

uBuild® User Guide

4.2.2

uBuild User Guide: 4.2.2

Publication date March 2013

Copyright © 2013 UrbanCode, Inc.

UrbanCode, AnthillPro, uBuild, uDeploy and any other product or service name or slogan or logo contained in this documentation are trademarks of UrbanCode and its suppliers or licensors and may not be copied, imitated, or used, in whole or in part, without the prior written permission of UrbanCode or the applicable trademark holder. Ownership of all such trademarks and the goodwill associated therewith remains with UrbanCode or the applicable trademark holder.

Reference to any products, services, processes, or other information, by trade name, trademark, or otherwise does not constitute or imply endorsement, sponsorship, or recommendation thereof by UrbanCode.

All other marks and logos found in this documentation are the property of their respective owners. For a detailed list of all third party intellectual property mentioned in our product documentation, please visit: <http://www.urbancode.com/html/company/legal/trademarks.html>.

Document Number: 4.2.2

About	1
How This User Guide is Organized	1
Product Support	1
Document Conventions	1
Introduction	3
uBuild Introduction	4
uBuild Concepts	4
Projects	4
Build Processes	4
Source Configurations	4
Source Repositories	5
Dependencies	5
Templates	5
Workflows	5
Build Lives	5
Secondary Processes	6
Architecture	7
Service Tier	8
Data Tier	9
Relational Database	9
CodeStation / File Storage	9
High Availability	10
High Availability Configuration	10
Agents	11
Server-Agent Communication	12
Crossing Network Boundaries and Firewalls	13
Mutual Authentication	14
Getting Started	15
Installation and Upgrade	16
Installation Recommendations	16
System Requirements	17
Server Minimum Installation Requirements	17
Recommended Server Installation	17
Agent Minimum Requirements	18
Download uBuild	18
Database Installation	18
Installing with Oracle	18
Installing with MySQL	19
Installing with Microsoft SQL Server	20
Server Installation	21
Installing the Server	21
Agent Installation	23
Installing an Agent	23
Upgrading uBuild	24
SSL Configuration	25
Configuring SSL Unauthenticated Mode for HTTP Communications	25
Configuring Mutual Authentication	26
Running uBuild	27
Running the Server	27
Running an Agent	27
Accessing uBuild	28
Stopping the Server	28
Stopping an Agent	28
Quick Start Tutorial—Creating a Workflow	29

Before Starting	29
Create a Git Repository	30
Create a Workflow	31
Create a Job	32
Assign the Job to the Workflow	39
Create a Template	40
Source Templates	40
Configuring Build Process Templates	41
Create a Project	42
Run the Build	44
Review	45
Reference	47
uBuild Plug-ins	48
Overview of uBuild Plug-ins	48
Creating Plug-ins	48
The plugin.xml File	49
Plug-in Steps--the <step-type> Element	51
The <server:property-group> Element	55
Upgrading Plug-ins	57
The info.xml File	58
System Settings	59
Installing Plug-ins	59
uBuild Scripting	60
Index	61

About

This book describes how to use UrbanCode's uBuild product and is intended for all users.

This book is available in PDF and HTML formats at UrbanCode's Documentation portal: <http://docs.urbancode.com/>. uBuild's online Help is installed along with the product software and can be accessed from the product's web-based user interface. A PDF version is also included with the product's installation package.

How This User Guide is Organized

This user guide is organized into the following parts.

Table 1. Organization of the User Guide

Part	Description
Introduction	Provides an overview of the product's concepts and describes its architecture.
Getting Started	Provides information about installation, requirements and upgrades.
Reference	Contains several reference-type chapters on topics such as scripting, and writing plug-ins, and is intended for all users.

Product Support

The UrbanCode Support portal, <http://support.urbancode.com/>, provides information that can address any of your questions about the product. The portal enables you to:

- review product FAQs
- download patches
- view release notes that contain last-minute product information
- review product availability and compatibility information
- access white papers and product demonstrations

Document Conventions

This book uses the following special conventions:

- Program listings, code fragments, and literal examples are presented in this typeface.
- Product navigation instructions are provided like this:

Configuration > Workflow > [selected workflow] > New Job [button]

This example, which explains how to add a job to a workflow, means: from the uBuild home page click the Configuration in navigation; select the Workflows tab; select a Workflow from the list; and click the New Job button.

- User interface objects, such as field and button names, are displayed with initial Capital Letters.
- Variable text in path names or user interface objects is displayed in *italic* text.
- Information you are supposed to enter is displayed in this format.

Introduction

uBuild Introduction

Enterprise software development has many unique challenges that a continuous integration server needs to address. uBuild addresses the concerns of scalability and geographical distribution that are required. Enterprise continuous integration needs to scale to the build and test demand of a large group and still maintain strict process and security enforcement across a massive amount of build configuration data.

uBuild's robust build system eliminates manual processes while providing visibility into every facet of the build process. uBuild's best-of-breed tools scale across the enterprise and provide audited processes that increase product quality while simultaneously decreasing time-to-market.

uBuild's management tools make capturing project data easy and intuitive. Key metrics are available for: compliance, governance, data aggregation and reporting, auditing/logging, and version/change control—in short, you can determine project status at any point and measure every aspect of the build cycle.

A short list of uBuild's features include:

- Build automation on schedules and triggering upon source change or promotion
- Web UI to access configuration and build data
- Remote build agents distributed builds for scalability and environmental requirements
- Configuration template for project standardization and re-use
- Dependency management system
- Extensive and open plug-in system provides built-in integrations third-party tools
- Powerful team and role-based security system to mirror how you organize development

uBuild Concepts

uBuild has a number of concepts that user's should familiarize themselves with to understand uBuild.

Projects

Projects in uBuild represent components and applications that are developed by your company. The granularity of a project is often the granularity of what can be built individually and has its own pipeline for release or distribution. Projects often correspond to how development is grouped in your source control management system. Projects may also be composites of many other projects using a dependency management system. If individual components are developed independently and then are composed into applications or products that are released, the components would be projects and would feed into projects that represent the application or product for packaging and distribution.

Build Processes

Projects contain one or more build processes. A build process is a configuration that can be executed to perform a build on some source and typically produces artifacts. A build process defines everything that happens during the build and everything that is needed to do the build. It defines where to get source and dependencies, what tools to build with and what tests to run.

Source Configurations

Build processes contain one or more source configurations. Source configurations are exactly what the sound like. They are the definition of what source to get as part of the build. They can define the branch or label to pull or to simply get the latest source. Multiple source configurations can be created on a build

process if you need to pull multiple source locations for a single build. These can span multiple repositories and be a combination of any type of source control system.

Source Repositories

Repositories are represented in uBuild so that access to them is reusable across source configurations and so that they can be secured. A repository contains the information that is required to connect to the source repository without the details of what source to get which is contained in the source configurations.

Dependencies

Dependencies can be configured on build processes to provide artifacts and build triggering between projects and/or third-party artifacts such as open-source libraries. When you create a dependency, you are saying that your build needs the dependent build's artifacts in order to run or build. Dependencies are highly configurable and you can specify criteria such as requiring a certain version of a dependency, requiring the latest that has achieved a certain promotion status. Other than artifacts, dependencies can also be used for triggering builds. This can work in two directions: down the dependency graph with pull dependencies or up the dependency graph with push dependencies. If you have a common component project that many projects use, you may want to trigger builds of the projects when the common project is changed. To do this, you configure a push dependency and when the common project build completes, the other projects that depend on it build so you can get immediate feedback if that change broke another project.

Templates

Templates are skeleton configuration that define the majority of the configuration of a project and processes. Each template represents a type of project. It encapsulates the general configuration of the source configuration and processes as well as what projects need to define to customize the template to build their project. Templates do this by defining properties that project managers provide values to when creating or editing their project using the template. For instance, a template may always invoke a particular type of build script during a build but the name or location of the build script varies across each project. The template would define a property for the build script location that each project using it will enter. There are a number of types of properties that can be configured to provide project managers with a robust interface for configuring their project. Templates allow for a very high level of configuration re-use by allowing you to decide what varies with properties.

Workflows

Workflows are the definition of what happens during a process and are reusable. Workflows define the jobs and steps that may go into a build. Templates reference workflows to provide them to projects. There are a number of built-in steps that can be used in a workflow, but the majority of steps come from plugins. Plugins are the extension point for integrating with build tools and other systems. Steps are collected into jobs and jobs are composed into workflows. When placing a job in a workflow, you choose some other options such as the agent pool its steps should run on, the working directory its steps should run in and other settings. Workflows allow you to parallelize jobs so you can split and run several different jobs at the same time and then join those paths later to run a single job. Jobs can also be iterated in a workflow which is running the same job a number of times. Each run can have its own unique set of properties. This is very useful for multi-platform parallel builds.

Build Lives

A build life is record of inputs into a build, the activities performed during the build, the artifacts produced and any activities performed on the artifacts after the build. It tracks everything from the source,

dependencies and build machine used to produce a build up to the deployment or release of the artifacts produced. All steps performed and their output are recorded. Any tests and analysis can be recorded. Defects, features and stories can be recorded. All of this information is collected together in a build life.

