated by running of configuration. When a particular configuration is triggered (either by user or by scheduling system), the *build necessary condition* of configuration will be evaluated. If the result is true, the configuration will run, and a new build will be generated. Lately information about this build can be accessed online, including published artifacts, build logs, revision logs, etc.

Repository

Repository stands for a place stores files needed for generating builds. It provides "what to build" information for a configuration. It includes but not limited to version control systems, such as CVS, ClearCase, Subversion, etc. Also ordinary directories can also be treated as a repository (in case you store your source codes in a plain file system folder, instead of some version control system). Generally any place be able to store build artifacts (either source codes, or intermediate build artifacts such as libraries which may be used for further build process) can be treated as a repository. The repository object obeys the *inherit and override* rule, which means repositories defined in descendent configurations with the same name will override those defined in ancestor configuration.

Builder

Builder defines how to perform a build. After necessary files have been checked out from configured repositories, QuickBuild will call related builders to actually run the build. Currently, Ant builder, NAnt builder, Maven Builder and a command line builder are supported. However, any builder facilities with a command line interface should be able to work with QuickBuild. The builder object obeys *inherit and override* rule.

Notifier

Notifier defines how to notify users about information of recent build. Notification message can be customized through using of Velocity templates. Currently QuickBuild supports Email notifier, Jabber notifier, MSN messenger Notifier, and Google Talk notifier. Thanks for the *inherit and override* rule, you can define notifiers in high level configuration, and reuse them in descendents to minimize your configuration work.

Step

In order to make the build process more flexible, the step concept is introduced in QuickBuild. It defines the build process for a particular configuration, and you can control which parts of the build process can be executed simultaneously, and which parts should e executed serially. The step object obeys *inherit and override* rule. Currently, the following steps can be used:

- checkout

  This step helps to check out codes from specified repository.

- build

  This step helps to perform build with specified builder.

- label

  This step helps to create a label in related repositories in order to make a snapshot of files used for this build.

- publish

  This step helps to publish files so that they can be accessed online from QuickBuild web interface.

  ### Note

  This step is only needed for files that are not under control of QuickBuild. For example, in your build script, you do not copy generated artifacts to the directory denoted by *artifactsDir* property. In this case, this publish step will make them visible through the web interface by creating soft links to these files.

- label

  This step helps to create a label in related repositories in order to remember correct version of files used for this build.

- notify

  This step helps to notify proper persons about status of current build.

- serial composition

This is a composition step that helps to create a step comprised of other steps. Included child steps will be executed serially.

- parallel composition

  This is a composition step that helps to create a step comprised of other steps. Included child steps will be executed simultaneously.

Login mappings

Login mapping is used to map repository logins to QuickBuild users in order to do certain things, such as sending out failure build notifications to users who has checked in codes into repository recently, etc. You can define multiple login mappings, and associate different login mappings with different repositories. This is very useful when you have multiple repositories and login name for a particular user is not consistent through out these repositories. The login mapping object obeys *override and inherit* rule.

Schedule

Schedule defines "When to build" aspect of the configuration. By defining a schedule, builds can be triggered at specified time, or can be triggered periodically.

Triggering instance

Triggering of a configuration is called a triggering instance, which will be put into build queue when all working threads for that queue have been used up. Execution of a triggering instance does not necessarily generate a new build. This is determined by *build necessary condition*. However, if you set value of *build necessary condition* as *true* (by choosing *force build* from the build necessary condition drop down menu), new build will always be generated when this triggering instance get the chance to be executed.

Build queue

Every configuration associates with a build queue (if not set explicitly, parent configuration's build queue will be used), which is used to control maximum concurrent builds in this configuration (and its descendents if choose to inherit). When a configuration is triggered, some of the triggering will be queued if all configured working threads in the queue have been used up.

Configuration's working directory

Every configuration has an working directory which is used to hold configuration logs, checkouts directory, and child configuration working directories. Let's assume that you've installed QuickBuild under `/opt/quickbuild`, the default working directory for configuration *root.department1.project1.nightly* will be `/opt/quickbuild/working/root/department1/project1/nightly`. If you've changed working directory setting for a particular parent , working directories of child configurations will be derived in the same way starting from that directory.

Configuration's checkouts directory

Configuration's checkouts directory is located in a sub directory named by `checkouts` under configuration's working directory. It is used to hold checked out stuffs from various repositories.

Configuration's publish directory

Every configuration has an publish directory which is used to hold produced builds, and child configuration publish directories. Let's assume that you've installed QuickBuild under `/opt/quickbuild`, the default publish directory for configuration *root.department1.project1.nightly* will be

`/opt/quickbuild/publish/root/department1/project1/nightly`. If you've changed publish directory setting for a particular parent , publish directories of child configurations will be derived in the same way starting from that directory.

Build's publish directory

Build publish directory is used to hold artifacts directory, various logs and reports generated by a build. It is named by version number of the build, and is located under `builds` sub directory under configuration's publish directory. Continue with the example in last section, if build *project1-3.0.0* is produced by configuration *root.department1.project1.nightly*, its publish directory will be `/opt/quickbuild/publish/root/department1/project1/nightly/builds/project1-3.0.0`.

Build's artifacts directory

Build artifacts directory is used to hold published artifacts for a particular build, and is located under a sub directory named by `artifacts` of build's publish directory.

Build's JUNIT html report directory

Build JUNIT html report directory is used to hold JUNIT html reports generated by a particular build, and is located under a sub directory named by `junitHtmlReports` of build's publish directory.

Build's Clover html report directory

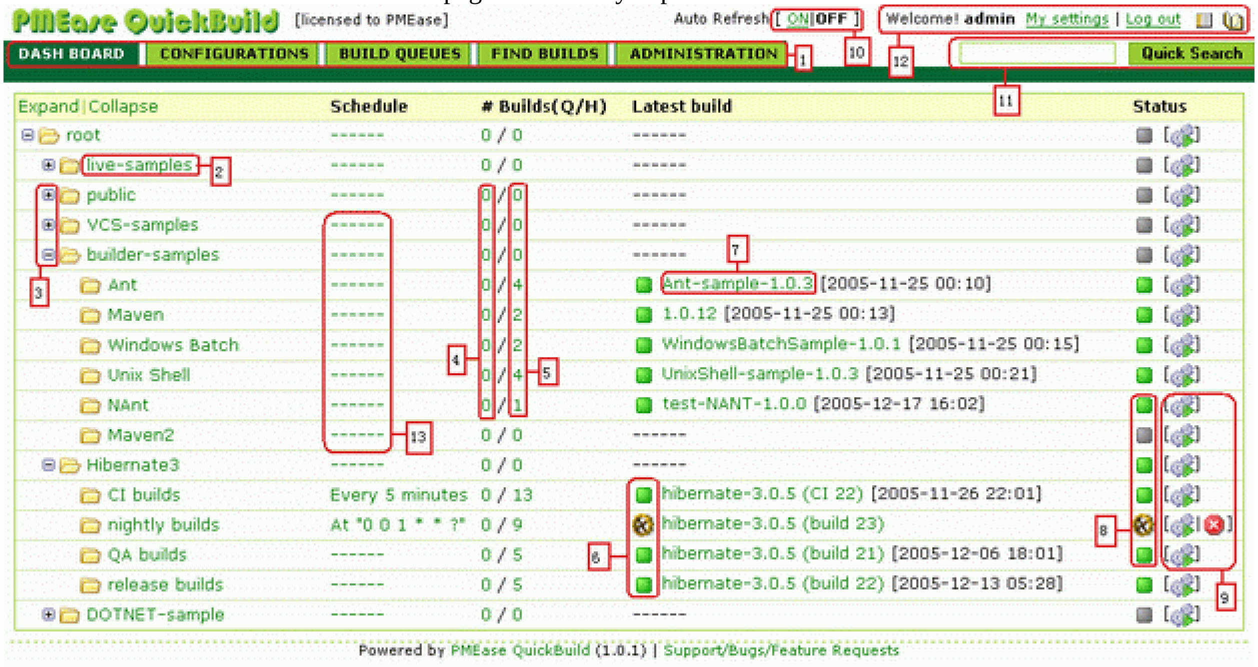
Build Clover html report directory is used to hold Clover html reports generated by a particular build, and is located under a sub directory named by `cloverHtmlReports` of build's publish directory.

Chapter 2. User interfaces

This chapter explains some typical user interfaces in QuickBuild application.

Dashboard

Dashboard is the first page of QuickBuild user interface. It gives an overview of all configurations and its build status. From this page, you can do most of your daily jobs, such as start/stop builds, monitor build status, access build logs, etc. Here is the screenshot for dashboard page followed by expanation of each number indicated function areas.



Expand | Collapse	Schedule	# Builds(Q/H)	Latest build	Status
⊟ root	------	0 / 0	------	⬛
⊞ live-samples	------	0 / 0	------	⬛
⊞ public	------	0 / 0	------	⬛
⊞ VCS-samples	------	0 / 0	------	⬛
⊟ builder-samples	------	0 / 0	------	⬛
Ant	------	0 / 4	🟩 Ant-sample-1.0.3 [2005-11-25 00:10]	🟩
Maven	------	0 / 2	🟩 1.0.12 [2005-11-25 00:13]	🟩
Windows Batch	------	0 / 2	🟩 WindowsBatchSample-1.0.1 [2005-11-25 00:15]	🟩
Unix Shell	------	0 / 4	🟩 UnixShell-sample-1.0.3 [2005-11-25 00:21]	🟩
NAnt	------	0 / 1	🟩 test-NANT-1.0.0 [2005-12-17 16:02]	🟩
Maven2	------	0 / 0	------	⬛
⊟ Hibernate3	------	0 / 0	------	🟩
CI builds	Every 5 minutes	0 / 13	🟩 hibernate-3.0.5 (CI 22) [2005-11-26 22:01]	🟩
nightly builds	At "0 0 1 * * ?"	0 / 9	hibernate-3.0.5 (build 23)	⊗
QA builds	------	0 / 5	🟩 hibernate-3.0.5 (build 21) [2005-12-06 18:01]	🟩
release builds	------	0 / 5	🟩 hibernate-3.0.5 (build 22) [2005-12-13 05:28]	🟩
⊞ DOTNET-sample	------	0 / 0	------	⬛

1. Main navigation area

   From this area, user can navigate to different function areas of QuickBuild:

DASHBOARD	Navigate to this page.
CONFIGURATIONS	Navigate to configuration details.
BUILD QUEUES	Navigate to build queue details.
FIND BUILDS	Navigate to builds search page.
ADMINISTRATION	Navigate to administrative pages.

2. Configuration name

   Displays configuration name. Detail information of the configuration will be displayed when click on this link.

3. Open/Close button

   Click on this to open or close a particular configuration node.

4. Queued builds

Number of queued builds for this configuration. When click on this link, detail information about the queue bound to current configuration will be displayed.

5. History builds

   Number of history builds for this configuration. When click on this link, detail information about the history builds will be displayed.

6. Build status indicator

   Build status indicator for latest build of current configuration. GREEN means a successful build, RED means a failed build, while a running gear means a running build. Build log about this build will be displayed when click on this status indicator icon.

7. Latest build

   Latest build about current configuration. Build detail will be displayed when click on the build version.

8. Configuration status indicator

   Configuration status indicator. GREEN means recent triggering of current configuration is successful. RED means recent triggering of current configuration is failed, while a running gear means the configuration is currently running. Configuration log about this build will be displayed when click on this status indicator icon.

   ## Note

   Configuration status is different from build status. Configuration status means triggering status of a configuration, while build status means status of the build process. When a configuration has been triggered, it may fail to generate new build (for instance, error occurs when determine next build version). This will result in a failed configuration triggering. On the other hand, a new build can be generated and run, but the build process fails for some reason(for instance, a compiling error). This will result in a failed build, but triggering of the configuration is still successful, because this configuration has been successfully triggered in regardless of actual build status. Information about the build process will be logged into build log, and information about triggering of the configuration (such as checking build necessary condition, calculating next build version, etc.) will be logged into configuration log.

9. Start/Stop icon

   You can manually trigger the configuration by clicking on this button. When a configuration is running, a stop icon will be displayed which can be used to forcely stop current running cycle of current configuration.

   ## Note

   When you click on the start/stop button, and does not see any changes in the page, just try to refresh the page after some time.

   ## Note

   For particular configuration, if there are more than one triggering instance waiting in the build queue, stoping the configuration will only stop and remove current triggering instance, which will cause other waiting triggering instance being executed consequently. If you want to stop and remove all triggering instance, just go to detail page about the build queue, and remove all waiting and running triggering instances.

   ## Warning

   The stop action interrupts all threads involved in a running configuration. Normally it also kills all external OS processes created by your builder (for example, Ant builder will spawn a Java process to run build script). However on Windows platform, if your builder is executed through a Windows batch file, processes created in that batch file will not get killed. In this case, you should kill these processes manually, otherwise when you stop a configuration and run it again, you may encounter errors stating that `checkouts` directory can not be deleted, which normally means the spawned OS process is still running and accessing that directory. This is true for QuickBuild's Ant and Maven builder (executed by `ant.bat` and `maven.bat`) on windows platform.

10. Auto refresh switch

    You can turn on auto-refresh by clicking on ON/OFF link here. By making page auto-refreshing, the page will be automatically refreshed, which helps build status monitoring.

11. Quick search

    You can search builds by input partial of a build version, and click on RETURN key.