Secondary Processes

While a build process produces a new build, secondary processes can be executed on the build life. Secondary processes can only be executed on a existing active build life with a completed build process. They are generally used to run additional tests or analytics that you might not want to run with every build process. If one of your test suites takes a really long time, you might want to schedule it to run nightly in a secondary process.

Architecture

uBuild's architecture consists of a centralized server and distributed agents. The server is the data store, the user interface provider and the coordinator or agents. All build activities occur on agents and agents only act when told to do so by the server.

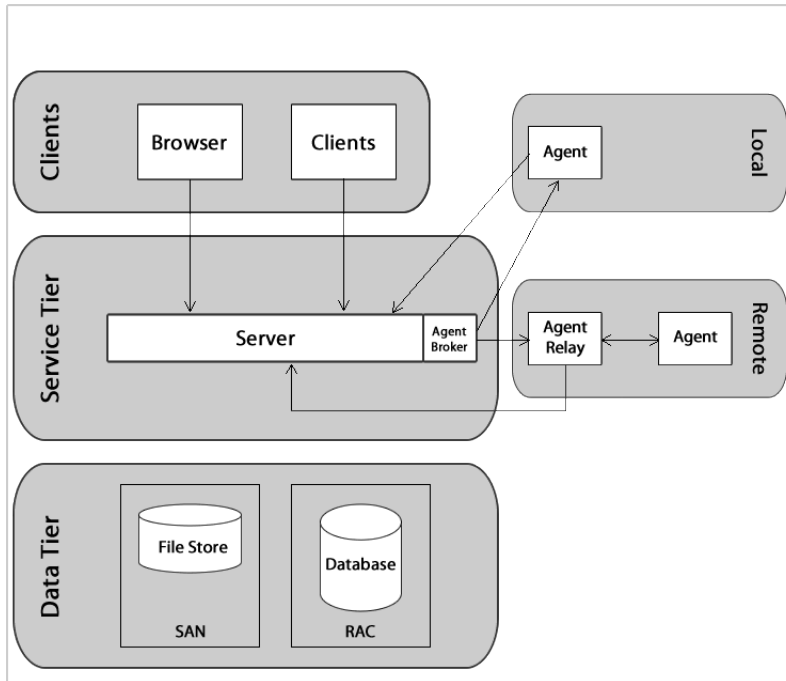
The uBuild server consists of a service tier and a data tier. The service tier has a central server that provides a web server front-end and core services, such as workflow, agent management, security, and others. A service can be thought of as a self-contained mechanism for hosting a piece of processing logic. Builds are orchestrated by the server and performed by agents distributed throughout the network. The server exposes a few interfaces. The first is a web user interface exposed through HTTP(S) that is mostly used by users with browsers. Another interface is web services using HTTP(S) and these are mostly used by agents and for other systems to integrate with. The last interface is for server-agent communication and it is done via JMS (discussed below) using TCP/IP.

uBuild's JMS server-agent communication and web services are *stateless* forms of communication. Stateless, as used here, means the server retains little session information between requests and each request contains all the information required to handle it. This allows for a communication system that performs well without a lot of resource overhead.

Server-agent communication is used to send work to agents and results back to the server. Because JMS connections are persistent and not based on a request-response protocol, uBuild does not have to continually open and close connections, which enables the server to communicate with agents at any time while remaining secure and scalable. The server listens on a port for agents to connect to it. For added security, agents do not listen on ports and merely make a connection to the server to provide 2-way communication. This simplifies firewall configuration since all connections are initiated in one direction, from the agent to the server.

Many uBuild services are exposed as REST-type (representational state transfer) web services. These services are used to send and receive information over HTTP(S). The information can be formatted in XML or JSON (JavaScript Object Notation). The REST-style services are stateless by ensuring that requests include all the data needed by the server to make a coherent response.

The data tier stores information in a relational database and on the file system. The data tier's relational database stores configuration and run-time data. The data tier's file store contains log files and build artifacts. Artifacts are stored in a system called CodeStation which is also a dependency management system. It stores the build artifacts in a content addressable storage format that minimizes storage requirement by not storing the same files more than once. By default, the file storage is within the var directory of the server installation. The location of this storage can be changed with server configuration or you could also map the directory to a network addressable storage location.

Figure 1. Architectural Overview

Service Tier

The uBuild server provides a variety of services, such as: the user interface, workflow engine, and security services, among others. The REST-based user interface provides the web-based front-end that is used to create projects and define workflows; request processes, and manage security and resources, among other things.

When a workflow is requested, many services are used to fulfill the request, which are shown in the following illustration:

Workflow requests are initiated with the user interface or a REST service.

Table 2. Services

Service	Description
User Interface	Used to manage the configuration of all items and see the current and historical runtime information of the system.
Workflow Engine	This performs the actual processing of workflows of build and secondary processes. This service also uses the Agent Manager service to acquire agents for jobs and run steps on agents. This service will also send out events about workflows starting and completing that are used by notification and dependency services for sending notification and triggering other workflows.
CodeStation	CodeStation is a dependency management system and artifact repository. CodeStation can store and process dependency configuration which enables builds to retrieve the artifacts they need at build-time. This resolution is also stored with the build life as a permanent history of what was used. CodeStation also exposes a interface as a Maven repository.

Service	Description
Agent Manager	Keeps track of agents and their offline/online status. This service also controls access to agents if a workflow needs to run a step on a agent. This also stores the properties of a agent and can be used to upgrade or restart a agent remotely.
Plug-in Manager	uBuild can interact with virtually any system through its extensible plug-in system. Plug-ins provide functions by breaking-down tool features into automated steps. Plug-ins can be configured at design- and run-time. When a plug-in step executes, the controlling agent invokes its run-time process to execute the step.
Event Service	The event service is a message bus with publishers and subscribers that provides messaging between services in a asynchronous method.
Notification Manager	Notifies users about event using notification schemes which comprise of the when, who, what and how of notifications. The when is what event triggers a notification. The who is to whom the notification is sent. The what is the content sent which is generated from templates. The how is the notification channel which could be email or instant message.
Security	This includes authentication and authorization functions. The authentication is configuration for determining who a user is and may use a internal database, LDAP, Active Directory, or a Single Sign-On service. The authorization is configuration for what a user can do. This includes information for assigning users into group and for assigning users and groups in roles.

Data Tier

Relational Database

The relational database is a critical element for performance and disaster recovery. The provided embedded database, while sufficient for proof-of-concept work, should not be used for production installations or performance testing. Full-featured databases like Oracle, MS SQL Server, or MySQL are better options. The database should be configured for high-availability, high-performance, and be backed-up regularly.

10-20 GB of database storage should be sufficient for most environments. For Oracle, an architecture based on Oracle RAC is recommended. For Microsoft SQL Server, a clustered configuration is preferred. For MySQL, utilize MySQL Cluster.

CodeStation / File Storage

The data tier also stores log file and artifact storage utilizing the file system of the server. The artifact storage system is part of CodeStation. Artifacts represent files, images, databases, configuration materials, or anything else associated with a software project that might be produced during a build or secondary process. By default, logs and artifacts are stored in the `var` sub-directory in the uBuild server installation directory. In an enterprise environment, the default installation might not be ideal, see the section called “Relocating CodeStation / File Storage” for a discussion about enterprise options.

CodeStation ensures that captured artifacts can be stored and clean up according to user-defined retention policies. It ensures artifacts are consistent and do not change once stored through the build's life-cycle. The retention policies of CodeStation also allow you to limit the storage size of artifacts while maintaining rules about how long to retain artifacts of builds that may have achieved a status indicating that they have been

released or deployed. CodeStation also provides more security than a simple network share that artifacts are dumped onto.

CodeStation uses content addressable storage to maximize efficiency of artifact transfers and minimize disk storage. Artifacts that are duplicates do not need to be transferred and are only stored in one location.

Relocating CodeStation / File Storage

By default, log files and CodeStation artifacts are stored in the `var` sub-directory within the uBuild server installation. Ideally, this data should be stored on robust network storage that is regularly synchronized with an off-site disaster recovery facility. In addition, the uBuild server should have a fast network connection to storage (agents do not need access to the storage location). In Unix environments, you can use *symbolic links* from the `var` sub-directory to network storage. On Windows platforms there are several options for redirecting logs and artifacts, including `mklink` (supported in Windows 7 and later).

You can also relocate file storage to a location other than the `var` directory by setting a `server/var.dir` property for the uBuild server. To do this, edit the server's `set_env.cmd` or `set_env` file in the server's `bin` sub-directory. Edit the definition of `JAVA_OPTS` to include the redirection property like this: `-Dserver/var.dir=/alternate/var/path`.

Distributed teams should also take advantage of uBuild location-specific CodeStation proxies to improve performance and lower WAN usage.