12. From this area, you can change your profile, password, access system log, and access system help.

    Note

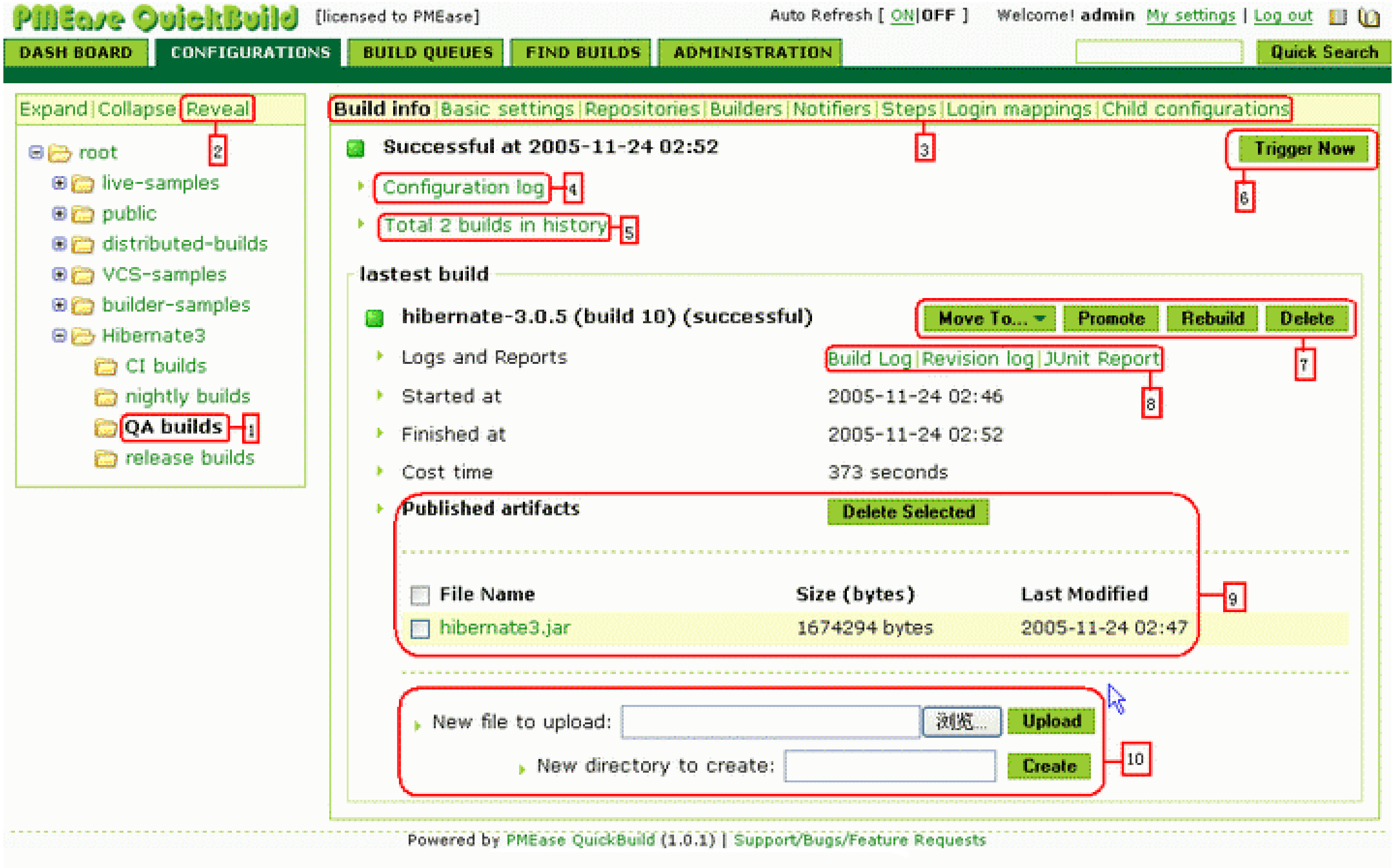
    System log is only available to administrator.

13. Schedule setting

    View or edit schedule setting for current configuration.

Configurations

This page shows detail information about particular configuration. User can navigate to this page by selecting *CONFIGURATIONS* from the main navigation area, or clicking on a particular configuration from the dash board page. Left side of the page is a tree of configurations, and right side is detail information about selected configuration in the tree.



1. Selected configuration

   User can click on particular configuration in the tree in order to view or edit information about that configuration.

2. Reveal selected configuration

   User can expand or collapse the whole configuration tree. When the tree is collapsed, user can click on the *reveal* link to only reveal selected configuration, and its ancestors, while keeping other configurations closed.

3. Tabs of a configuration

Build information	Display build information for selected configuration.
Basic settings	Display or edit basic setting for selected configuration.
Repositories	Display or edit repository information about selected configuration.
Builders	Display or edit builder information about selected configuration.
Steps	Display or edit step information about selected configuration. **Note** Step order in this tab does not make sense. Actual order of execution is determined by the serial composition step.
Login mappings	Display or edit login mapping information about selected configuration.
Child configurations	Display, create, delete, or move child configurations of selected configuration. **Warning** Before delete or move child configurations, you should make sure that there are no builds currently running inside them. Otherwise, the operation will fail.

4. Configuration log

   Click here to access log about selected configuration. Log level can be controlled in *basic settings* tab.

5. History builds information

   Display number of history builds. List of history builds will be displayed when click on this link.

6. Trigger button

   Click this button to create a new triggering instance inside a configuration.

7. Actions can be performed on a build

Move To	Move current build into another configuration.
Promote	Promote currrent build into another configuration. This action is only available when current build has a label in repository, because QuickBuild tries to retrieve and rebuild the same set of source codes in destinate configuration.
Rebuild	Re-generate current build. This action is only available when current build has a label in repository, because QuickBuild tries to retrieve and rebuild the same set of source codes.
Delete	Delete this build including all published artifacts. Artifacts published through soft links are not affected.

8. Logs and reports of a build

   Build log records log information about a build. Log level can be controlled through *basic settings* tab. Revision log records change logs since last build. JUNIT html report is a link to html reports generated by *junitreport* task in Ant. Clover html report is a link to html reports generated by *clover-report* task in Ant. Refer to this use case for details.

9. Published artifacts

   User can download published artifacts here. Exising artifacts can also be deleted here.

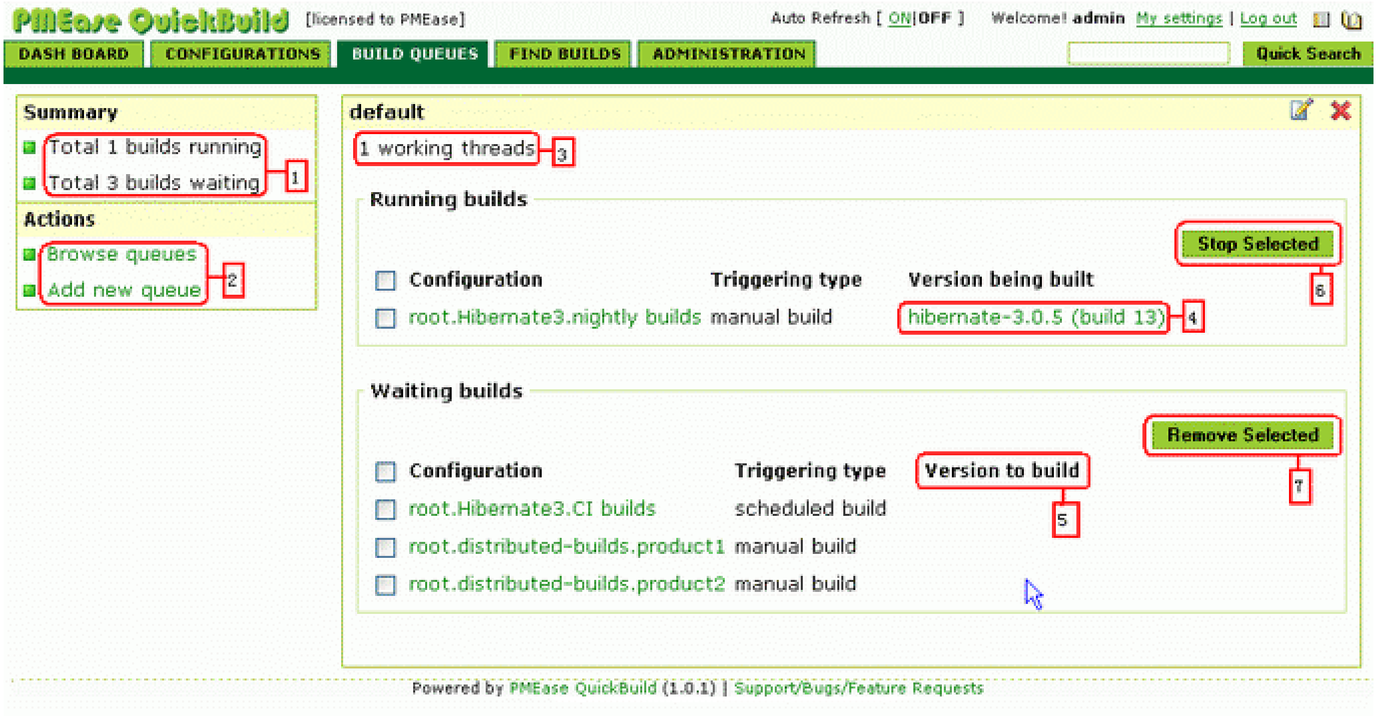
10. Upload new artifact or create new directory.

    User can upload new artifact or create new directory so that they are available as part of published artifacts. For example you can upload an installation guide here.

Build queues

This page shows detail information about build queues configured in the system. For every queue, there is a running builds section and waiting builds section. Running builds section lists all current running builds, and number of running builds is actually number of occupied working threads. Waiting builds section lists all waiting builds. Builds are put into waiting list for two reasons:

- There are no available working threads
- Another build is running and blocks current build (for example, if two configurations share the same working directory, one build in one configuration will block builds in another configuration).



Total 1 builds running

Total 3 builds waiting

1 working threads

Running builds

	Configuration	Triggering type	Version being built
☐	root.Hibernate3.nightly builds	manual build	hibernate-3.0.5 (build 13)

Waiting builds

	Configuration	Triggering type	Version to build
☐	root.Hibernate3.CI builds	scheduled build	
☐	root.distributed-builds.product1	manual build	
☐	root.distributed-builds.product2	manual build	

1. Summary section

   Summary section gives information about total number of running and waiting builds.

2. Actions section

   In this section, user can perform actions such as browse available queues, or create new queue.

3. Working threads

   Display total number of working threads configured in current queue.

4. Version being built

   Display the version currently being built. If the current build is a promotion build, from version and to version will be displayed as well.

5. Version to build

   Displays version to build. This value only available when you trigger a rebuild, or build manually with build version specified. For scheduled builds or manual builds without version specified, version to build can only be determined when this build is running.

6. Stop selected builds from running list

   User can select runnings builds, and click on this button to stop them forcely.

   ### Warning

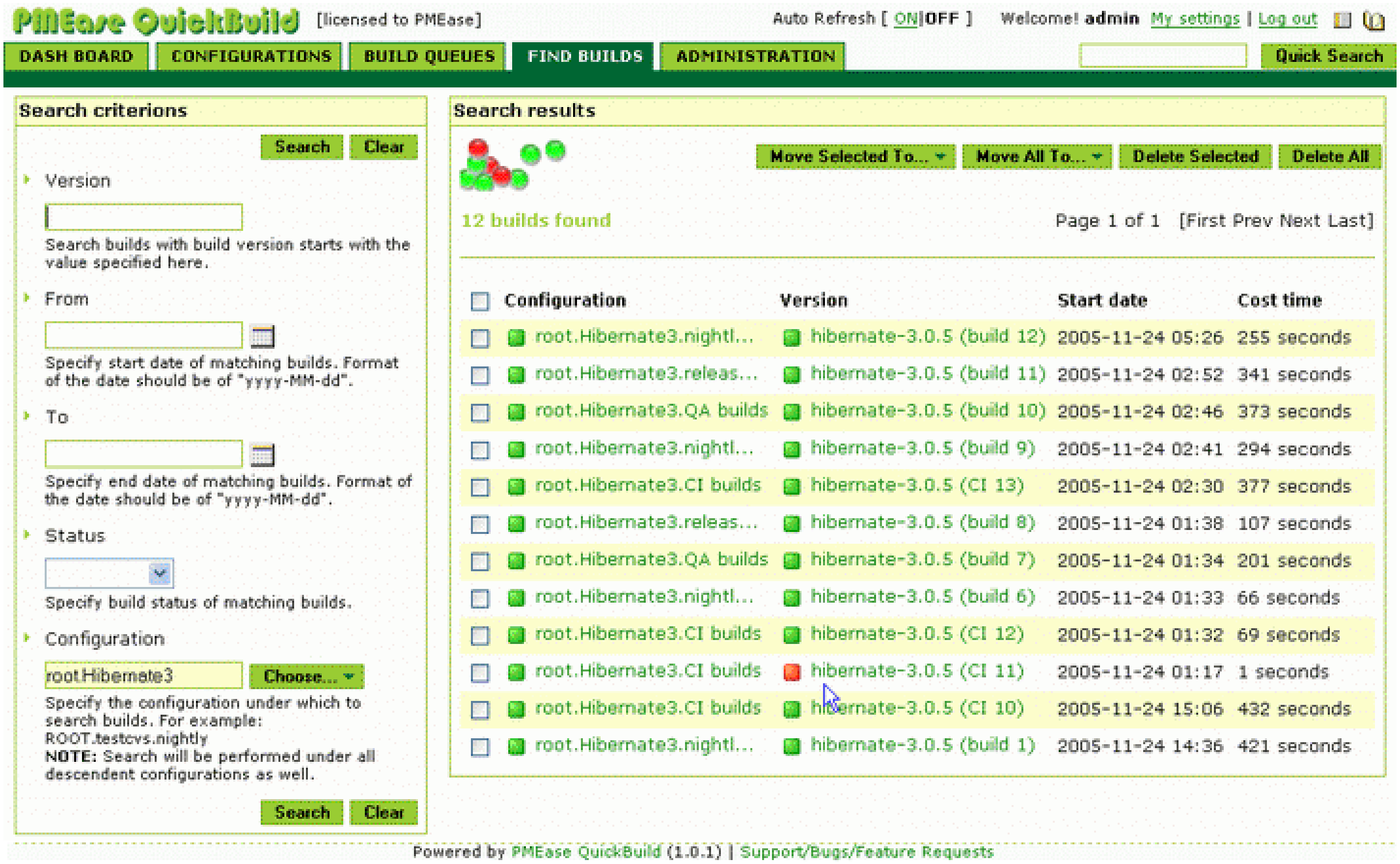
   On windows platform, if the builder command is a batch file (for example `ant.bat`), process created by commands in this batch file may still exist even the build is stopped.

7. Remove selected builds from waiting list

   User can select waiting builds, and click on this button to remove them from waiting list.

Find builds

In this page user can find particular builds by specifying search criterias such as version, date ranges, status, configuration. Right side of the page is search results. User can perform actions such as move, delete on these results.



Version

Search builds with build version starts with the value specified here.

From

Specify start date of matching builds. Format of the date should be of "yyyy-MM-dd".

To

Specify end date of matching builds. Format of the date should be of "yyyy-MM-dd".

Status

Specify build status of matching builds.

Configuration

root.Hibernate3 Choose...

Specify the configuration under which to search builds. For example: ROOT.testcvs.nightly
NOTE: Search will be performed under all descendent configurations as well.