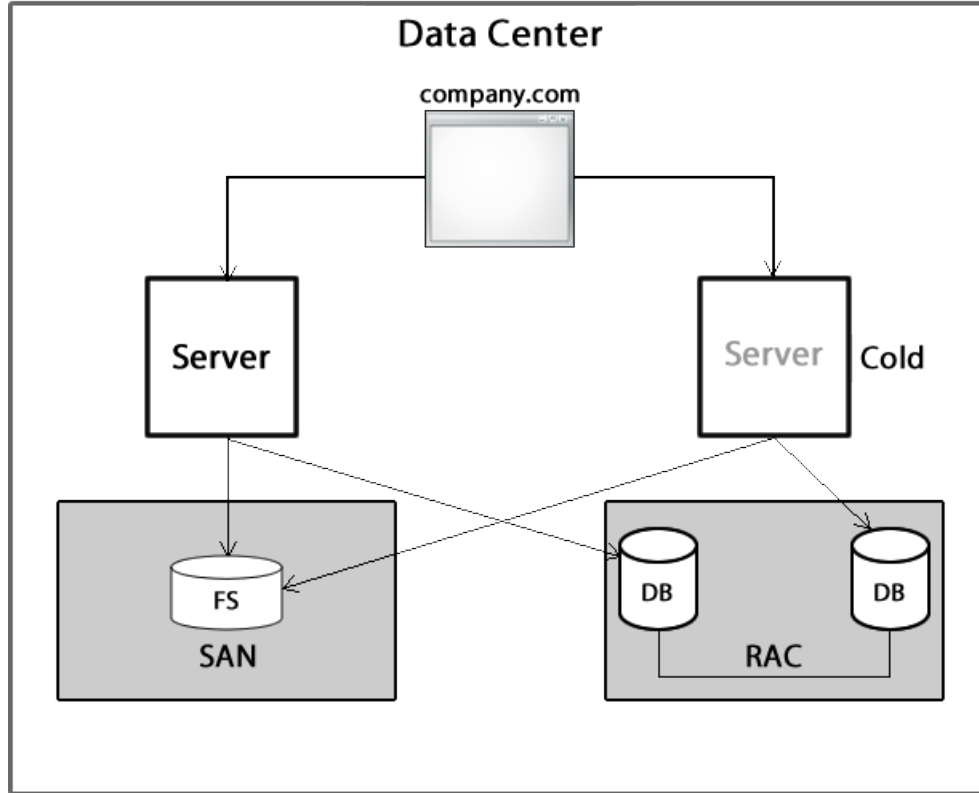
High Availability

High Availability Configuration

uBuild employs the cold standby HA (high availability) strategy for the server. This means that there are two installations of the server on different hardware. Only one installation is running at any given time, it is called the primary server node. The cold standby node is only started when the primary node fails. When the primary server node fails, the cold standby node is brought online and promoted to primary server node. When the primary fails, agents will be disconnected from it and will reconnect to the standby node when it becomes available. At start the standby will resume activity after a pause to allow agents to reconnect. The cold standby scenario has a few requirements:

- **Shared File Store** - In order for the cold standby to have access to logs and CodeStation, they need to be able to access the file store on the same network share location that is available when the primary node fails. See the section called “Relocating CodeStation / File Storage” to configure this.
- **Network Switch** - In order for agents and users to be transitioned from the primary node to the standby node, the resolution of the host name of the server needs to be able to be changed from the primary to the standby at failure.

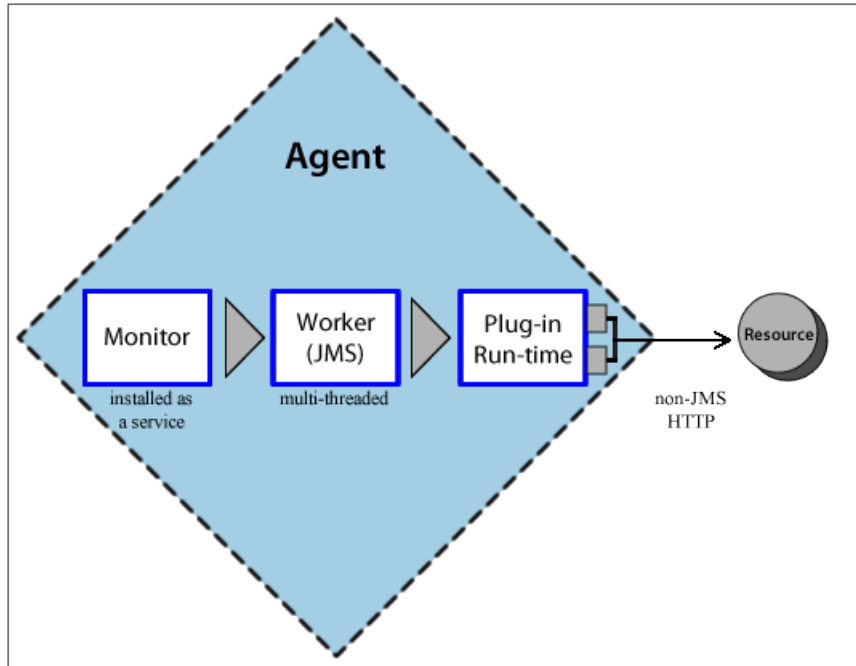
The following diagram illustrates a uBuild high availability configuration.

Figure 2. Cold Standby Configuration

Agents

Agents are the workers of the uBuild architecture. An agent is a lightweight process that runs on a host machine and communicates with the uBuild server. Agents perform the actual build activities which distribute the load of concurrently running builds for scalability. Once an installed agent has been started, the agent opens a connection to the uBuild server. Communication between server and agents uses a JMS-based (Java Message Service) protocol which uses SSL and can optionally enforce mutual authentication. This communication protocol is stateless and resilient to network outages (the benefits of statelessness are discussed below).

While we characterize an agent as a single process, technically an agent consists of a *worker* process and a *monitor* process. The monitor is a service that manages the worker process. It is responsible for starting and stopping the worker, handling restarts, and performing upgrades. If the worker terminates for an unexpected reason, the monitor will restart it. The worker is a multi-threaded process that connects to the server and performs work the server sends it. Agents do not need to be upgraded every time the server is upgraded. The server contains a minimum agent version in it which is the minimum version required to work with the server. The agent is designed in a way that this should not need to be changed. Thus agents are only updated rarely to take advantage of new features or fix bugs. Agents can be upgraded with the install process on the host machine or remote through the server web user interface. Once an agent is installed and connected to the server, it will appear in the web user interface where it can be activated and placed into agent pools to receive work. A newly installed agent will not receive any work unless it is activated in the web user interface.

Figure 3. Agent Processes

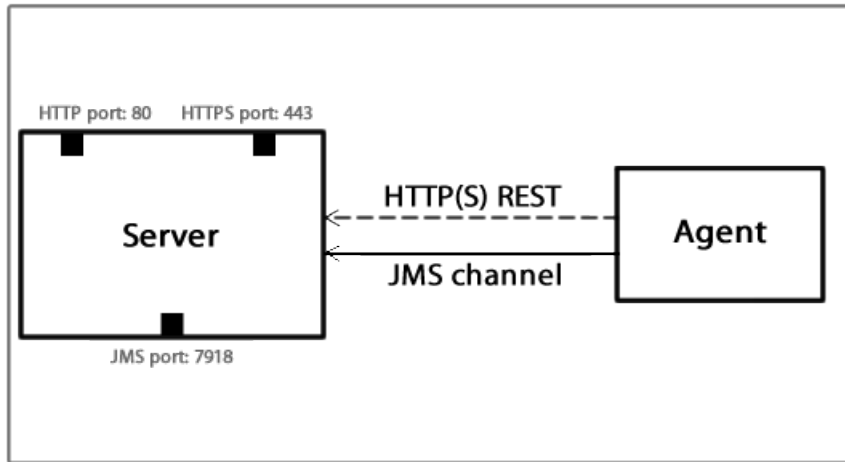
Agents are an important part of uBuild's scalability. By adding more agents, the throughput and capacity of the system increases almost exponentially and so can scale to fit even the largest enterprise.

Server-Agent Communication

Most agent communication is done with JMS, but some agent activities such as uploading logs, test results, or artifact files to CodeStation, use the web interface via HTTP(s) as needed. The JMS channel is uBuild's primary control channel. It is the channel the server uses to send agent commands. By default the server listens on port 7919 for JMS, port 8080 for HTTP and port 8443 for HTTPS. These ports are configured during the server installation. The agent only needs to know the server's host and JMS port. The web interface URL is sent to the agent when it is needed.

The server's agent manager service uses JMS for all server communications and for sending commands, such as "run step," to the worker process. The worker process uses JMS for system communications, and HTTP REST services when performing plug-in steps or retrieving information from the server.

Stateless server-agent communication provides significant benefits to performance, security, availability, and disaster recovery. Because each agent request is self-contained, a transaction consists of independent message which can be synchronized to secondary storage as it occurs. The server or agent can be taken down and brought back up without repercussion (other than lost time). If communications fail mid-transaction, no messages are lost. Once reconnected, the server and agent automatically determine which messages got through and what work was successfully completed. After an connection outage, the system synchronizes the endpoints and recovers affected processes. The results of any work performed by an agent during the outage are communicated to the server.