12 builds found

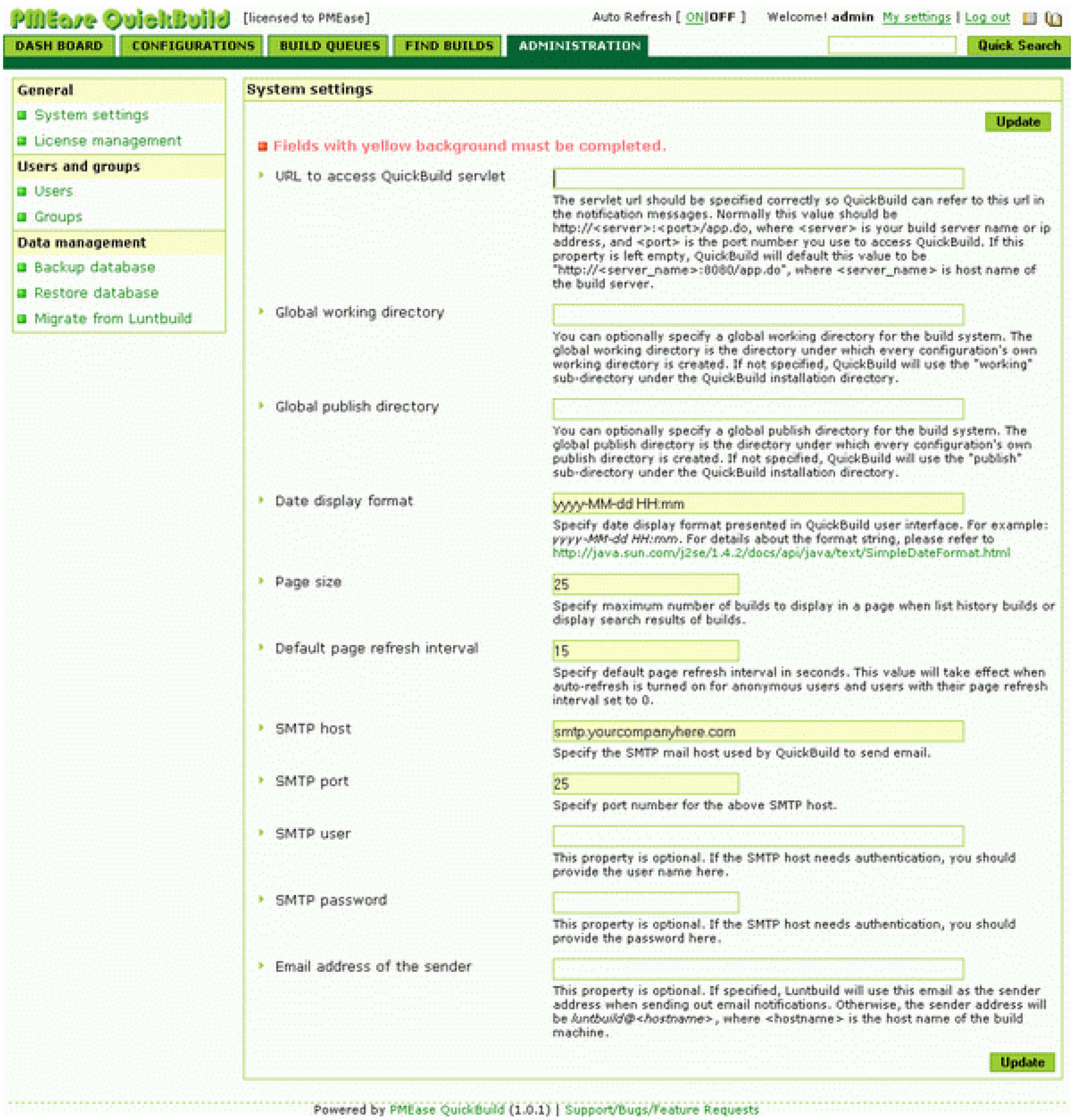
Page 1 of 1 [First Prev Next Last]

☐	Configuration	Version	Start date	Cost time
☐	🟩 root.Hibernate3.nightl...	🟩 hibernate-3.0.5 (build 12)	2005-11-24 05:26	255 seconds
☐	🟩 root.Hibernate3.releas...	🟩 hibernate-3.0.5 (build 11)	2005-11-24 02:52	341 seconds
☐	🟩 root.Hibernate3.QA builds	🟩 hibernate-3.0.5 (build 10)	2005-11-24 02:46	373 seconds
☐	🟩 root.Hibernate3.nightl...	🟩 hibernate-3.0.5 (build 9)	2005-11-24 02:41	294 seconds
☐	🟩 root.Hibernate3.CI builds	🟩 hibernate-3.0.5 (CI 13)	2005-11-24 02:30	377 seconds
☐	🟩 root.Hibernate3.releas...	🟩 hibernate-3.0.5 (build 8)	2005-11-24 01:38	107 seconds
☐	🟩 root.Hibernate3.QA builds	🟩 hibernate-3.0.5 (build 7)	2005-11-24 01:34	201 seconds
☐	🟩 root.Hibernate3.nightl...	🟩 hibernate-3.0.5 (build 6)	2005-11-24 01:33	66 seconds
☐	🟩 root.Hibernate3.CI builds	🟩 hibernate-3.0.5 (CI 12)	2005-11-24 01:32	69 seconds
☐	🟩 root.Hibernate3.CI builds	🟥 hibernate-3.0.5 (CI 11)	2005-11-24 01:17	1 seconds
☐	🟩 root.Hibernate3.CI builds	🟩 hibernate-3.0.5 (CI 10)	2005-11-24 15:06	432 seconds
☐	🟩 root.Hibernate3.nightl...	🟩 hibernate-3.0.5 (build 1)	2005-11-24 14:36	421 seconds

Powered by PMEase QuickBuild (1.0.1) | Support/Bugs/Feature Requests

Administration

In the administration tab, you'll be able to perform tasks such as edit system setting, manage licenses, manage users and groups. When the administration tab is selected, the first task "*System settings*" will be presented. You can choose other tasks through task navigation bar in left of the screen.



System settings

Fields with yellow background must be completed.

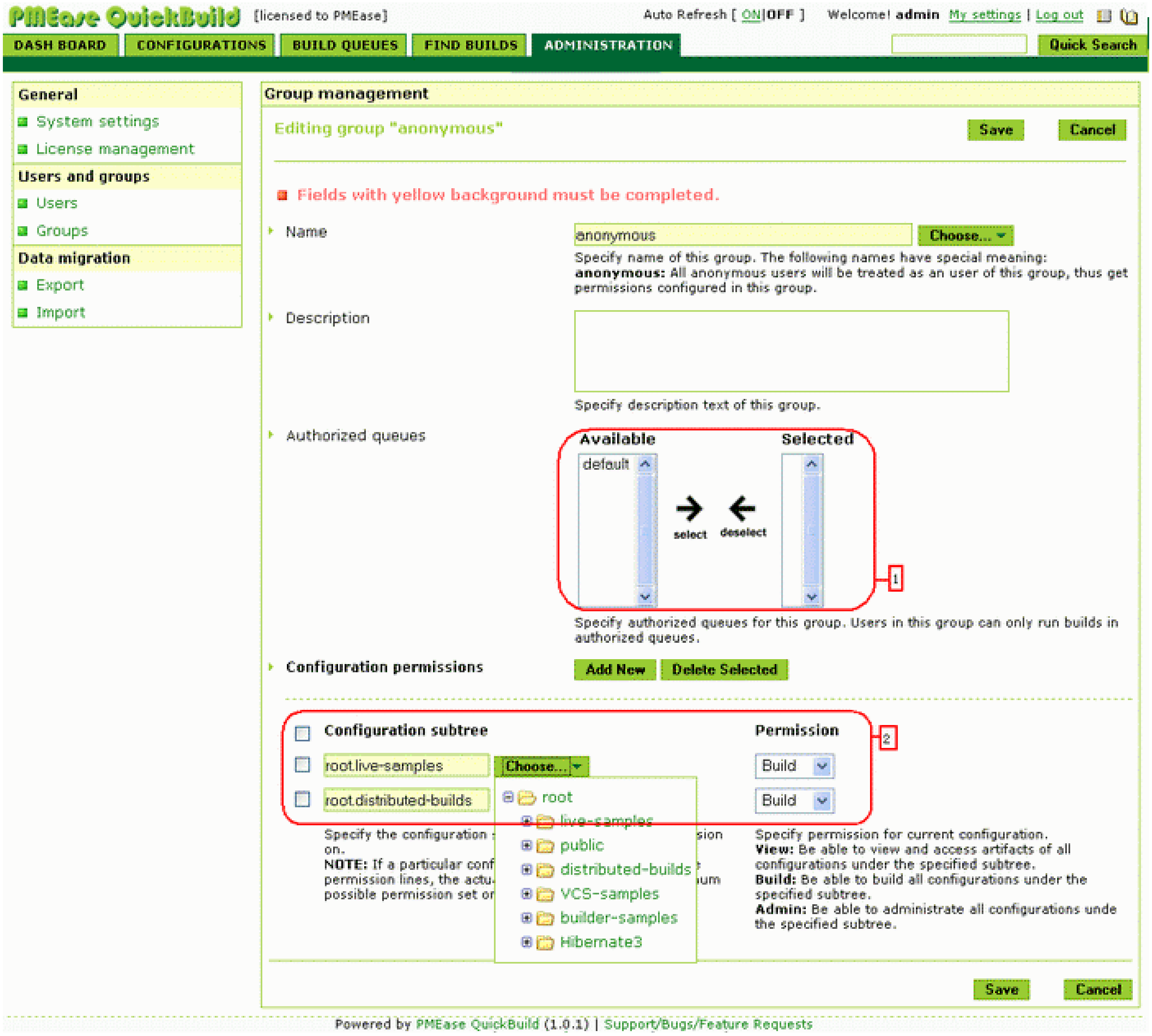
- URL to access QuickBuild servlet: The servlet url should be specified correctly so QuickBuild can refer to this url in the notification messages. Normally this value should be http://<server>:<port>/app.do, where <server> is your build server name or ip address, and <port> is the port number you use to access QuickBuild. If this property is left empty, QuickBuild will default this value to be "http://<server_name>:8080/app.do", where <server_name> is host name of the build server.
- Global working directory: You can optionally specify a global working directory for the build system. The global working directory is the directory under which every configuration's own working directory is created. If not specified, QuickBuild will use the "working" sub-directory under the QuickBuild installation directory.
- Global publish directory: You can optionally specify a global publish directory for the build system. The global publish directory is the directory under which every configuration's own publish directory is created. If not specified, QuickBuild will use the "publish" sub-directory under the QuickBuild installation directory.
- Date display format: yyyy-MM-dd HH:mm
  Specify date display format presented in QuickBuild user interface. For example: *yyyy-MM-dd HH:mm*. For details about the format string, please refer to http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html
- Page size: 25
  Specify maximum number of builds to display in a page when list history builds or display search results of builds.
- Default page refresh interval: 15
  Specify default page refresh interval in seconds. This value will take effect when auto-refresh is turned on for anonymous users and users with their page refresh interval set to 0.
- SMTP host: smtp.yourcompanyhere.com
  Specify the SMTP mail host used by QuickBuild to send email.
- SMTP port: 25
  Specify port number for the above SMTP host.
- SMTP user: This property is optional. If the SMTP host needs authentication, you should provide the user name here.
- SMTP password: This property is optional. If the SMTP host needs authentication, you should provide the password here.
- Email address of the sender: This property is optional. If specified, Luntbuild will use this email as the sender address when sending out email notifications. Otherwise, the sender address will be *luntbuild@<hostname>*, where <hostname> is the host name of the build machine.

Powered by PMEase QuickBuild (1.0.1) | Support/Bugs/Feature Requests

The following section gives explanation on some typical interfaces of administration tab.

Manage groups

User groups are designed to manage user permissions efficiently. By assigning a user to a particular group, you authorize that user doing operations permitted by that group. The following page depicts how to configure permissions for a group.



Editing group "anonymous"

Fields with yellow background must be completed.

Name: anonymous

Specify name of this group. The following names have special meaning:
anonymous: All anonymous users will be treated as an user of this group, thus get permissions configured in this group.

Description:

Specify description text of this group.

Authorized queues: Available: default; Selected:

Specify authorized queues for this group. Users in this group can only run builds in authorized queues.

Configuration permissions

☐	**Configuration subtree**	**Permission**
☐	root.live-samples	Build
☐	root.distributed-builds	Build

Specify the configuration on.
NOTE: If a particular conf permission lines, the actu possible permission set or

Specify permission for current configuration.
View: Be able to view and access artifacts of all configurations under the specified subtree.
Build: Be able to build all configurations under the specified subtree.
Admin: Be able to administrate all configurations unde the specified subtree.

1. Authorize queues for the group

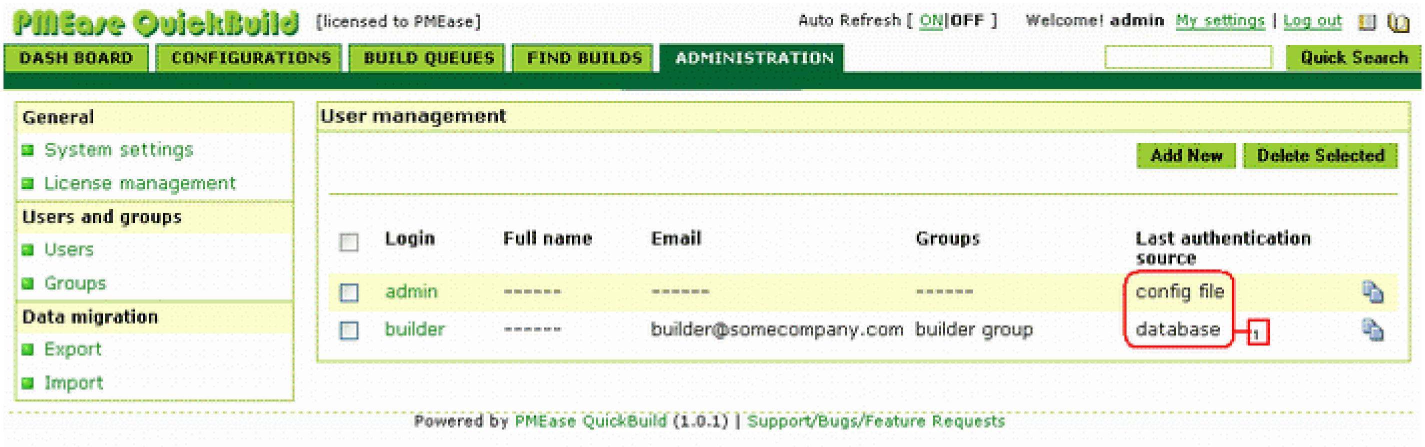
   From here you can set queue permissions for a group. For detailed information, refer to security chapter.

2. Set configuration permissions

   From here you can set configuration permissions for a group. For detailed information, refer to security chapter.

Manage users

The following is the user management page. From here, you can create/delete/edit users. For detailed information, refer to security chapter.



☐	Login	Full name	Email	Groups	Last authentication source
☐	admin	------	------	------	config file
☐	builder	------	builder@somecompany.com	builder group	database

1. Last authentication source

   Indicates the authentication source of this user's last login. Currently two authentication sources are supported, one is config file, which is only used for administrator, and another one is database, which is used by all users created through this page.

Chapter 3. Configure repositories

This chapter describes how to configure various repositories.