Figure 4. Stateless Communication

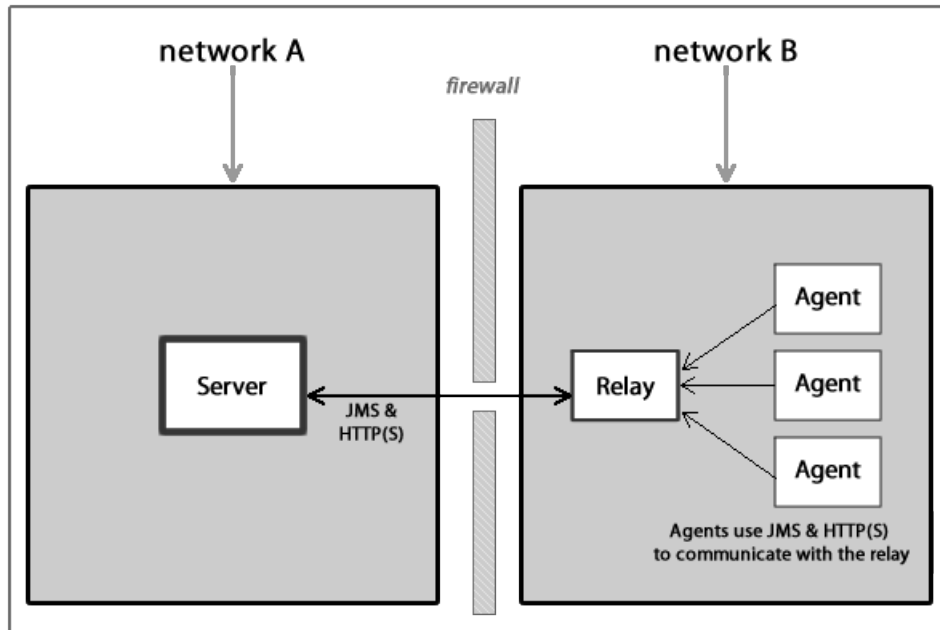
In Figure 4, “Stateless Communication”, the arrow represent the direction in which communications was established, but the flow can be in both directions with JMS.

Crossing Network Boundaries and Firewalls

uBuild supports agents in remote locations that cross network boundaries. As long as there is at least a low bandwidth WAN connection between the server and remote agents, the uBuild server can send work to agents located in other geographic locations. To aid performance and ease maintenance, uBuild uses *agent relays* to communicate with remote agents. An agent relay acts as a single point for remote agents in a network to connect to in order to communicate with the server. So the agent relay acts as a kind of JMS and HTTP proxy. An agent relay then becomes the only machine in the remote network that needs to contact the server. This can greatly simplify firewall rules that need to be created and managed. By default, agent relays connect to the server. If needed to fulfill requirements, the server can be configured to connect to agent relays instead.

The following, a simple artifact move, illustrates the mechanics of remote communications:

1. The server is started and is listening for connections from agents and agent relays via JMS.
2. The agent relay is started, connects to the server via JMS and is listening for connections from agents via JMS.
3. A remote agent starts and establishes a connection to the agent relay via JMS, which, in turn, alerts the uBuild server via JMS that the remote agent is online. The remote agent is now online and can be sent work.
4. The server sends an artifact download command to the relay via JMS, and the relay delivers the message to the remote agent (also via JMS).
5. The agent then attempts to download artifacts. The download command knows that it can not connect directly to the server with HTTP so it uses the agent relay as a HTTP proxy. The artifacts are downloaded using the agent relay as a proxy.
6. Once the remote agent completes the download, it sends a response of completion to the server via JMS which is delivered through the agent relay.

Figure 5. Crossing Network Boundaries

By default, agent relays open the connection to the uBuild server, but the direction can be reversed if your firewall requires it. Remote agents open connections to the agent relay.

In configurations with agent relays, agents in the same network as the server can directly communicate with the server normally.

Mutual Authentication

SSL (Secure Socket Layer) technology enables clients and servers to communicate securely by encrypting all communications. Data is encrypted before being sent and decrypted by the recipient. Communications cannot be deciphered or modified by third-parties. SSL can be used in two modes:

In *unauthenticated mode*, communication is encrypted and decrypted, but endpoints do not have to verify each others credentials. By default uBuild uses this mode for its JMS-based server/agent communication.

In *mutual authentication mode*, communications are encrypted as above, but endpoints are also required to authenticate themselves by providing certificates. A *certificate* is a cryptographically signed document intended to assure others the identity of the endpoint using the certificate. Mutual authentication is enabled on a per-endpoint basis.

When mutual authentication mode is enabled on a endpoint, the endpoint requires that the other endpoint it is communicating with has a trusted certificate. A certificate is trusted if that certificate is signed by another trusted certificate such as a certificate authority or is a trusted certificate itself. A certificate is trusted if it has been imported into the endpoint's keystore as a trusted certificate.

When the server has mutual authentication enabled, the agent certificates used must have been signed by a trusted certificate authority or have been imported into the server's keystore. The same applies for the agent. It is not required that the server and the agent both have mutual authentication enabled together. The setting only applies to the endpoint and how it accepts certificates of endpoints it communicates with.

Mutual authentication mode can be implemented during server/agent installation, or activated afterward. See the section called “SSL Configuration” for information about activating this mode and exchanging certificates between the server and agents.

Getting Started

Installation and Upgrade

A uBuild installation consists of the uBuild server (with a supporting database), and at least one agent. Typically, the server, database, and agents are installed on separate machines, although for a simple evaluation they can all be installed on the same machine. In addition, Java must be installed on all agent and server machines.

Note

The supplied embedded database should only be used for short-term evaluation installations that are expected to be removed when complete. If you are installing uBuild in a production or long-term environment, use one of the databases: Oracle, Microsoft SQL Server, or MySQL.

Installation Steps

1. Review the system requirements. See the section called “System Requirements”.
2. If not using an installer that comes with Java, ensure that Java is installed on the server and agent machines. All machines require Java JRE 5 or greater. Set the `JAVA_HOME` environment variable to point to the directory you intend to use. A JDK can be used and is recommended for the server installation for the debugging tools it includes.
3. Download the uBuild server and agent installation files from the UrbanCode support portal.
4. Download your production or evaluation license from the UrbanCode support portal.
5. The embedded database included in the installers should only be used for short-term evaluation installations that are expected to be removed when complete. If you are installing uBuild in a production, long-term, or performance testing environment, use one of the databases: Oracle, Microsoft SQL Server, or MySQL. If you are going to be installing using a database other than the embedded database, the database should be installed before the server, ideally on a separate machine. See the section called “Database Installation” for more details about supported databases.
6. Download the appropriate Java JDBC driver for your selected database type. This is not needed for the embedded database. JDBC drivers are supplied by database vendors. See the section called “Database Installation” for details about supported JDBC drivers.
7. For all databases other than the embedded database, create an empty database schema for uBuild with a dedicated user account that has permission to create and modify tables and data. See the section called “Database Installation” for details about creating database schemas.
8. Install the server. See the section called “Server Installation”.
9. Install at least one agent. See the section called “Agent Installation”.

Installation Recommendations

The following are installation recommendations and best practices:

- **Install the server as a dedicated user account.** The server should be installed as a dedicated system account whenever possible. While not recommended, uBuild can run as the root user (or local system user on Windows) and running in this manner avoids all permission errors. If you need to run on a privileged port in Unix/Linux, it is recommended to run the server behind an Apache daemon.

- **Install each agent as a dedicated user account.** Ideally, the account should only be used by uBuild. Because uBuild agents are command execution engines, it is advisable to limit what they can do on host machines by creating dedicated users and then granting them appropriate privileges. If you install an agent as the root user (or local system user on Windows), ensure that agent processes cannot adversely effect the host file system.
- **Except for evaluation purposes, do not install an agent on the uBuild server machine.** Because the agent is resource intensive, installing one on the server machine can degrade performance.
- **Install a single agent per host machine.** Multiple agents on the same machine will not increase performance. The amount of work that a agent can perform is limited by the machine's hardware or VM settings, not by the number of agent installations. A single agent installation is not limited to a fixed number of builds.
- **If testing performance, do not use the embedded database.** Use one of the supported databases installed on a separate machine.
- **On Unix/Linux, check the ulimit.** A busy server can use a large number of file handles on Unix/Linux. You may need to increase the ulimit to compensate for this. To check this value, you can run "ulimit -n" as the server's user. It is recommended to set the ulimit to 65535.

System Requirements

uBuild will run on Windows and Unix-based systems. While the minimum requirements provided below are sufficient for an evaluation, you will want server-class machines for production deployments.

Server Minimum Installation Requirements

- **OS:** Windows 2000 Server (SP4) or later, Linux (any distribution). Ask support for other operating systems.
- **Processor:** Single core, 1.5 GHz or better.
- **Memory:** 2 GB, with 512 MB available to uBuild.
- **Java:** 32-bit or 64-bit JRE 5 or later.
- **Disk Space:** 300 MB minimum. (more needed when publishing build artifacts and using the embedded database)

Recommended Server Installation

- **Two server-class machines**

UrbanCode recommends two machines for the server: a primary machine and a standby for fail-over. In addition, the database should be hosted on a separate machine in a fail-over configuration.

- **Separate machine for the database**
- **Processor:** 2 CPUs, 2+ cores for each.
- **Memory:** 8 GB
- **Java:** 32-bit or 64-bit JDK 6 or later.

- **Disk Space:** Individual requirements depend on usage, retention policies, and application types. In general, the larger number of artifacts kept in uBuild's artifact repository (CodeStation), the more storage needed.

Note

CodeStation is installed when the uBuild server is installed.

For production environments, use the following guidelines to determine storage requirements:

- 10-20 GB of database storage should be sufficient for most environments.
- To calculate CodeStation storage requirements:

*average artifact size * number of versions imported per day * average number of days before cleanup*

- Approximately 1MB of database storage per build; varies based build process.

For further assistance in determining storage requirements, contact UrbanCode support.

- **Network:** Gigabit (1000) Ethernet with low-latency to the database.

Agent Minimum Requirements

Designed to be minimally intrusive (typically, an idle agent uses 5Mz CPU), agents require 64-256 MB of memory and 100 MB of disk space. Additional requirements are determined by the processes the agent will run. In production environments, agents should be installed on separate machines.

Download uBuild

The installation package is available from the UrbanCode support portal (<https://support.urbancode.com>). In order to download uBuild, you need a evaluation or production license associated with your account.

1. Open a browser the UrbanCode support portal (<https://support.urbancode.com>). If you do not have an account, please create one. If you do not have a license, you must request an evaluation license.
2. Once you have logged in and have a license, click the **Products** tab and select uBuild from the list of products. Then select the latest version to see its downloads.
3. The uBuild server installer comes in three varieties. There are installers for Windows and Unix/Linux that include a Java 6 JRE. The platform-independent installer will work on any of our supported OS platforms but requires a Java JRE to already be installed.
4. Also download your license, you will need it after installing the server. If you do not see a license, ensure that you are the Supportal account administrator. Licenses are not available to all Supportal users. You may have to contact the account administrator for your group account.

Database Installation

uBuild supports the following databases: Oracle, Microsoft SQL Server, MySQL, and an included embedded database (Apache Derby).

Installing with Oracle

Before installing the uBuild server, create an Oracle database. It is ideal that Oracle is installed on a different machine than the server.

When you install uBuild, you will need the connection information, and a user account with at least these minimum privileges:

RESOURCE, *CONNECT*, *CREATE SESSION*, and *CREATE TABLE*.

uBuild supports the following editions:

- Oracle Database Enterprise
- Oracle Database Standard
- Oracle Database Standard One
- Oracle Database Express

Version 10g or later is supported for each edition.

To install the database

1. Obtain the Oracle JDBC driver. The JDBC jar file is included among the Oracle installation files. The driver is unique to the edition you are using.
2. Copy the JDBC jar file to *uBuild_installer_directory\lib\ext*.
3. Begin server installation, see the section called “Server Installation”. When you are prompted for the database type, enter *oracle*.
4. Provide the JDBC driver class uBuild will use to connect to the database.

The default value is *oracle.jdbc.driver.OracleDriver*.

5. Provide the JDBC connection string. The format depends on the JDBC driver. It is in the format:

jdbc:oracle:thin:@[DB_URL]:[DB_PORT]:[SID]

For example:

jdbc:oracle:thin:@localhost:1521:XE

6. Finish by entering the database user name and password.

Note

The database schema to use should be set as the user's default schema.

Installing with MySQL

Before installing the uBuild server, install MySQL. It is ideal that MySQL is installed on a different machine than the server.

When you install uBuild, you will need the MySQL connection information, and a user account with table creation privileges.

To install the database

1. Create a database:

```
CREATE DATABASE uBuild;  
  
GRANT ALL ON uBuild * TO 'uBuild'@'%'  
  
IDENTIFIED BY 'password' WITH GRANT OPTION;
```

2. Obtain the MySQL JDBC driver. The JDBC jar file is included among the MySQL installation files. The driver is unique to the edition you are using.
3. Copy the JDBC jar file to *uBuild_installer_directory\lib\ext*.
4. Begin server installation, see the section called “Server Installation”. When you are prompted for the database type, enter *mysql*.
5. Provide the JDBC driver class uBuild will use to connect to the database.

The default value is *com.mysql.Driver*.

6. Next, provide the JDBC connection string. Typically, it is formatted:

```
jdbc:mysql://[DB_URL]:[DB_PORT]:[DB_NAME]
```

For example:

```
jdbc:mysql://localhost:3306/uBuild.
```

7. Finish by entering the database user name and password.

Installing with Microsoft SQL Server

Before installing the uBuild server, install a SQL Server database. It is ideal that SQL Server is installed on a different machine than the server.

When you install uBuild, you will need the SQL Server connection information, and a user account with table creation privileges.

Before installing the uBuild server, install an SQL Server database. If you are evaluating uBuild, you can install the database on the same machine where the uBuild server will be installed:

To install the database

1. CREATE DATABASE uBuild;

```
USE uBuild;
```

```
CREATE LOGIN uBuild WITH PASSWORD = 'password';
```

```
CREATE USER uBuild FOR LOGIN uBuild WITH DEFAULT_SCHEMA = uBuild;
```

```
CREATE SCHEMA uBuild AUTHORIZATION uBuild;
```

```
GRANT ALL TO uBuild;
```


2. Obtain the SQL Server JDBC driver from the Microsoft site. The JDBC jar file is not included among the installation files.
3. Copy the JDBC jar file to *uBuild_installer_directory\lib\ext*.
4. Begin server installation, see the section called “Server Installation”. When you are prompted for the database type, enter *sqlserver*.
5. Provide the JDBC driver class uBuild will use to connect to the database.

The default value is *com.microsoft.sqlserver.jdbc.SQLServerDriver*.

6. Next, provide the JDBC connection string. The format depends on the JDBC driver. Typically, it is similar to:

```
jdbc:sqlserver://[DB_URL]:[DB_PORT];databaseName=[DB_NAME]
```

For example:

```
jdbc:sqlserver://localhost:1433;databaseName=uBuild.
```

7. Finish by entering the database user name and password.

Server Installation

The server provides services such as the user interface used to configure project builds, the workflow engine, the security service, and the artifact repository, among others. The properties set during installation are recorded in the *installed.properties* file located in the *server_install/conf/server/* directory.

If the following steps fail, contact UrbanCode support and provide the log from standard output.

Note

If you are installing the server in a production environment, install and configure the database you intend to use before installing the server. See the section called “Database Installation”.

Installing the Server

1. Download and unpack the installation files to the *installer_directory*.
2. From the *installer_directory*, run *install-server.bat* (Windows) or *install-server.sh* (Unix/Linux).

Note

Depending on your Windows version, you might need to run the batch file as the administrator.

The uBuild Installer is displayed and prompts you to provide the following information:

3. **Enter the directory where the uBuild Server will be installed.** For example: *C:\Program Files\ubuild\agent* (Windows) or */opt/ubuild/agent* (Unix). If the directory does not exist, enter *Y* to instruct the Installer to create it for you. If you enter an existing directory, the program will give you the option to upgrade the server. For information about upgrading, see the section called “Upgrading uBuild”.

Note

Any default values suggested by the program (displayed within brackets) can be accepted by simply pressing **Enter**. If two options are given, such as *Y/n*, the capitalized option is the default value.

4. Please enter the home directory of the JRE/JDK used to run the server.

If Java has been previously installed, uBuild will suggest the Java location as the default value. To accept the default value, press *ENTER*, otherwise override the default value and enter the correct path.

5. Enter the IP address on which the Web UI should listen. UrbanCode suggests accepting the default value *all available to this machine*.

6. Do you want the Web UI to always use secure connections using SSL?

Default value is *Y*.

If you use SSL, turn it on for agents too or the agents will not be able to connect to the server. This also applies if using mutual authentication. If you change the port numbers for agent communication, you need to provide the port numbers when installing the agents.

This sets the `install.server.web.always.secure=` property in the `installed.properties` file.

7. Enter the port where uBuild should listen for secure HTTPS requests. The default value is *8443*.

This sets the `install.server.web.ip=` property in the `installed.properties` file.

8. Enter the port on which the uBuild server should redirect unsecured HTTP requests.

The default value is *8080*.

9. Enter the URL for external access to the web UI.

10. Enter the port to use for agent communication.

The default value is *7919*.

11. Do you want the Server and Agent communication to require mutual authentication?

If you select *Y*, a manual key must be exchanged between the server and each agent. The default value is *N*.

This sets the `server.jms.mutualAuth=` property in the `installed.properties` file.

12. Enter the database type uBuild should use.

The default value is the supplied database *Derby*. The other supported databases are: *mysql*, *oracle*, and *sqlserver*.

If you enter a value other than *derby*, the uBuild Installer will prompt you for connection information, which was defined when you installed the database. See the section called “Database Installation”.

13. Enter the database user name. The default value is *ubuild*. Enter the user name you created during database installation.

14. **Enter the database password.** The default value is *password*.

15. **Do you want to install the Server as Windows service?** (Windows only). The default value is *N*.