Configure Base Clearcase

You should have Clearcase client installed on the build machine. Also you should make sure that the account running your application server or servlet container is able to access your Clearcase server and that it can make snapshot views. Here is the list of properties you should configure for this repository:

Clearcase view stgloc name	Name of the Clearcase server-side view storage location which will be used as-stgloc option when creating Clearcase view for the current project. Either this property or "Explicit path for view storage" property should be specified.
Explicit path for view storage	This property is required only when the Clearcase view stgloc name property is empty. If specified, it should be parent directory of .vws directory for created snapshot view. For example, if you specify \\server1\dir1 here, QuickBuild will use \\server1\dir1\<view tag>.vws as the -vws option to create Clearcase view. Here <view tag> will be replaced by actual view tag. **Note** This value should be a writable UNC path on Windows platform.
Config spec	Config spec used by QuickBuild to create Clearcase snapshot view for a build. If you copy the config spec from your dynamic view, do not forget to add load lines after the config spec for each directory you need, like this:

```
include \\server\ClearCase\configspecs\myconfigspec.txt
load \myvob\modules\module1
load \myvob\modules\module2
```

Modification detection config	This property will take effect if there are some LATEST versions from some branch to fetch in the above config spec. It is used by QuickBuild to determine, if there are any changes in the repository since the last build. This property consists of multiple entries with each entry per line. Each entry is of the format <path>:<branch>. <path> is a path inside a vob. This path should start from the vob name, for example: \myvob\modules\mymodule. <branch> stands for name of the branch. For sub branches, you don't need to specify the names of any "super" -branches, just the name of the actual branch is enough.
Extra options when creating snapshot view	You may optionally specify extra options for the cleartool *mkview* sub command used by QuickBuild to create related Clearcase snapshot view for the current project. Options that can be specified here are restricted to *-tmode*, *-ptime*, and *-cachesize*. For example you can specify *-tmode insert_cr* to use Windows end of line text mode.
Path to cleartool executable	Specify path to your cleartool executable file. For example: /usr/local/bin/cleartool. It should be specified here, if it does not exist in the system path.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other

	checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.

Configure Clearcase UCM repository

You should have Clearcase client installed on the build machine. Also you should make sure that the account running your application server or servlet container is able to access your Clearcase server and that it can make snapshot views. Here is the list of properties you should configure for this repository:

Clearcase view stgloc name	Name of the Clearcase view storage location, which will be used as *-stgloc* option when creating Clearcase view for this project.
Project VOB tag	Tag for the project vob, for example: *\pvob1*.
Explicit path for view storage	This property is required only when the Clearcase view stgloc name property is empty. If specified, it should be parent directory of .vws directory for created snapshot view. For example, if you specify *\\server1\dir1* here, QuickBuild will use *\\server1\dir1\<view tag>.vws* as the *-vws* option to create Clearcase view. Here <view tag> will be replaced by actual view tag. **Note** This value should be a writable UNC path on Windows platform.
UCM stream name	Name of the UCM stream.
What to build	Specifies baselines you want to build inside the stream. Multiple baselines are separated by space. The following values have particular meaning:

<latest>	means build with all the latest code from every component.
<latest_bls>	means build with all the latest baselines from every component.
<rec_bls>	means build with all the recommended baselines.
<found_bls>	means build with all the foundation baselines.

Modification detection config	This property will only take effect when the What to build property equals to *<latest>*. It is used by QuickBuild to determine, if there are any changes in the repository since the last build. This property consists of multiple entries with each entry per line. Each entry is of the format *<path>:<branch>*. *<path>* is a path inside a vob. This path should start from the vob name, for example: *\myvob\modules\mymodule*. *<branch>* stands for the name of the branch. For sub branches, you needn't specify the names of any "super" -branches, just the name of the actual branch is enough.
Extra options when creating snapshot view	You may optionally specify extra options for the cleartool *mkview* sub command used by QuickBuild to create related clearcase snapshot view for the current project. Options that can be specified here are restricted to *-tmode*, *-ptime*, and *-cachesize*. For example you can specify *-tmode insert_cr* to use

	Windows end of line text mode.
Path to cleartool executable	Specify path to your cleartool executable file. For example: `/usr/local/bin/cleartool`. It should be specified here, if it does not exist in the system path.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.

Configure CVS repository

In order to use this adaptor, install appropriate CVS client based on your platform from http://www.cvshome.org or http://www.cvsnt.org if you are using Windows platform.

Note

Please keep time of the build server machine in sync with the Cvs server machine to allow build server to detect repository changes in Cvs server more accurately. Please make sure that times recorded in the Cvs revision log are in UTC time format instead of local time format.
Here is the list of properties you should configure for this repository:

Cvs root	The Cvs root for this repository, for example, :pserver:administrator@localhost:d:/cvs_repository. If you are using ssh, the :ext: protocol will need to be specified, and proper ssh environment needs to be set outside of Luntbuild. Please refer to your Cvs User's Guide for details.
Cvs password	The Cvs password for above Cvs root if connecting using pserver protocol.
Is cygwin cvs?	This property indicates whether or not the cvs executable being used is a cygwin one. The possible values are "yes" or "no". When omitted, the "no" value is assumed.
Disable "-S" option for log command?	This property indicates whether or not the "-S" option for the log command should be disabled. The possible values are "yes" or "no". When omitted, the "no" value is assumed. The -S option used in the log command can speed up modification detection, however some earlier versions of Cvs do not support this option. In this case you should enter "yes" value to disable it.
Disable history command?	This property indicates whether or not to disable the history command when performing modification detection. The possible values are "yes" or "no". When omitted, the "no" value is assumed. Using the history command in conjunction with the log command can speed up modification detection, however some Cvs repositories may not hold history information of commits. In this case you should enter "yes" value to disable it.
CVS executable path	Path to your cvs executable. For example: C:\program files\cvsnt\cvs.exe.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a

	build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.
Modules	Here are list of properties you should configure for a module entry.

Source path	Represents a module path in the CVS repository, for example */testcvs, /testcvs/web, or testcvs,* but you can not define a source path using / or \.
Branch	Specify the branch for the above source path. When left empty, main branch is assumed.
Label	Specify the label for the above source path. If specified, it will take preference over branch. When left empty, latest version of the specified branch will be retrieved.

At lease one module should be configured for this repository. *Source path* represents a module path in the CVS repository, for example */testcvs, /testcvs/web*, or *testcvs*, but you can not define a source path using "/" or "\". *Branch* stands for a CVS branch and *Label* stands for a CVS tag. Only one of these properties will take effect for a particular module. If both of them are not empty, label will take preference over branch. If both of them are empty, QuickBuild will get the latest code from main branch for a particular module.

Configure File system repository

Source directory	This is an optional property. If specified, changes can be detected in the source directory based on modification time, and modified files under this directory will be copied to the configuration's checkouts directory to perform build.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.

Note

Label step has no effect on this repository.

Configure Perforce repository

You should have Perforce client installed on the build machine. Contact http://www.perforce.com for licensing information. Here is the list of properties for this repository:

Perforce port	The Perforce port in the format of <port>, or <servername>:<port>, where <servername> and <port> will be replaced by the actual Perforce server name and the port number.
User name	User name to access the above Perforce server. This user should have the rights to create and edit client specifications and to checkout and label code.
Password	Password for the above user. Can be blank, if your Perforce server does not use password based security.

Line end	Set line ending character(s) for client text files. The following values are possible: local: use mode native to the client unix: UNIX style mac: Macintosh style win: Windows style share: writes UNIX style but reads UNIX, Mac or Windows style This property is optional. If not specified, the value will default to "local".
Path to p4 executable	Specify path to your p4 executable file, for example: /usr/local/bin/p4. It should be specified here, if it does not exist in the system path.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.

Modules Here are list of properties you should configure in order to define a module.

Depot path	Specify the Perforce depot side path, such as *//depot/testperforce/...*
Label	Specify the label for the above depot path. When empty, the latest version (head) of the above depot path will be retrieved.
Client side path	Specify the client side path, such as */testperforce/...*

Note

You should not put client name in this path. Before check out, QuickBuild will automatically generate a proper client name before this path, to form a client path like *//<generated client name>/testperforce/...*

Note

To exclude files or directories, create a separate module for each exclusion and precede the Depot path property with a minus (-) sign, for example:

```
Depot path:  -//depot/module1/...
Client path: /module1/...
```

At lease one module should be configured for Perforce repository. QuickBuild will construct client specification from module information configured here, and check out codes from Perforce repository accordingly. The user specified in Perforce connection information at the project level should have enough access rights to create and edit Perforce client specification.

Configure Subversion repository

In order to use this repository, Subversion client software should be installed on your build machine. You can download subversion from http://subversion.tigris.org [http://subversion.tigris.org/].

Note

Please keep time of the build server machine in sync with the Subversion server machine to allow build server to detect repository's changes in Subversion server more accurately.

Here are list of properties for this repository:

Repository url base	The base part of Subversion url, for example, you can enter *svn://buildmachine.foobar.com/*, or *file:///c:/svn_repository*, or *svn://buildmachine.foobar.com/myproject/othersubdirectory*, etc. Other definitions such as tags directory, branches directory, or modules are relative to this base url. **Note** If you are using *https://* schema, you should make sure that svn server certificate has been accepted permanently by your build machine.
Directory for trunk	Directory used to hold trunk for this url base. This directory is relative to the url base. Leave it blank, if you didn't define any trunk directory in the above url base.
Directory for branches	Directory used to hold branches for this url base. This directory is relative to the url base. If left blank, "branches" will be used as the default value.
Directory for tags	Directory used to hold tags for this url base. This directory is relative to the url base. If left blank, "tags" will be used as the default value.
Username	User name to use to login to Subversion.
Password	Password to use to login to Subversion.
Path to svn executable	Specify path to your svn executable file. For example: */usr/local/bin/svn*. It should be specified here, if it does not exist in the system path.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.

Modules Here are list of properties you should configure in order to define a Subversion module:

Source path	Represents a path in the Subversion repository, for example */, testsvn/web, or /testsvn*. This path is considered to be relative to the url base defined above. When branch or label properties are defined, this path will be mapped to another path in the svn repository.
Branch	Specify the branch for above source path. When left empty, trunk is assumed. **Note** Subversion does not internally has the notion of branch. Value specified here will be used by QuickBuild to do url mapping for the above source path so that actual effect is just like a branch in CVS repository.
Label	Specify the label for the above source path. If specified, it will take preference over branch. When left empty, head version of the specified branch is assumed. **Note** Subversion does not internally has the notion of label. Value specified here will be used by QuickBuild to do url mapping for the above source path so that

	actual effect is just like a tag in CVS repository.
Destination path	If specified, the contents from Subversion repository will be retrieved to the destination path relative to the configuration's checkouts directory. Otherwise the contents will be retrieved to source path (with no regard to branch or label) relative to the configuration's checkouts directory.

At least one module should be defined for Subversion repository. *Source path* represents a path in the Subversion repository, for example *testsvn*, *testsvn/web*, or */testsvn.* This path will be mapped to another path in the Svn repository based on other properties. In order to demonstrate this path mapping, we define following properties:

Repository url base: *svn://localhost*
Directory for trunk: *trunk*
Directory for branches: *branches*
Directory for tags: *tags*

We will examine the following module settings and give them the url mapping:

Source path: `testsvn/web`, branch: `<empty>`, label: `<empty>`, destination path: `<empty>`

QuickBuild will check out code from url `svn://localhost/trunk/testsvn/web` to directory `<configuration's checkouts directory>/testsvn/web`.

Source path: `testsvn/web`, branch: `dev2.0`, label: `<empty>`, destination path: `<empty>`

QuickBuild will check out code from url `svn://localhost/branches/dev2.0/testsvn/web` to directory `<configuration's checkouts directory>/testsvn/web`.

Source path: `testsvn/web`, branch: `<empty>`, label: `v1.0`, destination path: `<empty>`

QuickBuild will check out code from url `svn://localhost/tags/v1.0/testsvn/web` to directory `<configuration's checkouts directory>/testsvn/web`.

Source path: `testsvn/web`, branch: `dev2.0`, label: `v1.0`, destination path: `testsvn/web/dev2.0`

QuickBuild will check out code from url `svn://localhost/tags/v1.0/testsvn/web` to directory <`configuration's checkouts directory>/testsvn/web/dev2.0`.

Note

Branch definition is ignored here because label definition takes preference.

When QuickBuild tags a version for example *v1.0* for code checked out to directory `<configuration's checkouts directory>/testsvn/web`, the following command will be issued: *svn copy <configuration's checkouts directory>/testsvn/web svn://localhost/tags/v1.0/testsvn/web*

Of course you can avoid the above url mapping, by giving *Directory for trunk* property empty value, and giving *Branch* and *Label* properties both empty values. This way, you can control where to check out the code from, and where to put checked out code to, by just using the *Source path* and *Destination path* properties (in this case, source path will only be prefixed with *repository url base* property defined at the project level).

As you may realized, if QuickBuild need to create label *v1.0* for *moduleA* and *moduleB*, two directories will be created in Subversion repository, respectively `svn://localhost/tags/v1.0/moduleA`, and `svn://localhost/tags/v1.0/moduleB`. In this way under a certain url base, only one *tags* directory is needed to hold tag information for every module. This is the preferred tags or branches schema in QuickBuild. However, the suggested schema in Subversion user manual (every module has its own trunk, branches, or tags directory) is also supported by defining multiple repositories:

1. Create a Subversion repository with the following properties

Repository url base	svn://localhost/moduleA
Directory for trunk	trunk
Directory for branches	branches
Directory for tags	tags
Modules	Define moduleA with the following properties:

Source path	/
Branch	moduleA

2. Create the second Subversion repository with the following properties

Repository url base	svn://localhost/moduleB
Directory for trunk	trunk
Directory for branches	branches
Directory for tags	tags
Modules	Define moduleB with the following properties:

Source path	/
Branch	moduleB

3. Create two checkout steps to check out codes from the above two repositories respectively, and include these two steps in the default step.

Configure Visual Sourcesafe repository

In order to use this repository, visual sourcesafe need to be installed in your build machine. Download Visual Sourcesafe from http://download.microsoft.com.

Warning

You need to run the application server (Tomcat by default) that hosts QuickBuild as a foreground process (instead of a NT service) in order to work with Sourcesafe repository.

Note

In order to keep output of history command of Visual Sourcesafe accurate, time setting of all developer workstations, and the build server should be kept in sync.

Note

Currently only English version of Sourcesafe is supported.

The following list of properties needs to be configured:

Sourcesafe path

The directory where your srcsafe.ini resides in. For example: `\\machine1\directory1`.

Note

You should login to the remote machine first.

Username

User name to use to login the above Sourcesafe database.

Password

Password for the above user name.

Datetime format

Specify the date/time format used for the Sourcesafe history command. This property is optional. If left empty, Luntbuild will use "M/dd/yy;h:mm:ssa" as the default value. The default value is suitable for English language operating systems using US locale. For other English speaking countries with different date format like UK, Australia, and Canada the Visual Sourcesafe Date format to use (assuming you're using the appropriate locale setup as Visual Sourcesafe honors the local locale settings) should be as follows:

```
'd/M/yy;H:mm'
```

If QuickBuild is running on non-english operating systems, use the following method to determine the datetime format:

Open Visual Sourcesafe *installed on your build machine*, select an existing VSS database and choose to view one of the projects with files in it. There should be a list of files shown with several fields including the "Date-Time" field. You should use the *datetime format* property from value specified in this field. For example, if one of the values of this field is *04-07-18 20:19*, the *datetime format* property should be *yy-MM-dd;HH:mm*. The *semicolon* between date and time format should be specified. You are encouraged to specify the property as *yy-MM-dd;HH:mm:ss* to add the accuracy. Take another example, if the value shown in Visual Sourcesafe is *7/18/04 8:19p*, the *datetime format* should be *M/dd/yy;h:mma*. Format *M/dd/yy;h:mm:ssa* would increase the accuracy in this case.

The following is a list of format character meanings copied from JDK document:

Table 3.1. Date/Time format characters

Character	Meaning	Example
y	Year	1996 ; 96
M	Month in year	July ; Jul ; 07
d	Day in month	10
a	Am/pm marker	p
H	Hour in day (0-23)	0
h	Hour in am/pm (1-12)	12
m	Minute in hour	30
s	Second in minute	55

For details about the format string, please refer to http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html [http://java.sun.com/j2se/1.4.2/docs/api/java/text/SimpleDateFormat.html]

Path to ss.exe

Path to your ss.exe. For example: `C:\Program Files\Microsoft Visual Studio\Common\VSS\win32\ss.exe`. It should be specified here, if it does not exist in the system path.

Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.
Modules	Here are list of properties you should configure in order to define a module:

Source path	Specify the path in the VSS repository, for example: */testvss*, or *$/testvss*. To specify the whole repository, just use /, or $/.
Label	Specify the label for the above source path. This property is optional. If left empty, latest version is assumed.
Destination path	Specify the directory relative to the checkouts directory of current configuration, where the contents under the above source path should be retrieved to. If left empty, retrieved code will be put into directory defined by the source path relative to the checkouts directory.

Source path represents a project path relative to the root of Sourcesafe, for example *testvss, /testvss, or /testvss/web*, etc. Path / or \ can be used to retrieve the whole contents of the repository. *Label* stands for a VSS label. VSS implements branches by creating a new shared Sourcesafe projects. So you may need to configure different modules in order to get code from different branches. If *Label* is left empty, QuickBuild will get latest code for that module from VSS. If *Destination path* is defined, contents from Sourcesafe will be retrieved to *Destination path* relative to the configuration's checkouts directory. Otherwise the contents will be put to *Source path* relative to configuration's checkouts directory.

Warning

Because Visual Sourcesafe has the limitation that only one label can be attached to a particular version (except for head version). So if you define a module with a particular label, retrieve it for build, and set new label after the build, the original label will get removed, which will cause failure of subsequent builds (because original label can not be found in VSS repository). Take an example, for a particular configuration, we define a VSS repository with the following modules:

- The first module retrieves latest code from componentA

Source path	/src/componentA

- The second module retrieves code from componentB with label *v1.0*

Source path	/src/componentB
Label	v1_0

We define steps to retrieve and build against codes from this repository, and create new label after build. After the first build in this configuration, latest version under `/src/componentA` will get a new label (*v2_0* for instance), and label *v1_0* of /src/componentB will be replaced by *v2_0*. Everything goes well now, but the second build in the same configuration will get failed, because label *v1_0* under `/src/componentB` does not exist. Instead you should replace *v1_0* with *v2_0* in second module definition in order to get it build successfully again. So you should change label value of second module to latest label everytime you a new label has been generated. This is somewhat unacceptable. A better solution is to share *v1_0* of source path `/src/componentB` into another project (select *branch after share* option), say `/src/componentB-v1_0`, and get it pinned. Then for second module, the definition can be:

Source path	/src/componentB-v1_0
Destination path	/src/componentB

In this way, new labels are attached to head version (get pinned of course) of `/src/componentB-v1_0`, which is allowed in Sourcesafe, and the effect is the same as retrieving codes from label *v1_0* under `/src/componentB`.

Configure StarTeam repository

For Windows platform, you will need to have a full installation of StarTeam SDK runtime (which will install some runtime dlls and put them in the Windows system path). Normally this is the part of StarTeam client installation. Please go to http://www.borland.com for licensing information. Here is the list of properties for this adaptor:

Project location	Location of a StarTeam project is defined as: <servername>:<portnum>/<projectname>, where <servername> is the host where the StarTeam server runs, <portnum> is the port number the StarTeam server uses, default value is 49201. <projectname> is a StarTeam project under this StarTeam server.
User	User name to login to the StarTeam server.
Password	Password to login to the StarTeam server.
Convert EOL?	The following values are possible: *yes:* all ASCII files will have their end-of-line characters adjusted to the EOL type of the local machine on checkout *no:* the files will be checked out with whatever EOL characters are used on the server This property is optional. If not specified, it will default to *yes*.
Time difference threshold	Specify time difference threshold (measured in seconds) between build server and StarTeam server. Time difference between build server and StarTeam server should not exceed this value. Otherwise, checkouts may fail due to trying to pull codes of future time from StarTeam server. However, you should not set a too large threshold in order to check out latest codes. For most cases, 10 seconds will be a good choice.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.
Modules	Here are list of properties should be configured in order to define a StarTeam module:

StarTeam view	Specify a StarTeam view. If it is left empty, the root view of the current StarTeam project will be used.
Source path	Specify a path relative to the root of the above StarTeam view. Enter / to specify the root.
Label	Specify the label for the above StarTeam view. When left empty, the latest version of specified view is assumed.
Destination path	Specify the directory relative to the checkouts directory of current configuration. Contents under the above source path will be retrieved to this directory. When left empty, retrieved code will be put into directory specified in source path, relative to the checkouts directory.

StarTeam view stands for a StarTeam view, and *Label* stands for a label of this StarTeam view. If *StarTeam view* is left empty, the root StarTeam view will be used. *Source path* is a path relative to the root of the chosen StarTeam view. If *Destination path* is defined, the contents from StarTeam repository will be retrieved to the *Destination path* relative to configuration's checkouts directory, otherwise the contents will be put to the *Source path* relative to the configuration's checkouts directory.

Warning

When define modules, If you want QuickBuild to create new label after build, you should define only one module per StarTeam view. The reason is: When create label, QuickBuild goes through each defined module and tries to create view label inside the StarTeam view associated with that module. If more than one module is defined for a particular StarTeam view, QuickBuild will try to create the same view label more than once in that view, and causes an *Label already exist* exception. Take an example, let's assume we've defined two modules:

- The first module

StarTeam view	development
Source path	componentA

- The second module

StarTeam view	development
Source path	componentB

When QuickBuild tries to create new label, say *v2_0*, it will go through these two defined modules, and try to create view label *v2_0* for their associated StarTeam view respectively, which will cause the same view label being created twice in *development* view, and causes error consequently. To avoid this, at StarTeam side, you can create another view, say *componentB-development* rooted at `/componentB` and set *branch on change* option. At QuickBuild side, change definition of second module to be:

StarTeam view	componentB-development

Source path	/
Label	componentB

In this way, view label *v2_0* will be created in different branch view, which is allowed in StarTeam.

Configure Accurev repository

You are able to build against particular Accurev stream. When perform build, QuickBuild creates reference tree to check out contents of that stream. You can get location of this reference tree through method call *getWorkspaceDir(build)*. Normally this is a sub directory named by build stream under current configuration's *checkouts* directory. As regard to label step in QuickBuild, it will result in creating corresponding snapshots in Accurev.

Build stream	Specify stream name to build against. You can also specify a snapshot name here to build against a particular snapshot. Other settings such as Accurev server, user name, password will be taken from Accurev client installed at this build machine.
Additional streams to detect changes from	Specify additional streams from which to detect changes. This is useful when you want to detect promotions made into other streams whose file changes will be propagated to the build stream. Multiple streams should be separated by spaces, and single stream name containing spaces should be quoted.
Accurev executable path	Path to your Accurev executable. For example: C:\Program Files\AccuRev\bin\accurev.exe. It should be specified here, if it does not exist in the system path.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.

Configure QuickBuild repository

QuickBuild repository is used to check out artifacts from other QuickBuild configurations (may resides in a different build machine). Here is the list of properties for this repository:

Remote QuickBuild URL	Specify servlet URL for the QuickBuild system you want to retrieve artifacts from. For example, "http://another-server:8080/app.do". If not specified, it will default to current QuickBuild system.
Configuration	Specify configuration of the above QuickBuild system, for example: "root.project1.release". This configuration and the following build property will uniquely identify the build where you want to retrieve artifacts from.
Build	Specify version of the build from which you want to retrieve artifacts from, for example: "myproduct-1.0.0". If not specified, latest build will be assumed. Meaning

of some special build version is listed as below:

<latest build>	Triggers destination configuration and then check out artifacts from latest build from specified configuration.
<last build>	Check out artifacts from last build of specified configuration. Last build is the latest build that has been finished.
<last successful build>	Check out artifacts from last successful build of specified configuration.

User to login	Specify user name to login to the specified QuickBuild system. It should have the permission to view the above configuration. If you are trying to retrieve artifacts of the latest build, you should have build permission for that configuration, because it will be triggered to generate a new build if necessary. If this property is not specified, anonymous user will be assumed.
Password	Specify password of the above user.
Quiet period	Number of seconds current repository should be quiet (without checkins) before QuickBuild decides to check out the code from this repository for a build. This is used to avoid checking out code in the middle of some other checkins. This property is optional. When set as 0, quiet period will not be used before checking out code to build.
Login mapping	Choose login mapping for this repository. Login mapping is used to map repository login to QuickBuild user. It can be configured at "login mappings" tab of the configuration.
Modules	Here are list of properties should be configured in order to define a QuickBuild module:

Source path	Specify source path to retrieve artifacts from. This path is relative to artifacts directory of specified build above. So "." will refer to the artifacts directory itself.
File name patterns	Specify file name patterns of artifacts to retrieve, for example: *.zip, ${build.version}*.zip, ${build.version}.*. Multiple patterns can be specified as long as they are seperated by spaces(Of course, a single pattern includes spaces should be quoted). If left empty, all files will be retrieved.
Retrieve recursively	Whether or not to retrieve matched artifacts recursively under specified source path.
Destination path	Specify the directory relative to the checkouts directory of current configuration, where the matching artifacts under the specified source path should be retrieved to. If left empty, retrieved code will be put into directory defined by the source path relative to the checkouts directory.

Chapter 4. OGNL expressions

Although OGNL expression is widely used in QuickBuild, generally, you need not know anything about it due to QuickBuild's deliberate designed user interface. For most properties that need OGNL expression, you can choose from drop down menu besides that property simply. However, knowledge of OGNL expression will give you ability to set up very complicated configurations. http://www.ognl.org is the official site for OGNL where you can learn everything about it. This chapter assumes that you have some knowledge of it, and will concentrate on properties and methods that can be used in QuickBuild.

Generally, there are two types of application of OGNL in QuickBuild. The first one is for boolean type properties, including *Build necessary condition*, *Step necessary condition*, *Build success condition*, *Step success condition*, etc. These properties expect an OGNL expression that will be evaluated to a boolean value. The second one is for string type properties. Inside this string, any number of OGNL expressions can be embedded as long as they are embedded in ${...}, and QuickBuild expects they evaluate to a value of string type. Strings outside of ${...} will simply keep the same during the evaluation. Every string-typed property in QuickBuild accepts OGNL expressions embedded within ${...} as long as there is not explicitly statement of nonsupport. Please be noted that double back slashes should be used for windows path in an OGNL expression, for example, *${"c:\\program files\\apache-ant-1.6.5\\bin\\ant.bat"}*. Further more, if you assign windows path as value of a variable, and refer to that variable in OGNL expression, then that windows path should also use double back slashes. For example, in order to assign windows path of ant executable to variable *pathToAnt*, you may need to define it as:

```
pathToAnt = "c:\\program files\\apache-ant-1.6.5\\bin\\ant.bat"
```

The path is quoted because there are spaces inside it.

Maybe you know, every OGNL expression should have a root object in order to perform the evaluation. In QuickBuild, the root object is always current configuration object. From this configuration object, you can access its exposed methods or properties, such as name, status, variables, repositories, builders, steps, current build, current object, etc. Once you get a stuff out of the configuration object, you can recursively get other properties or methods exposed by that particular property. The definitive guide of these exposed methods or properties is the JavaDoc [../javadoc/com/pmease/quickbuild/model/Configuration.html]. There are a lot of methods or properties in the JavaDoc, you should only pay attention to methods or properties with **OGNL:** at the very beginning of the comment. Here are some typical OGNL expressions:

Get the build object that is running

build

Get version of the build object that is running

build.version

Get last build object

lastBuild

Get last successful build object

lastSuccessBuild

Determine if last build is successful

lastBuild.successful

Determine if last build is failed

lastBuild.failed

Determine if last build is running

lastBuild.running

Determine if contents of repository "*cvs1*" have been modified since last build

repository["cvs1"].modified

Get source path of first module of repository "*svn*1"

repository["svn1"].modules.get(0).srcPath

Get head revision number of repository "svn1"

repository["svn1"].headRevision

Get workspace directory of repository "*accurev1*" for current build

repository["accurev1"].getWorkspaceDir(build)

Get version of the remote build used by a QuickBuild repository

repository["quickbuild1"].remoteBuild.version

Determine if contents of all used repositories have been modified since last build

effectingRepositoriesModified

Determine if contents of repository "cvs1" have been modified since last successful build

lastSuccessBuild==null or repository["cvs1"].isModifiedSince(lastSuccessBuild.startDate, configuration)

Get string value of variable "*var1*"

var["var1"]

Get value of variable "*var1*" as integer

var["var1"].intValue

Increase value of variable "*var1*" as integer

var["var1"].increaseAsInt()

Set value of variable "*var1*" as "*value1*"

var["var1"].setValue("value1")

Increase value of variable "*var1*" as integer, and get the increased result

var["var1"].(increaseAsInt(), value)

Determine if value of variable "*var1*" equals "*true*"

var["var1"].value=="true"

Get working directory of current configuration

```
workingDir
```

Determine if execution of specified command is successful

```
system.execute("/path/to/my/command") == 0
```

Get current hour

```
system.calendar.hour
```

Get day of week

```
system.calendar.dayOfWeek
```

Determine if execution of step "*step1*" is successful

```
step["step1"].successful
```

Chapter 5. Velocity templates

Velocity templates are used to customize content of build notifications. Detailed information about velocity template can be accessed from Apache's web site [http://jakarta.apache.org/velocity/index.html]. Two variables are defined to help inserting information related to QuickBuild. The first variable is *build [../javadoc/com/pmease/quickbuild/model/Build.html]*, which refers to current build object. From this variable, information such as build version, build status, configuration, build log, revision log can be accessed. JavaDoc [../javadoc/com/pmease/quickbuild/model/Build.html] of the build class will be the definitive guide on what methods or properties can be accessed from this variable. There are a lot of methods defined in this class, you should pay attention to those started with **OGNL:** prefix (In QuickBuild, all properties or methods for OGNL expressions can also be used to construct velocity templates). The second variable is *system [../javadoc/com/pmease/quickbuild/util/system.html]*, which refers to current QuickBuild system. From this variable, information such as calendar, system url can be accessed. Again from the JavaDoc [../javadoc/com/pmease/quickbuild/util/system.html], you can get the full guide on what methods can be called on this variable.