When installed as a service, uBuild only captures the value for the PATH variable. Values captured during installation will always be used, even if you make changes later. For recent Windows versions, you will need to execute the command as Administrator.

Note

If you install the server as a service, the user account must have the following privileges:

- SE_INCREASE_QUOTA_NAME "Adjust memory quotas for a process"
- SE_ASSIGNPRIMARYTOKEN_NAME "Replace a process level token"
- SE_INTERACTIVE_LOGON_NAME "Logon locally"

The LOCAL SYSTEM account is on every Windows machine and automatically has these privileges. You might want to use it as it requires minimal configuration.

Agent Installation

For production environments, UrbanCode recommends creating a user account dedicated to running the agent on the machine where the agent is installed.

Each agent needs the appropriate rights to communicate with the uBuild server.

At a minimum, each agent should have permission to:

- **Access the user's home directory.** By default, a CodeStation cache is located in a .codestation directory the home directory of the user running the agent. The cache can be moved or disabled in uBuild under the agent's configuration.
- **Open a TCP connection.** The agent uses a TCP connection to communicate with the server's JMS port. Default port is 7919.
- **Open a HTTP(S) connection.** The agent must be able to connect to the uBuild web user interface in order to download and upload information such as logs and artifacts.
- **Access the file system.** Agents should have full access to their own installation directories. Agents may also need access to other directories depending on what they will be building and where it occurs.

Installing an Agent

1. After downloading and extracting the installation package, open the *installer_directory*.
2. From the *installer_directory* run *install-agent.bat* (Windows) or *install-agent.sh* (Unix/Linux).

Note

Depending on your Windows version, you might need to run the batch file as the administrator.

The uBuild Agent Installer is displayed and prompts you to provide the following informatio

3. **Enter the directory where agent should be installed..** For example: C:\Program Files\ubuild\agent (Windows) or /opt/ubuild/agent (Unix). If the directory does not exist, enter *Y* to instruct the Installer to create it for you. If you enter an existing directory, the program will give you the option to upgrade the agent. For information about upgrading, see the section called “Upgrading uBuild”.

Note

Any default values suggested by the program (displayed within brackets) can be accepted by simply pressing **Enter**. If two options are given, such as *Y/n*, the capitalized option is the default value.

4. **Please enter the home directory of the JRE/JDK used to run the agent.** If Java has been previously installed, uBuild will suggest the Java location as the default value. To accept the default value, press *ENTER*, otherwise override the default value and enter the correct path.
5. **Will the agent connect to a agent relay instead of directly to the server?** The default value is *N*.
6. **Enter the host name or address of the server the agent will connect to.** The default value is *localhost*.
7. **Enter the agent communication port for the server.** The default value is *7919*.
8. **Does the server agent communication use mutual authentication with SSL?.** Default value is *Y*.
- If you use SSL, turn it on for server too or the agent will not be able to connect to the server. This also applies if using mutual authentication. If you change the port numbers for agent communication, you need to provide them when installing the agents.
9. **Enter the name for this Agent.** Enter a unique name; the name will be used by uBuild to identify this agent. Names are limited to 256 characters and cannot be changed once connected.
10. **Do you want to install the Agent as Windows service?** (Windows only). The default value is *N*. When installed as a service, uBuild only captures the value for the PATH variable. Values captured during installation will always be used, even if you make changes later. For recent Windows versions, you will need to execute the command as Administrator.

Note

If you will be installing the agent as a Windows service, the user account must have the following privileges:

- SE_INCREASE_QUOTA_NAME "Adjust memory quotas for a process"
- SE_ASSIGNPRIMARYTOKEN_NAME "Replace a process level token"
- SE_INTERACTIVE_LOGON_NAME "Logon locally"

The LOCAL SYSTEM account is on every Windows machine and automatically has these privileges. You might want to use it as it requires minimal configuration.

Upgrading uBuild

You can upgrade the uBuild server and agents using the same exact process you used to install them. To upgrade a server or agent, download the appropriate installation package from the UrbanCode support portal, and extract it.

1. Before upgrading, make sure the target server or agent you are upgrading has been shut down.
2. Run the installation script for the item you want to upgrade. To upgrade the server, for example, run the *install-server* script; to upgrade an agent, run the *install-agent* script.
3. When prompted for the location of the installation directory, enter the path to an existing installation. When you specify an existing installation, uBuild will ask if you want to upgrade the installation (instead of installing a new version). If you answer Yes, the script will lead you through the required steps. The upgrade steps are a subset of the installation steps. If you need information about the steps, see the section related to the item you are upgrading--server, agent, agent relay.

SSL Configuration

SSL (Secure Socket Layer) technology enables clients and servers to communicate securely by encrypting all communications. Data are encrypted before being sent and decrypted by the recipient--communications cannot be deciphered or modified by third-parties.

uBuild enables the server to communicate with its agents using SSL in two modes: unauthenticated and mutual authentication. In unauthenticated mode, communication is encrypted but users do not have to authenticate or verify their credentials. uBuild automatically uses this mode for JMS-based server/agent communication (you cannot turn this off). SSL unauthenticated mode can also be used for HTTP communication. You can implement this mode for HTTP communication during server/agent/agent relay installation, or activate it afterward, as explained below.

Important

uBuild automatically uses SSL in unauthenticated mode for JMS-based communications between the server and agents (JMS is uBuild's primary communication method). Because agent relays do not automatically activate SSL security, you must turn it on during relay installation or before attempting to connect to the relay. Without SSL security active, agent relays cannot communicate with the server or remote agents.

In mutual authentication mode, the server, local agents, and agent relays each provide a digital certificate to one another. A digital certificate is a cryptographically signed document intended to assure others about the identity of the certificate's owner. uBuild certificates are self-signed. When mutual authentication mode is active, uBuild uses it for JMS-based server, local agents, and agent relay communication.

To activate this mode, the uBuild server provides a digital certificate to each local agent and agent relay, and each local agent and agent relay provides one to the server. Agent relays, in addition to swapping certificates with the server, must swap certificates with the remote agents that will use the relay. Remote agents do not have to swap certificates with the server, just with the agent relay it will use to communicate with the server. This mode can be implemented during installation or activated afterward, as explained below

Note

When using mutual authentication mode, you must turn it on for the server, agents, and agent relays, otherwise they will not be able to connect to one another--if one party uses mutual authentication mode, they all must use it.

Configuring SSL Unauthenticated Mode for HTTP Communications

To activate unauthenticated mode for HTTP:

1. Open the `installed.properties` file which is located in the `server_install/conf/server` directory. The `installed.properties` file contains the properties that were set during installation.
2. Ensure that the `install.server.web.always.secure` property is set to `Y`.
3. Ensure that the `install.server.web.ip` property is set to the port the server should use for HTTPS requests.
4. Save the file and restart the server.

Note

To complete unauthenticated mode for HTTP, contact UrbanCode Support.

Configuring Mutual Authentication

To use mutual authentication, the server and agents must exchange keys. You export the server key (as a certificate) and import it into the agent keystore, then reverse the process by exporting the agent key and importing it into the server keystore. When using an agent relay, the relay must swap certificates with the server and with the remote agents that will use the relay.

Before exchanging keys, ensure that the following properties are set:

1. The `server.jms.mutualAuth` property in the server's `installed.properties` file (located in the `server_install/conf/server` directory) is set to `true`.
2. For each agent, the `locked/agent.mutual_auth` property in the agent's `installed.properties` file (located in the `agent_install\conf\agent` directory) is set to `true`.
3. For each agent relay, the `agentrelay.jms_proxy.secure` property in the relay's `agentrelay.properties` file (located in the `relay_install\conf` directory) is set to `true`.
4. For each agent relay, the `agentrelay.jms_proxy.mutualAuth` property in the relay's `agentrelay.properties` file is set to `true`.

To exchange keys:

1. Open a shell and navigate to the server installation `conf` directory.
2. Run:

```
keytool -export -keystore server.keystore -storepass changeit  
-alias server -file server.crt
```

3. Copy the exported file (certificate) to the local agent/agent relay installation `conf` directory.
4. Import the file by running from within the agent's `conf` directory (or agent relay's `jms-relay` directory):

```
keytool -import -keystore agent.keystore -storepass changeit  
-alias server -file server.crt -keypass changeit -noprompt
```

You should see the `Certificate was added to keystore` message.

5. For each local agent or agent relay, export the key by running the following (change the name of the file argument to match the agent name):

```
keytool -export -keystore agent.keystore -storepass changeit  
-alias ud_agent -file [agent_name].crt
```

You should see the Certificate stored in file (agent_name.crt) message.

6. Copy the exported file to the server's conf directory.

7. From within the server's conf directory, import each certificate by running the following command (change the name of the file argument and alias to match the certificate's name):

```
keytool -import -keystore agent.keystore -storepass changeit  
-alias [agent_name] -file [agent_name].crt -keypass changeit -noprompt
```

You should see the Certificate was added to keystore message.

8. Restart the server and agents/agent relays.

To connect an agent relay with the remote agents that will use it, swap certificates as explained above: each remote agent must import the certificate for the relay it will use, and the relay must import the certificate from each remote agent that will use it. Agents using relays do not have to swap certificates with the server.