Other files can be included in the template, as long as they are put under `<QuickBuild installation directory>/templates`. For example, the default body template for Email notifier contains just one line: *#parse ("html_notification.vm")*. This line includes and parses content of file `html_notification.vm` in `templates` sub directory of QuickBuild installation directory.

To facilitate writing of your own notification templates, some typical variable references are listed below:

Get current build version

$build.version

Determine if current build is successful

$build.successful

Determine if current build is failed

$build.failed

Get current build url

$build.url

Get build log url of current build

$build.buildLogUrl

Get revision log url of current build

$build.revisionLogUrl

Get build log path of current build

$build.buildLogPath

Get revision log path of current build

$build.revisionLogPath

Get name of current configuration

`$build.configuration`

Get url of current configuration

`$build.configuration.url`

Get log url of current configuration

`$build.configuration.logUrl`

Get log path of current configuration

`$build.configuration.logPath`

Get 50 lines arround the error line of build log

`$system.readFileAsHtml("$build.buildLogPath", ".*ERROR.*", 25)`

Get content of revision log

`$system.readFileAsHtml("$build.revisionLogPath")`

Chapter 6. Security

With QuickBuild, you will be able to control that who can access your build system, and what they can do on it. Further more, you can authorize anonymous users accessing certain part of your system.

User authentication

Currently QuickBuild supports authenticating users from two sources: configuration file and database. Users from other authentication source such as CAS, LDAP will be supported in later versions. Configuration file authentication source is only used for administrator with initial **user/password** as **admin/admin**. Path to the configuration file is `<QuickBuild installation directory>/config.properties`. Administrator password can be changed by editing this file. User **admin** is the only super user in QuickBuild system, and has the permission to perform tasks such as managing users/groups, configuring system settings, managing licenses, managing build queues, backing up/restoring system, or exporting/importing data, etc.

Another supported authentication source is database. Users created by administrator in QuickBuild are saved into database, and will be authenticated through that database later on. Please be noted that the *Users* menu lists not only all users in the database, but also users from other authentication source (here you will see *admin* been listed). The reason is that although QuickBuild authenticate them through other authentication sources, it still maintains contact information such as Email, MSN messenger account, which may be used to send out build notification to these users. Changing password for these users here has no effect. Instead, password should be changed from their own authentication source (such as editing config file in order to change password for the *admin* user).

User authorization

User authorization functionality is achieved by assigning users to proper groups. You can create several groups; with each group has its own set of permissions. Multiple groups can be assigned to a single user, which means this user has all permissions set in those groups. You can create new group or editing existing group through *Groups* menu of *Administration* tab. While creating new groups, the special name *anonymous* is reserved for anonymous users, that is, all anonymous users will be assigned to this group automatically, thus get permissions set in this group. Basically there are two type of permission you need to set for a group: queues and configurations.

Set queue permissions

By specifying authorized queues for a group, you allow users of that group be able to bind those queues to configurations they have permission to edit. This is useful, for example, if you want that only one build can be performed at the same time (in order to save CPU cycle) for some projects. In order to do achieve this, you create a queue with only one working thread, and create a group with only this queue authorized, and assign users from those projects to this group. In this way, users of that group can only use that particular queue, which will streamline all triggered builds, instead of run them simultaneously.

Set configuration permissions

By specifying configuration permissions for a group, you authorize users of that group be able to access certain configurations in the system, and what operations they can do on those configurations. This is done by choosing desired configuration subtree, and set appropriate permission on that tree; then all configurations under choosed subtree will get specified permission. There are three types of permission you can set:

View	Be able to view and access artifacts of all configurations under the choosed subtree.
Build	Be able to build all configurations under all configurations under the choosed subtree.
Admin	Be able to administrate all configurations under the choosed subtree.

For a particular configuration, if it is affected by multiple permission lines, the actual permission will be the maxium possible permission. In order to demonstrate this, let's assume that configuration permissions are set as below:

Table 6.1. Configuration permissions

Configuration subtree	**Permission**
root.department1	Admin
root.department1.project1	View

Although second permission line states that all configurations under *root.department1.project1* have *View* permission, but because the first line states that all configurations under *root.department1* have *Admin* permission, configuration *root.department1.project1* will take *Admin* permission which is the maximum possible permission. This rule also applies when determine permission for a configuration that is affected by multiple groups assigned to the same user.

Chapter 7. Remote and parallel build support

Mechanism

Let us suppose the following scenarios:

1. Your product consists of components that should be built on different platforms. For example, in order to package your product, you need to build component1 on windows, build component2 on linux, and collect build result of them to form a single package.
2. It may take a long time to build your product. Ideally, you may want to build different part of the product on different machine simultaneously, and then collect back build results of these parts to form a single product.
3. Build a product on one machine, and send build result to another machine for smoking/automation tests. If tests pass, mark current build as successful;otherwise, mark it as failed.

QuickBuild can be used to handle these complicated scenarios quite easily. The key is a special repository type: *QuickBuild*. The QuickBuild system itself can be treated as a type of repository just like other repositories such as CVS. This is reasonable, cause repository is just a place holding artifacts which can be used during a build process. These artifacts can be source code saved in CVS repository, and can also be Jars or DLLs generated by and saved in a QuickBuild system. Several use cases are created to demonstrate how QuickBuild supports multi-platform remote builds.

Deadlock prevention

Certain mis-configuration of remote builds may cause deadlock. They are listed here so that you can avoid of doing so (It is highly recommended that you read these use cases first to get thorough understanding of QuickBuild's remote build support before try to understand deadlock scenarios listed here):

1. If *configuration1* and *configuration2* shares the same working directory, and you set up *configuration1* to be dependent on *configuration2*. When *configuration1* is triggered, it triggers execution of *configuration2* and tries to wait for completion of that execution, but *configuration2* will wait on *configuration1* to release shared working directory lock. Thus deadlock occurs.
2. If *configuration1* depends on *configuration2*, and *configuration2* depends on *configuration1* (see this use case). If you actively trigger *configuration1* and *configuration2* at the same time, deadlock may occur. The reason is that, if you actively trigger *configuration1*, it will hold working directory lock of *configuration1* and dependently triggers *configuration2*, and wait until that triggering has been finished. Dependent triggering of *configuration2* tries to get working directory lock of *configuration2*, which may now be hold by an active triggering of *configuration2*. Just like active triggering of *configuration1* does, this active triggering of *configuration2* will also cause dependent triggering of *configuration1*, which will wait on working directory lock of *configuration1*. Thus, working directory lock of both configurations can never be released, and deadlock occurs.

Chapter 8. Data management

Backup and restore database

You are able to backup and restore contents of the database used by QuickBuild through *data management* section of the *Administration* tab. It is a good idea to backup your database regularly to ensure you can restore your data in case of a hardware failure. Please be noted that the backup and restore process only affect data saved in the database (such as configurations, groups, users etc.), all other contents in the disk (such as generated build artifacts) will not be touched.

Warning

The backup process may take a long time if the system contains large amount of builds. You can speed up this by deleting unwanted builds. Also it is a good idea to set a *reserve by days* (or *reserve by count*) build cleanup strategy for configurations which will generate many many builds over time (for example configurations created for continuous integration purpose).

Data migration between different databases

By backing up the database, QuickBuild actually saves contents of the database into a portable XML file, which can be restored to a different database. In this way you are able to migrate data between different databases. For instance, if you want to migrate data from HSQLDB (the default database used by QuickBuild) to MySQL, the following steps can be performed:

1. Bring up QuickBuild and backup database to an XML file, say `data.xml`.
2. Shutdown QuickBuild, and edit `<QuickBuild installation directory>/config.properties` to connect to MySQL.
3. Bring up QuickBuild again, and restore database from `data.xml`.

Migrate data from QuickBuild PREVIEW2 and PREVIEW3

To migrate from QuickBuild PREVIEW2 and PREVIEW3, please go through the following steps:

1. If you are using MySQL as database, edit file "config.properties" to connect to your existing MySQL database.

   ### Note

   Please do not simply overwrite "config.properties" with the same file from existing installation, cause this file has been changed since PREVIEW3.

2. If you are using HSQLDB as database:
   - Remove the "data" sub directory of QuickBuild 1.0.1.
   - Copy "data" sub directory from existing installation to QuickBuild 1.0.1.
3. Copy "publish" sub directory from existing installation to QuickBuild 1.0.1.

Migrate data from Luntbuild 1.2.x

To migrate data from Luntbuild 1.2.x, please go through the following steps:

1. Start up Luntbuild, and export data into "data.xml" under Luntbuild installation directory.
2. Login to QuickBuild as "admin", change to "administration" tab, and select "Migrate from Luntbuild".
3. During the migration, all build artifacts will be picked up from your existing Luntbuild instance and modified to be compatible with QuickBuild's artifacts structure, and put under the "publish" sub directory of QuickBuild. This may take a long time if you have many many builds, so be patient...
4. If you just want to migrate configuration data (such as projects, schedules), but not build artifacts, you can simply select "Restore database" menu item, and specify path to "data.xml" created in step 1.
5. After the migration, please be aware of the following things:
   - OGNL expressions used in "next build version", "build properties", and "build environments" are not migrated. You should make sure they conform to new OGNL expressions described in Chapter 4 in this document.
   - Project dependencies are not migrated, cause dependency mechanism in QuickBuild is very different from Luntbuild. You need to understand the new dependency mechanism and get them re-configured.
   - Properties such as "build necessary condition" and "build success condition" have been reset to conform to new format of QuickBuild. You can customize them further to fit your needs.

For Luntbuild 1.x users, please migrate to Luntbuild 1.2.x first, then following the above instructions to migrate to QuickBuild.

Chapter 9. REST API

QuickBuild has a REST (REpresentation State Transfer) API implemented in Hessian [http://www.caucho.com/hessian/] protocol (a lightweight SOAP protocol). This API is created to help you integrate QuickBuild with other systems (such as issue tracking application), or extend the ability of QuickBuild. The following are some scenarios on using this API:

1. Trigger a build right away when somebody checks code into the version control system. In order to do this, you should write a program calls the *TriggerBuild* method and associate this program with the checkin triggers of your version control system.

2. In the project dependency use case, build of *componentA* will be triggered (depends on the *build necessary condition*, a build may or may not be generated in *componentA*) before triggering *projectA* (because *projectA* uses artifacts of *componentA*, and thus depends on it), However if *componentA* is triggered first, *projectA* will not be triggered automatically. But you can achieve this by writing a program calls the *TriggerBuild* method in the API and run this program just before end of *componentA*'s build script.

3. Backup QuickBuild database to XML regularly.

4. Create configuration programmatically.

5. Sometimes it is necessary to delete unwanted builds (particularly generated by continuous integration configurations) to save space. QuickBuild have two built-in build clean up strategies: *reserve builds by days*, and *reserve builds by count*. However if they do not suffice your particular needs, you can write your own program to periodically delete selected builds from the system.

6. Create users programmatically. This is useful to import large number of users from other systems.

7. Get information on last successful build of a particular configuration, and feed some of them (such as build version number) into other systems such as an issue tracking application.

8. Get information on build of particular version. This is useful if you want to access information for a particular build from other systems such as an issue tracking application.

All information about the API is available in the `api` directory under your QuickBuild installation. Within this directory:

- `docs` sub directory contains the API reference.

- `lib` sub directory contains necessary jars to compile and run your program.

- `samples` sub directory contains some samples demonstrating how to use the API.

Chapter 10. Use cases

This chapter shows how to use QuickBuild by going through a couple of typical use cases. All use cases are using CVS as sample repository. You can also use other type of repository here. Configurations used in these use cases resides under configuration *root.live-samples* which will be available after you've installed QuickBuild. You need to edit basic settings of the *root* configuration, and set value for variable such as *ant*, *cvs*, *maven* according to your environment. These variables will be referenced in child configurations.

Note

Most of the samples are configured to check out codes from our sample repository *:pserver:anonymous@cvsdemo.pmease.com:/home/cvsroot*. It will be very helpful to check out these codes and look into them.

General use cases

Working with your first build

Check out code from CVS, build them with Ant, and tag the source when build is successful. Also upon a failure build, notification should be sent to users who have checked in codes recently.

1. From the dashboard, trigger the configuration *root.live-samples.sample1* , the *Editing manual trigger settings* page will be displayed. For property *Build necessary condition*, choose menu item *Force build* from the drop down menu. This will force a build even there are no changes in CVS repository since last build. Click on OK, and the build should be kicked out and run. After some time, refresh the dashboard page; you'll see the build has finished. If all things go well, this should be a successful build. Click on status icon of the newly generated build, the build log will be displayed. In case the build cannot be generated, you can click on status icon of the configuration, and examine what has happened in the configuration log.

2. You now have successfully run the first build. Let's examine some important aspects for this configuration.

- Basic settings

  This page shows some basic settings of the configuration. You can set working directory and publish directory for this configuration. Working directory is used to store stuffs specific to current configuration, for example, configuration logs. In this working directory, a directory named *checkouts* will be created to hold stuffs checked out from configured repositories. In the configuration's publish directory, a directory named *builds* will be created to hold generated builds, including build logs, published artifacts, etc. If these two directories are set as empty value, they will inherit settings from parent configuration. If you specify a relative path, it is assumed that they are relative to the parent's working or publish directory.

- Repositories

  Create the CVS repository in this page. Pay particular attention to the *CVS executable path* property which we assign it with the value *${var["pathToAnt"]}* here. The expression embedded in *${...}* is OGNL expression. Almost all text properties in QuickBuild can embed OGNL expressions as long as they are surrounded by *${...}*. Root of the OGNL expression is always current configuration object. Here we are referring to *var* property of the configuration object. For properties that can be used in your OGNL expression, please refer to OGNL reference section.

  Multiple repositories can be defined. Particularly, for the newly created repository, if you choose a name that is the same as another repository defined in ancestor configurations, newly created repository will override previously defined one. This is also true for builders and steps. So you can define common objects in ancestor configurations, and define particular objects specific to particular child configurations in descendent configurations. In this way, your life of configuring new builds is made simple.

- Builders

  Create the Ant builder in this page. Here again we use OGNL expressions in *Path to Ant executable* and *Build properties*. Take a look at *Build properties*, we passed several properties to Ant. Among them, *buildVersion* is used to pass in current build's version; *artifactsDir* is used to pass in the directory path to which your build results should be copied. This directory is a sub directory of current build's publish directory, and is named as *artifacts*. Current build's publish directory is a sub directory named by its version under current configuration's publish directory. *junitHtmlReportDir* is used to pass in directory where you should store your JUNIT html reports. *cloverHtmlReportDir* is used to pass in directory where you should store your Clover html reports. Of course you can choose property names other than *artifactsDir* or *buildVersion*, as long as you refer to these property names in your Ant build script file.

  Multiple builders can be defined. Particularly, for the newly created builder, if you choose a name that is the same as another builder defined in ancestor configurations, newly created builder will override previously defined one.