To list the certificates loaded into a keystore, run the following from within the keystore directory:

```
keytool -list -keystore agent.keystore -storepass changeit
```

Running uBuild

Both Unix and Windows-based installations require the uBuild server and at least one agent. Once both the server and agent are installed, you can start them and begin using them.

Running the Server

1. Navigate to the *server_installation\bin* directory
2. Run the *start_server.cmd* batch file (Windows), or execute the command *./server start* (Unix/Linux).

Note

When using the start command in Unix/Linux, the process will run in the background

Running an Agent

After the server has successfully started:

1. Navigate to the *agent_installation\bin* directory
2. Run the *start_ubuildagent.cmd* batch file (Windows), or execute the command *./ubuildagent start* (Unix/Linux).

Note

When using the start command in Unix/Linux, the process will run in the background

3. Once the agent has started, navigate to the uBuild web user interface and display the **Agents** tab and click on the **Available** agents link. The agent should appear with a status of **Online**. Click on the agent to edit it and mark it as configured and place it in a agent pool.

Accessing uBuild

1. Open a web browser and navigate to the host name and port you configured during installation.
2. Log onto the server by using the default credentials.

USERNAME: *admin*

PASSWORD: *admin*

You can change the password later by using the **profile** link in the upper-right section of the uBuild web application.

3. Activate the license. A license is required in order to perform builds. Without a license, uBuild will be unable to run builds.

Stopping the Server

1. Navigate to the *server_installation\bin* directory
2. Run the *stop_server.cmd* batch file (Windows), or execute the command *./server stop* (Unix/Linux).

Stopping an Agent

1. Navigate to the *agent_installation\bin* directory
2. Run the *stop_ubuildagent.cmd* batch file (Windows), or execute the command *./ubuildagent stop* (Unix/Linux).

Quick Start Tutorial—Creating a Workflow

This section gets you started by providing immediate hands-on experience using key product features. The tutorial provides step-by-step instructions for creating a project that builds UrbanCode's open-source product, Terraform. The source code will be downloaded from GitHub (a popular open source code repository) and built with Maven. See UrbanCode's product page for information regarding Terraform.

Before Starting

The tutorial assumes you have installed the uBuild server and at least one agent. In addition, to prepare your environment, perform the following:

Table 3.

Activity	Description
Install open source tools	Install and configure Git, Maven, and Groovy on your machine, if you haven't already. Important Ensure that Maven is part of the PATH environment variable for the machine where the agent is located.
Install Plug-ins	The tutorial requires the following plug-ins: Git, Maven, Groovy, and JUnit. See the section called “Installing Plug-ins” for information about installing plug-ins.
Configure a Maven repository	As shipped, uBuild provides integration with Maven. To perform the tutorial's steps, display the Maven page (<i>System > Maven</i>) and configure a Maven repository with the following values: Name= <i>maven.org</i> , and URL= <i>http://repol.maven.org/maven2</i>
Configure an artifact set	Display the Artifact Sets page (<i>System > Artifact Sets</i>) and create a new artifact set named <i>maven</i> .
Create an agent pool	The tutorial requires an <i>agent pool</i> . An agent pool is a collection of agents that can share work. Before starting, create an agent pool (<i>Agents > Agent Pools > Create Fixed Agent Pool</i>). You can name it anything you like. Next, click the name of the newly created pool and use the displayed dialog to add at least one agent to the pool.
Configure Java	The agent that will run the tutorial project must be configured to use a Java JDK instance, and not a JRE. If the agent was configured to use a JRE (the <i>java.home</i> property configured during agent installation), you must reconfigure the

Activity	Description
	agent to point to a JDK, or add the <i>JDK_HOME</i> property to its <i>installed.properties</i> file (agent_installation/conf/agent). In addition, the tutorial's Maven Build step must reference that property with its <i>JAVA_HOME</i> setting (<i>\${JDK_HOME}</i>).

In brief outline, during the tutorial, you will:

1. Create a Git repository from which your project will download the source for Terraform.
2. Create a workflow and associate an agent pool to it.
3. Create a job that checks-out the Terraform source from GitHub and uses Maven to build Terraform, and run tests and generate JUnit test reports.
4. Create a template that defines several properties. Values for these properties will be supplied when you create the project.
5. Create a project based on the template. During configuration, you will apply values for properties defined earlier.
6. Finally, you will run a successful build.

Note

If you do not want to create a team for the tutorial, you can use the default System Team.

Create a Git Repository

Repository integrations enable uBuild to check out code, access the changelog, and other activities. Repositories are configured with the system settings.

To Create a Git Repository

1. Display the Create Repository pane (*System > Repositories > Create New button*).
2. Select *Git* from the Source Plugin drop-down list box.

Note

Available sources are derived from the installed source-type plug-ins; ensure that the Git plug-in is installed.

3. Complete the New Repository pane's fields, then save your work. Bold fields are required.

Table 4.

Field	Description
Name	Enter a name for the repository. Typically, the name reflects the purpose of the repository. To follow the tutorial, enter <i>GitHub</i> .
Description	The optional description can be used to convey additional information about the repository. I entered 'Anonymous access to github.com.'
Repository Base URL	Enter the GitHub URL: <i>https://github.com/</i> , then save your work.

The GitHub repository can now be used in the workflows you create.

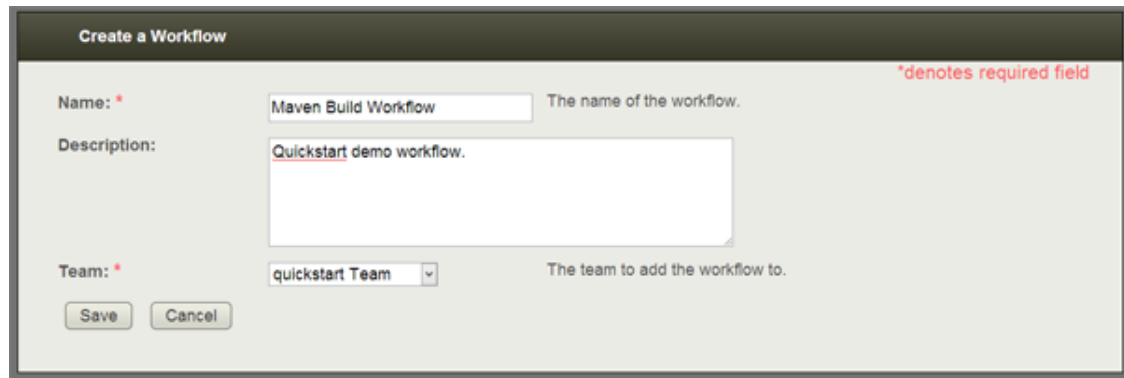
Create a Workflow

Workflows define the jobs and steps that go into a build. Steps are collected into jobs and jobs are composed into workflows. Templates reference workflows to provide them to projects. When placing a job in a workflow, you choose options such as the agent pool where its steps will run, the working directory, as well as other settings.

To Create a Workflow

1. Display the Create a Workflow dialog (*Configuration > Workflows tab > Create button*).

Figure 6. Create a Workflow Dialog



2. Complete the dialog's fields, then save your work.

Table 5.

Field	Description
Name	Enter a name for the workflow. Typically, the name reflects the type of work performed by the workflow, such as <i>Trunk Build</i> . To follow the tutorial, enter <i>Maven Build Workflow</i> .
Description	The optional description can be used to convey additional information about the workflow.
Team	Select the team that will manage the workflow.

Saved workflows are listed on the Workflows page. To modify a workflow, click its name or click the associated Edit action icon, which is highlighted in the following illustration.

Figure 7. Edit Workflow Action



Create a Job

A job is a series of actions—steps—that perform a build. The steps are executed one at a time in the order you specify. A job's steps are created with plug-ins. A plug-in's customizable steps can be thought of as distinct pieces of automation. By combining steps from various plug-ins, you can create fully-automated build processes. The number of steps provided by a particular plug-in can vary, and each step has a variable number of properties. Property values can be supplied when defining a job or entered at process run-time.

Note

Although you can use the functions available with the CLI to perform most of the activities described in this tutorial, the easiest way to learn uBuild is to let the UI guide you through the process.

To Create a Job

1. Display the Create a Job dialog (*Configuration > Workflows tab > Maven Build Workflow > Create button*). If you named your workflow something other than *Maven Build Workflow*, you would select that, of course.
2. Enter a name for the job and save your work. To follow the tutorial, enter *Maven Build Job*.

Saved jobs are listed on the workflow's page. To modify a job, click its name or click the associated Edit action icon.

The following sections describe the steps in the Maven Build Job.