  Let's take a look at what should be done in your Ant build script file. After you have successfully run the build, change to directory `<QuickBuild installation directory>\working\root\live-samples\sample1\checkouts`, and you'll find checked out stuffs from the configured CVS repository. Change to directory `sample1\build`, and open file `build.xml`; pay attention to the target *distribute*. This target creates distribution file under the directory denoted by property *artifactsDir*. Also in target *test*, we generate JUNIT html report into the directory denoted by property *junitHtmlReportDir*; in target *cloverreport*, we generate Clover coverage report into the directory denoted by property *cloverHtmlReportDir*. In this way, we can access build results, JUNIT html report, and Clover html report from QuickBuild's web interface.

  ### Note

  There are two methods to make build results accessible from QuickBuild's user interface:

  a. Save build results under directory denoted by *artifactsDir* property. This is what we do in this use case.

  b. Put build results in any directory you want, and use the publish step to create soft links under directory denoted by *artifactsDir* property. These soft links will point to your build results. This is what we do in next use case.

  ### Note

  Clover html report will only be generated if you have installed Clover.

- Notifiers

  Create desired notifiers in this page. Here we defined an Email notifier. This notifier will be referred to when define the notification step, which is used to send failure build notification to users who has checked in since last successful build. Message title and body for this notifier can be customized through using Velocity template. Notification sent out using this notifier will contain links to affecting configuration, build, build log, revision log, and also several lines arrounding the error line inside the build log.

  ### Note

  You can define your notifiers in a high level configuration, so that they can be used by every descendent configuration without need of re-definition.

- Steps

  Create desired steps in this page. Here we've defined four steps, *check out from cvs*, *build with ant*, *create label for successful builds*, *send notification for failed builds* and *default*, respectively. Pay particular attention to the *default* one. Actually, When the configuration got running, QuickBuild will only locate and execute the *default* step (will look for this step in ancestor configurations if not found in current configuration. The same is true of other steps). As you may have noticed, the default step is a serial composite step that will trigger other three steps one by one.

- Login mappings

  This page is used to map repository logins to QuickBuild users. As you see in steps definition, upon a failed build, the *send notification for failed builds* step will collect CVS logins who has checked codes into CVS repository since last successful build, and send notifications to them. Before send this notification, these logins should be mapped to corresponding QuickBuild users in order to get contact information such as Emails, MSN messenger accounts, etc. When define repositories, you can refer to these login mappings, so that logins in those repositories can be resolved to correct users in QuickBuild system.

  ### Note

  You can define your login mapping objects in a high level configuration, so that they can be used by every descendent configuration without need of re-definition.

- Child configurations

  Create child configurations under current configuration. Currently it is the only place to create, delete or move configurations.

Working with Maven

Check out code from CVS, and build them with Maven. Instruct Maven to use version managed by QuickBuild, and make artifacts published to Maven repository also accessible from QuickBuild's web interface.

1. Trigger configuration *root.live-samples.maven-sample* with *force build* option. QuickBuild should check out code from CVS repository, trigger the maven builder. During the build, Maven publishes generated artifacts to `/maven-repository` (configured in file `<current configuration's checkouts dir>/maven-sample/project.properties`). Of course, you can change local repository to any other directory if you like, but please **do not** check them into QuickBuild's demo CVS, in order not to disturb other persons using this live sample.

   ### Note

   Do not forget to set property *path to maven executable* in basic settings of the *root.live-samples* configuration.

2. You now have successfully gotten the build running with Maven integration. On build detail page, you should see several files are listed there as published artifacts. Please note that these files are actually located in Maven's repository. They appear here because soft links to them are created under artifacts directory of current build. This was done by the step of type *publish* (see steps tab of this configuration). In this step, we specify `/maven-repository/maven-sample/jars` as source directory for publishing, and specify *maven-sample-${build.version}.** as the file name pattern. Before this publish step runs, OGNL expression in this pattern will be replaced with current build's version, for example, *1.0.3*, and the publishing file name pattern will actually be *maven-sample-1.0.3.**, which means all files under the source directory with file name starting with *maven-sample-1.0.3* will be published into artifacts directory of current build, that is, soft links to these files have been created under artifacts directory.

   ### Note

   If your build version contains spaces, you should surround the pattern *maven-sample-${build.version}.** with quotations; otherwise, it will be treated as multiple patterns separated by spaces.

3. Also Maven is using version number managed by QuickBuild as current version. This is done in two steps:

   - When define Maven builder, pass in a property named by *buildVersion* (You can choose other name of course), and set its value to be *"${build.version}"*. Here quotes are used in case that evaluated OGNL expression contains white spaces.

- In the project definition file (`<checkouts dir>/maven-sample/project.xml` here), instruct Maven to use value of passed in *buildVersion* property as current version:

  ```
  <currentVersion>${buildVersion}</currentVersion>
  ```

4. In case of Maven2, please refer to configuration *root.live-samples.maven2-sample*.

Working with project dependencies

Some of my products depend on common components. For flexibility, separate build for these components have been set up. Before build those products, common components should be checked first to see if they need to be built.

1. Let's assume that we have two Java projects: *productA* and *componentA*. *productA* uses build result (`componentA-xxx.jar`) of *componentA*, and thus depends on *componentA*.

2. Create configuration for *componentA*, say *root.componentA*. This configuration checks out code for *componentA* from CVS repository, build with Ant and generate `componentA-xxx.jar` into current build's artifacts directory. This Jar file is needed by *productA*.

3. Create configuration for *projectA*, say *root.productA*. Set up the following things for this configuration:

- Create below repositories:

repository1	This repository is a CVS repository which is used to check out source codes of productA from CVS.
repository2	This repository is a QuickBuild repository which is used to check out `componentA-xxx.jar` from latest build of configuration *root.componentA*. Modules information are set up so that componentA-xxx.jar will be put into directory `<configuration root.productA's checkouts directory>/productA/componentA`.

- Create a Ant builder to build productA. In the build script file, it uses Jar file generated by *componentA*

- Create below steps:

check out productA from CVS	This step uses *repository1* to check out source codes of productA from CVS.
check out build results of componentA	This step uses *repository2* to check out `componentA-xxx.jar` from CVS.
create label	This step does the following things: ? Creates a label on source code of productA in CVS. ? Creates a label on *root.componentA* to mark the build number of *componentA* whose artifacts are used by this version of *productA*.
default	This step is a serial composition step that runs the above five steps serially.

4. In this way, *productA* is set up to be dependent on *componentA*. When *root.productA* is triggered, *root.componentA* will also be triggered to see if it is necessary to be built, and build results of *componentA* will be checked out to work space of *productA* to accomplish *productA*'s build.

Warning

When you set up a configuration to be dependent on another configuration, you should never let them share the same working directory. Otherwise, deadlock will happen. See here for the reason.

5. A live demo is available at *http://livedemo.pmease.com:8081*. Within this live demo:

   - *root.live-samples.sample4.productA* represents *root.productA* we talked about.
   - *root.live-samples.sample4.componentA* represents *root.componentA* we talked about.

Working with multiple branches

I want to set up build for multiple branches of my product. I do not want to input repositories or builders information multiple times for each branch, how can we do that?

1. Let's assume that you want to set up build for two branches for your product: *bugfix* and *main*. First set up a configuration for your product, say *root.product1*. In this configuration, set up informations such as repositories, builders, and steps. Particularly, for *branch* property of your repository, set it to be: *${var["branch"]}*. This means that value of *branch* variable defined in your configuration will be used as actual branch to check out.

2. Create configuration *bugfix* under *root.product1*. For this configuration, you only need to set up *next build version* and define the following variable:

   ```
   branch=bugfix
   ```

   In this way, configuration *root.product1.bugfix* is set up to build against *bugfix* branch of your repository.

3. Create configuration *main* under *root.product1*. For this configuration, you only need to set up next build version and define the following variable:

   ```
   branch=
   ```

   In this way, configuration *root.product1.main* is set up to build against main branch of your repository.

   ### Note

   An empty value for *branch* means main branch.

4. A live demo is available at *http://livedemo.pmease.com:8081*. Within this live demo:

   - *root.live-samples.sample6.bugfix branch* represents *root.product1.bugfix* we talked about.
   - *root.live-samples.sample6.main branch* represents *root.product1.main* we talked about.

Working with build promotion

For a particular project, I want to set up nightly build, test build, and release build respectively. Nightly builds should not tag the source in any case, while test and release builds should tag the source whenever there is a successful build. Further more, for particular test build that has passed QA tests for example *"myproduct-QA-5"*, I want to promote it as a release build, and assign it with a formal version, for example *"myproduct-1.0.5"*

1. Open configuration *root.live-samples.sample3*. This is the top level configuration for our sample project. We define necessary repositories, builders, and steps in this configuration.

2. Under this configuration, we defined three child configurations *nightly*, *test* and *release*, respectively. For test and release configuration, the only thing need to do is setting proper *next build version* in the basic settings tab. Other objects are inherited from parent, including repositories, builders and steps. For nightly configuration, the label step is not wanted, so we create a new step in the steps tab, and choose *overriding 'Label CVS'* from the drop down menu of the *name* property, and choose *never run this step* as value for the *step necessary condition* property. In this way the *label CVS* step created in parent configuration has been overriden, and will never get chance to run. Another way to avoid labeling is to override the default step, and remove the *label CVS* step from the serial composition.

Note

For release and test builds, it is highly recommended to set them as clean builds. For nightly builds, be set as increment builds will speed up the build process. Generally speaking, clean build is more reliable, and increment build is faster.

3. Now try to forcibly trigger these three child configurations, you can see all of them should run happily. By examining builds in these three child configurations, you'll find that builds in *release* and *test* configuration can be promoted and rebuilt, while builds in *nightly* configuration can not. The reason is that rebuild and promotion need to repeat the build, which is only possible when labels were created in repository for these builds.

4. Now suppose that our QA team has thoroughly tested build *myproduct-QA-5*. We are satisfied with its quality, and want to promote it as a release build. In order to do this, just go to detail page of *myproduct-QA-5*, click on promote button, the *Edit promote settings* page will appear. In this page, choose *root.live-samples.sample3.release* as the destination configuration, and click on OK. Now *myproduct-QA-5* is being promoted in *release* configuration. The new version will be *next build version* of release configuration, which we assume as *myproduct-1.0.5* here. After the promotion, original test build will be deleted, and the new build will create a new label called *myproduct-1_0_5* in CVS.

Note

Under the hood, promotion retrieves sources from repository with label of original build, and go through steps defined in destination configuration to perform the new build. So in order to repeat the original build exactly, source configuration and destination configuration should use the same set of repositories and builders during build.

Note

Another way to perform promotion is just to move a build from one configuration into another, this will not trigger a new build. The limitation is that the version attached original build can not be changed.

Sharing working directories

In above use case, every child configuration of "*root.live-samples.sample3*" has its own working directory and holds its own copy of checked out codes from CVS. For a large project, this may consumes a lot of disk spaces. Is there any way to check out only one copy of codes for these three child configurations?

1. By default, a newly created configuration will use *${name}* as value of the *working directory* property. This means new configuration will create a sub directory under working directory of its parent configuration, and name of the sub directory will be the same to name of newly created configuration. So there will be three different working directories for *nightly*, *test* and *release* configuration. In order to use the same working directory for these three child configurations, the simplest way is to left their *working directory* property as empty. In this way, they will all use parent configuration's working directory. Of course, you can point them to other arbitrary directories, as long as they all refer to the same directory.

2. Now *nightly*, *test* and *release* configurations have the same working directory, and there will be only one copy of codes checked out for build, which resides in `<working directory>/checkouts`.

Note

For configurations sharing the same working directory, only one can be executed at one time. If you trigger others while one configuration is already running, the newly triggered configuration will be put into queue, until the current one has finished its execution.

Note

If multiple configurations share the same working directory, and some of them are configured to incremental build, it is highly recommended that all these configurations check out the same set of source code, build with the same set of builders. Otherwise, increment builds may be incorrect for codes may be incremental updated based on different code base.

Sharing build versions

In use case working with build promotion, I want all child configurations share the same stream of build version. That is, if the most recent build (among all child configurations) takes version "*myproduct-1.0.1 build 4*", then the next happened build should take version "*myproject-1.0.1 build5*" in regardless of its belonging configuration.

1. In basic settings tab of configuration *root.live-samples.sample3*, set a value for property *next build version*, for example: *myproduct-1.0.1 build 1*.

2. For all child configurations under *root.live-samples.sample3*, set an empty value for property *next build version*. In this way, all child configurations will inherit *next build version* from the parent configuration, that is, share the same stream of build version. If *next build version* of the parent configuration is also empty, it will inherit the value from its own parent, until non-empty value has been found, or reaches the root configuration.

Using date and iteration as part of build version

In use case working with build promotion, I want current date and iterations of current date can be embedded into build version of child configurations.

1. Let's take *test* child configuration as example. From the drop down menu of property *next build version*, choose *date and iteration*, a complicated value contains quite a lot of OGNL expressions will be set for *next build version*. It will generate versions like *2005-Sep-25.4* as default, where *2005-sep-25* indicates date of the build, and *4* here means iterations in this date.

2. If you are not satisfied with the format of this default one, you can modify the value of *next build version*. Before doing this, make sure you know the grammar of OGNL expressions [???] as well as exposed date and time properties in QuickBuild. For example, you can add the string *myproduct-QA-* at the very start of *next build version*, then, the version generated will like *myproduct-QA-2005-Sep-25.4, myproduct-QA-2005-Sep-25.5*, etc.

   ### Note

   When you choose *date and iteration* as next build version, and run the configuration, QuickBuild will automatically put two variables in current configuration, viz. *day* and *dayIterator*. For variables that have not been defined (either in current configuration or in ancestor configurations), QuickBuild will assume it has an empty value when referenced as string, or 0 when referenced as number. And QuickBuild will automatically create these variables in current configuration if they have been assigned values.

Managing major, minor, and iteration part of a version string

In use case working with build promotion, I want to define a version schema with major release part, minor release part and iteration part. Major release part will be set manually. The "*release*" configuration increases minor release part of the version, while "*test*" configuration increases iteration part of the version. When minor release part of the version changes, iteration part should be reset to 1.

1. Define the following variables in configuration *root.live-samples.sample3* :

   ```
   majorRelease=myproduct-1.0
   minorRelease=1
   iteration=0
   ```

2. Define *next build version* of *root.live-samples.sample3.test* configuration as:

   ```
   ${var["majorRelease"]}.${var["minorRelease"]} iteration
   ${var["iteration"].increaseAsInt()}
   ```

3. Define *next build version* of *root.live-samples.sample3.release* configuration as:

   ```
   ${var["majorRelease"]}.${var["iteration"].setValue(1),
   var["minorRelease"].(increaseAsInt(), value)}
   ```

4. In this way, builds in *root.live-samples.sample3.release* will get versions like: *myproduct-1.0.1*,

myproduct-1.0.2, myproduct-1.0.3, ..., and builds in *root.live-samples.sample3.test* will get versions like: *myproduct-1.0.1 iteration 1, myproduct-1.0.1 iteration 2, myproduct-1.0.1 iteration 3, ..., myproduct-1.0.2 iteration 1, myproduct-1.0.2 iteration 2, ...*

Specifying label to build against

I've set up a configuration to build against latest code of my project. But, when build this project manually, I want to be able to specify the label to build against. Is there anyway to do this?

1. Open configuration *root.live-samples.sample5*, and check modules definition of its CVS repository setting, you'll find that label value of source path *sample1* has the value of *${var["label"]}*. And in the basic settings tab of this configuration, a variable *label* was defined with an empty value like this:

   ```
   label=
   ```

   It means that this configuration will still build against latest code unless you specify a non-empty value for the variable *label*.

2. Now forcibly trigger this configuration, in the appeared *Editing manual trigger settings* page, provides a different value for *label* variable like this:

   ```
   label=myproduct-1_0_0
   ```

   In this way, configuration *root.live-samples.sample5* is triggered to build against label *myproduct-1_0_0*.

   ### Tip

   By using variables, you can make almost any part of repositories, builders, or steps definition be overridable when manually triggers the build. Also it is possible to override these variables in child configurations, which gives you the flexibility to modify part of objects defined in ancestor configurations.

Updating information of many projects

I have a CVS server with many projects, and set up dozens of configurations for these projects in QuickBuild. After sometime, server name (or ip address) of our CVS server has been changed. So I need to go through all defined repositories and change server name (or ip address) accordingly. Is there any simple way to do this, or handle such kind of batch processing?

1. Define a variable for example *cvsServerName* in a high level configuration (a proper candidate for this high level configuration can be your department or team's root configuration), and set its value as server name of ip address of your CVS server.

2. Refer to the above variable when define *CVS root* of your repositories, for example:

   ```
   :pserver:build@${var["cvsServerName"]}:/cvsroot
   ```

   Now if you want to point all your CVS repositories to the new CVS server, simply modify value of the variable *cvsServerName*.

   ### Tip

   It is a good idea to extract dynamic parts (maybe changed frequently) of repositories, builders and steps, and then put them as variables in higher level configurations. In this way, you can easily change property of all affected objects.

Working with build queues

I have set up a number of configurations for projects of deparment1, and all of them are under configuration "*root.department1*". Now I want to make sure that at the same time, only two builds can be performed for this department in order to save CPU resources for other departments.

1. Create a queue, say *queue_department1*, with two working threads.

2. Edit *root* configuration, and set the build queue as *queue_department1* (This step is necessary that, *queue_department1* will still be used even administrator of *root.department1* set build queue as *inherit from parent*).

3. For users of department1, assign them to groups with only *queue_department1* authorized.

4. In this way, subtree under *root.department1* are limited to only use *queue_department1*.

Working with public configurations

Make artifacts of some projects be publicly accessible, but do not want these projects be modified or built publicly.

1. Set up public accessible projects under a particular configuration, for example, *root.public*.

2. Add a group named by *anonymous*, and configure this group to have *View* permission on configuration subtree rooted at *root.public*.

3. In this way, anonymous users can only access configurations under *root.public*, without the permission to build or edit these configurations.

Remote and parallel use cases

Building multi-platform products

product1 comprises of *component1* and *component2*. *component1* should be built on Windows, while *component2* should be built on Linux. After these components have been built, they should be collected together on Windows platform and packaged into *product1*.

Warning

In order to make remote build work, you should keep time of different build machines in sync (Normally a slight difference within 5 seconds is tolerable).

1. Install QuickBuild on Windows and Linux. Create configuration *root.product1* in QuickBuild system running on Windows (refer as *root.product1@Windows* later), and create configuration *root.component2* in QuickBuild system running on Linux (refer as *root.component2@Linux* later). *root.product1@Windows* will be used to build component1 and package component1 and component2 into product1, while *root.component2*@Linux will be used to build component2.

2. *In root.component2@Linux*, create appropriate repository, builder and steps so that it can check out source code of component2 from CVS, build component2, and create label on component2 in CVS if build is successful. Artifacts of component2 will be published into some directory of generated build. Do a test on this configuration to make sure it works.

   ### Note

   root.component2@Linux will maintain separate build versions from *root.product1@Windows*. This is reasonable, cause it's the common case that a product comprises of multiple components of different and independent version numbers.

3. In configuration *root.product1@Windows*:

   - Create below repositories:

repository1	This is a CVS repository which is used to check out source codes of component1 from CVS.
repository2	This is a QuickBuild repository which is used to check out build artifacts of component2

from latest build of configuration *root.component2@Linux*.

- Create two builders: *builder1* and *builder2*. *builder1* is used to build component1, and *builder2* is used to package artifacts of component1 and component2 into product1.
- Create below steps:

check out component1 from CVS	This step uses *repository1* to check out code of component1 from CVS
build component1	This step uses *builder1* to build component1
retrieve component2 artifacts	This step uses *repository2* to check out artifacts from latest build of component2 at Linux box. **Note** By defining this step, *root.product1@Windows* is considered to be dependent on *root.component2@Linux*, and *root.component2@Linux* will be triggered automatically during the dependency resolving phase, which is happened before retrieving component2 artifacts.
package artifacts of component1 and component2	This step uses *builder2* to package artifacts of component1 and component2 into product1
create label	This step does the following things: ? Creates a label on source code of component1 in CVS. ? Creates a label on *root.component2@Linux* to mark the build number of component2 whose artifacts are packaged into this version of product1.
default	This step is a serial composition step that runs the above five steps serially.

4. In this way, product1 can be built easily. Further more, If product1 contains component3 which should be built on Solaris, you can install another QuickBuild system on a Solaris machine, configure it to be able to build component3, and add corresponding QuickBuild repository to check out artifacts from component3 at Windows side.
5. A live demo is available through QuickBuild's demo site *http://livedemo.pmease.com:8081/*. Within this live demo:
   - *root.remote-builds.product1* stands for *root.product1@Windows* we talked about.
   - *root.remote-builds.LinuxBox.component2* simulates the configuration *root.component2@Linux*.

Working with parallel builds

My product contains multiple components, which can be built independently. Is there any way to build them simultaneously on multiple machines to speed up the build process?

1. In order to demonstrate this ability, we will take the example cited in use case Building multi-platform products, but try to build component1 and component2 simultaneously. Nothing needs to be changed at *root.component2@Linux* side. At *root.product1@Windows* side, make the following changes:
   - Add a repository say *repository3*. This repository is defined so that it contains source path of both component1 and component2.

   - Change *build necessary condition* to be *repository["repository3"].modified* at *basic settings* tab of current configuration. The reason for not using *effectingRepositoriesModified* is that: By using *effectingRepositoriesModified*, when decides whether or not a new build is necessary, QuickBuild will examine contents of every repository referenced in step definitions to see if they have been modified. During this phase, *root.component2@Linux* specified in *repository2* may be triggered and built (and contents of *repository2* are considered to be modified if latest build of component2 happens after latest build of product1), which is not we want. By using *repository["repository3"].modified* instead, we ignores *repository2* during build necessary condition evaluation phase of product1, but still be able to detect changes in both component1 and component2.

   - Create the following steps:

Check out component1 from CVS	This step uses *repository1* to check out code of component1 from CVS.
Build component1	This step uses *builder1* to build component1.
Check out and build component1	This step is created to execute the above two steps serially.
Retrieve component2 artifacts	This step uses *repository2* to check out artifacts from latest build of component2 at Linux box. This step will cause component2 been built if necessary (depends on *build necessary condition* of *root.component2@Linux*).
Prepare artifacts of component1 and component2	This step is a parallel composition of step "*checkout and build component1*" and "*retrieve component2 artifacts*". By using this step, component1 and component2 will be built simultaneously.
Package component1 and component2 into product1	This step packages prepared artifacts of component1 and component2 into product1.
Create label	This step does the following things: ? Creates a label on source code of component1 in CVS. ? Creates a label on *root.component2@Linux* to mark the build number of component2 whose artifacts are packaged into this version of product1.
default	This step is a serial composition of step "*prepare artifacts of component1 and component2*", "*Package component1 and component2 into product1*", and "*create label*".

2. A live demo is available through QuickBuild's demo site *http://livedemo.pmease.com:8081/*. Within this live demo:

   - *root.remote-builds.product1-parallel* stands for *root.product1@Windows* we talked about in this use case.

   - *root.remote-builds.LinuxBox.component2* simulates the configuration *root.component2@Linux*.

Performing automation/smoking tests on a machine other than build machine

After my product has been built, it should be sent to another machine to run smoking/automation tests. If tests pass, mark current build as successful; otherwise, mark it as failed.

1. Take "*product2*" as example, we create a configuration *root.product2* at machine1 to build this product. In this configuration, set up the following things:

- Create two repositories:

repository1	This repository is created to check out source code of product2 from CVS.
repository2	This repository is created to retrieve test log from latest build of configuration *root.test-product2@machine2*. In this repository, we define a module with *source path* be "..", and *file name pattern* be "*build_log.txt*". In this way, test log will be retrieved (Note that source path is relative to artifacts directory, so we use ".." to change to parent directory of artifacts, which includes the build log file).

- Create two builders:

builder1	This builder is used to build product2.
builder2	This builder is used to publish test log into some sub directory of artifacts directory, so that it can be accessed from web interface.

- Create five steps:

check out from CVS	This step uses *repository1* to check out source code of product2.
build with Ant	This step uses *builder1* to build product2.
retrieve test results	This step uses *repository2* to check out test log of product2. At this point, you may curious about how build result of product2 is sent to machine2. The trick is at *root.test-product2@machine2* side. In that configuration, we will define steps to pull latest build result of product2.
publish test results	This step uses *builder2* to publish test log. *Step necessary condition* property of this step should be set to *true* in order to publish the test results for review even the step "*retrieve test results*" fails.
default	This step executes the above four steps one by one.

2. Another configuration *root.test-product2* needs to be created at machine2 to run test against product2. In this configuration, set up the following:

    - Set value of property *build necessary condition* as *true*, which means new build will always be generated upon triggering. This is needed because this configuration will be dependently triggered by *root.product2@machine1*, and for each dependent triggering, we want new build of *root.test-product2* been generated so that tests can be run.

    - Create two repositories:

repository1	This repository is created to check out build result of product2 from latest build of configuration *root.product2@machine1*.
repository2	This repository is created to retrieve test script from CVS.

    - Create a builder *builder1*, which will be used to run test script.

    - Create four steps as below:

check out test scripts	This step uses *repository2* to check out test scripts from CVS.
check out build results of product2	This step uses *repository1* to check out build results of product2.
run test against product2	This step uses *builder1* to run test.

default	This step executes the above steps one by one.

3. In this way, build result of product1 will be tested on machine2, and test results will be collected back to machine1. If test fails (that is, build of *root.test-product2@machine2* fails), step "*retrieve test results*" at *root.product2@machine1* side will also fail, which causes build of product2 failed.

   ### Note

   As you may noticed, *root.product2@machine1* depends on *root.test-product2@machine2* to retrieve test results, and *root.test-product2@machine2* depends on *root.product2@machine1* to retrieve product2 build results. Obviously there is a dependency loop here. QuickBuild is clever enough to handle this correctly:

   - When a configuration is triggered actively (that is, triggered by user or schedule, instead of dependency resolver), other configurations detected in a dependency loop is considered as subordinate configurations which will not be taken into account when evaluating *build necessary condition* of the active configuration. In our case, *root.product2@machine1* is triggered actively, and *root.test-product2@machine2* plays the role as subordinate configuration.

   - At *root.product2@machine1* side, when step "*retrieve test results*" executes, it will wait until the newly generated build at *root.test-product2@machine2* side completes (either successful or failed), and then downloads test results from that build.

   - At root.test-product2@machine2 side, when step "*check out build results of product2*" executes, it knows that it is a subordinate configuration, and will download build results of product2 right away from *root.product2@machine1* without waiting for build of *root.product2@machine1* been finished. Otherwise, deadlock will occur. Consequently, you should make sure build results of product2 are available for retrieving before step "*retrieve test results*" gets running in configuration *root.product2@machine1*.

   ### Warning

   You should never actively trigger a subordinate configuration. Otherwise, deadlock may occur.

4. A live demo is available through QuickBuild's demo site *http://livedemo.pmease.com:8081/*. Within this live demo:

   - *root.remote-builds.product2* stands for configuration *root.product2@machine1* we talked about.

   - *root.remote-builds.LinuxBox.test-product2* simulates configuration *root.test-product2@machine2*.

REST API use cases

Set up real-time continuous integration build

Set up a continuous integration configuration, and want build of this configuration be triggered whenever there is a checkin made into the version control system.

1. Take configuration *root.realtime-CI* for example. Let's assume this configuration checks out module *realtime-CI* from CVS repository and builds with Ant.

2. Modify the TriggerBuild.java sample, so that it triggers build of configuration *root.realtime-CI*. Of course, you also need to change the login information, and hessian service URL. Compile this program with jars under api/lib directory, and write a script `triggerbuild.sh` to run this program with Java.

3. Checkout "`loginfo`" file under CVSROOT directory of your CVS repository, and append a line like this:

   ```
   realtime-CI /path/to/triggerbuild.sh
   ```

Note

Before editing, the file "`loginfo`" should be checked out first using your cvs client, just like you edit other files in your cvs repository.

4. Check in the "`loginfo`" file. From now on, the checkins under CVS module "*realtime-CI*" will trigger the triggerbuild.sh command, which will result in triggering build in configuration *root.realtime-CI*.

5. A live demo is available through QuickBuild's demo site *http://livedemo.pmease.com:8081/*. Within this live demo, *root.api-samples.realtime-CI* stands for configuration *root.realtime-CI* we talked about. Just connect to CVS repository at *:pserver:anonymous@cvsdemo.pmease.com/home/cvsroot*, and make some checkins into module *realtime-CI*, you'll see configuration *root.api-samples.realtime-CI* will be triggered and running to ensure health of the code base.

Trigger other builds after build of particular project

productA depends on *componentA*. After *componentA* finishes build, *productA* should be built to take the most up-to-date artifacts of *componentA*.

Note

This scenario can not be addressed through project dependency use case, because the dependency mechanism only guarantees that build of *projectA* can trigger build of *componentA*, but not vice versa.

1. Let's assume configuration *root.projectA* depends on *root.componentA*, and *root.componentA* uses Ant to perform build.

2. Modify the TriggerBuild.java sample, so that it triggers build of configuration root.projectA. Of course, you also need to change the login information, and hessian service URL. Compile this program with jars under api/lib directory.

3. Call the following task before end of *componentA*'s Ant build script:

```
<java classname="TriggerBuild">
  <classpath>
     <pathelement path="<the directory which contains TriggerBuild.class>"/>
     <pathelement location="/path/to/hessian-3.0.8.jar"/>
     <pathelement location="/path/to/quickbuild-api.jar"/>
  </classpath>
</java>
```

4. A live demo is available through QuickBuild's demo site http://livedemo.pmease.com:8081/. Within this live demo:

   - *root.api-samples.productA* stands for configuration *root.productA* we talked about.
   - *root.api-samples.componentA* stands for configuration *root.componentA* we talked about.