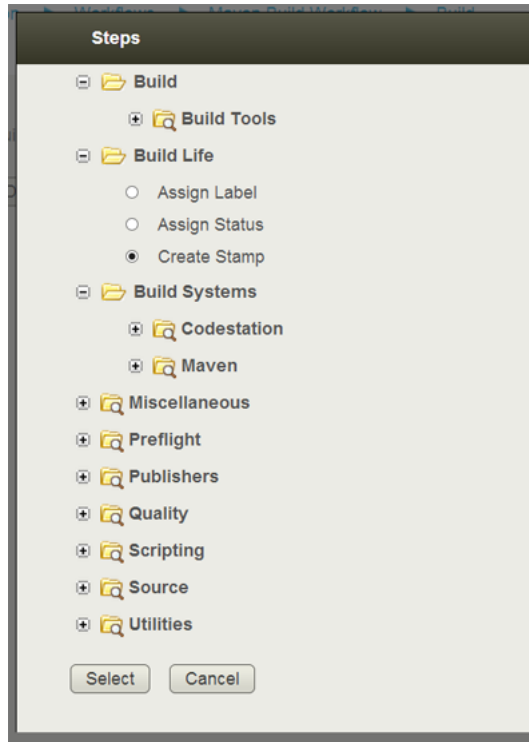
Create a Stamp

A stamp is a custom identifier applied to a build. Stamps are used in addition to the automatically-generated build life identifier, and enable you to apply your own identifiers. This step is part of most jobs.

To Configure a Create a Stamp Step

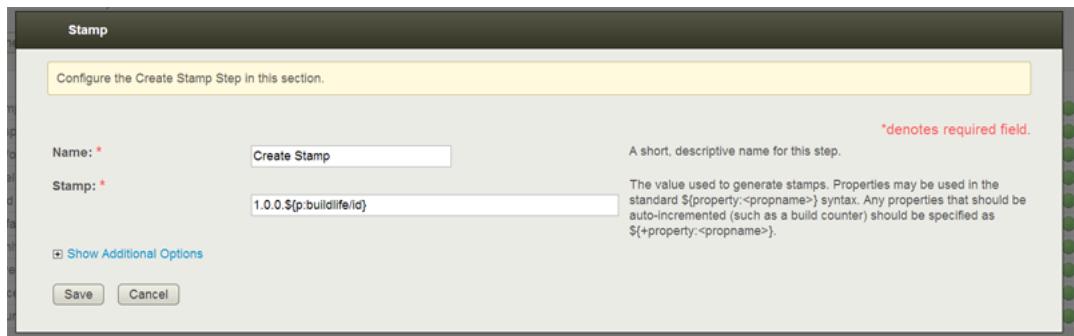
1. Display the Steps dialog (*Configuration > Workflows tab > Maven Build Workflow > Maven Build Job > Create button*). All installed plug-ins are listed, and each step is selectable.

Figure 8. Steps Dialog



2. Select the Create Stamp step, which is contained in the Build Life plug-in. The Stamp dialog is displayed. The displayed dialog is always tailored for the selected step, as we will see in the following sections.

Figure 9. Stamp Dialog



3. Enter a name for the stamp and a script that will create a unique stamp. The simplest way to make a unique stamp is to associate it with the build life number. Use the following to generate a stamp:

`1.0.0.${p:buildlife/id}`

The short script looks-up the current build ID and applies it to the stamp.

Additional Options for Steps

Each step dialog has additional options available that can be displayed by clicking the Show Additional Options link. The options are the same for all steps and are set to default values that are generally applicable. The default value for the Pre-Condition Script field is *All Prior Success*, which means a step will only run if the preceding steps have been successful. Except for two steps in the job, each step will use the default value for this field and all other additional fields.

Figure 10. Additional Options

Stamp

Configure the Create Stamp Step in this section.

Name: * Create Stamp A short, descriptive name for this step. *denotes required field.

Stamp: * 1.0.0.\$(p:buildid) The value used to generate stamps. Properties may be used in the standard \$(property:<proname>) syntax. Any properties that should be auto-incremented (such as a build counter) should be specified as \$(<property>:<proname>).

[Hide Additional Options](#)

Is Active: * Yes Select 'No' if you want to temporarily deactivate this step without having to delete it.
○ No

Run in Preflight: * Yes Select 'No' if you do not want this step to execute in a preflight workflow.
○ No

Run in Preflight Only: ○ Yes Select 'No' if you do not want to run this step only in preflight workflows.
* No

Pre-Condition Script: All Prior Success The pre-condition script which must pass before the step will execute.

Ignore Failures: ○ Yes Select 'Yes' if this step should not affect the determination for step continuation or the status determination of the job. This is used if the success or failure of this step is irrelevant.
* No

Post Process Script: -- Use Plugin Default -- Select a script for determining when commands of this step should count as fail or succeed.
[New Script](#) / [Edit Script](#)

Timeout: 0 Enter the time in minutes after the start of the step when uBuild will consider the step as timed out and abort it.
The step will be marked as failed and the job execution will continue based on that outcome.
Default value is 0 which means that the step never times out.

[Save](#) [Cancel](#)

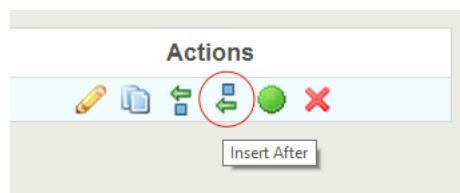
Clean Workspace

To ensure that files remaining from a previous build are not incorporated into the new one, clean the workspace before populating it. This step is part of most jobs.

To Configure a Clean Workspace Step

1. On the Maven Build Job page, use the Insert After action for the Create Stamp step. The Insert After Action icon is shown in the following illustration. (If you named your job something other than *Maven Build Job*, you would select that, of course.)

Figure 11. Step Actions



Note

Once a job has its first step, additional steps are inserted before or after existing steps.

2. Select the Clean Workspace step, which is contained in the Source plug-in. The Cleanup Source Plug-in dialog is displayed.
3. Enter a name for the step. The Name field is the only field displayed

Populate Workspace

This step places the checked-out code (defined in the project's source configuration) in the workspace, see the section called “Create a Project”.

To Configure the Populate Workspace Step

1. On the Maven Build Job page, use the Insert After action for the Clean Workspace step.
2. Select the Populate step, which is contained in the Source plug-in. The Populate Workspace Source Plugin Meta dialog is displayed.
3. Enter a name and save your work.

Get Changelog

This step retrieves source changes since the last build. The prior build life is located using status or stamp parameters.

To Configure the Get Changelog Step

1. On the Maven Build Job page, use the Insert After action for the Populate Workspace step.
2. Select the Get Source Changes step, which is contained in the Source plug-in. The Get Changelog Source Plugin Meta dialog is displayed.
3. Complete the dialog's basic fields (no additional options are required), then save your work. Required fields are in bold type.

Table 6.

Field	Description
Name	Enter a name; to follow the tutorial, enter <i>Get Changelog</i> .
Start Status	The value selected determines which source items are retrieved; only items with the selected value are retrieved. Select <i>Success</i> .
Start Stamp Pattern	You can restrict which items to retrieved by defining a stamp pattern—only items with stamps matching the pattern are retrieved. For the tutorial, leave the field blank.
Save Changes in Database	Ensure this check box is checked, which it is by default.

Maven Build

Note

Reminder: the tutorial assumes you have installed Maven and configured a Maven repository.

To Configure the Maven Build Step

1. On the Maven Build Job page, use the insert a step after the Get Changelog step. Select the Maven Build step, which is contained in the Maven plug-in (*Build Systems > Maven*).
2. Complete the dialog's basic fields using the values in the following table, then save your work.

Note

For the remainder of the walk through, only dialog fields required by the tutorial are described.

Table 7.

Field	Description
Name	To follow the tutorial, enter <i>Maven Build</i> .
Working Directory Offset	This specifies the working directory used by uBuild. By leaving it blank, the current directory will be used.
Maven POM file	Enter the name of your POM file. The POM file (Project Object Model) is Maven's tool for controlling its builds. To follow the tutorial, accept the default <i>pom.xml</i> .
Goals	Goals are similar to Ant tasks and define the actions that Maven will perform. The value entered here will point to a parameter defined in the template we will eventually use. For the tutorial, enter: <code>\${p:maven-goals}</code> .
Maven Flags	Flags are command options passed to Maven at run-time. The value entered here will point to a parameter defined in the template we will eventually use. For the tutorial, enter: <code>\${p:maven-flags}</code> .
Properties	To follow the tutorial, enter: <code>altDeploymentRepository=ubuild-repo::default:: \${env/WEB_URL}/rest/maven2dist/\${p:buildlife/id}</code> .
Maven Version	Select <i>Maven 2</i> or <i>3</i> (the default).
JAVA_HOME	Enter the directory where Java is installed. To follow the tutorial, accept the default <code>\${env/JAVA_HOME}</code> which automatically reads the system variable (if set).

Upload Artifacts

This step enables you to use CodeStation as a Maven repository. It will replicate the Maven repository in Codestation and store the artifacts in it.

To Configure the Upload Artifacts Step

1. On the Maven Build Job page, use the insert a step after the Maven Build step. Select the Upload Artifacts step, which is contained in the CodeStation plug-in (*Build Systems > CodeStation*).
2. Complete the dialog's basic fields (no additional options are required), then save your work.

Table 8.

Field	Description
Name	To follow the tutorial, enter <i>Upload Artifacts</i> .
Artifact Set Name	An artifact Set is a label for a collection of build artifacts. You can create as many sets as you want and later associate build artifacts with a particular set. For the tutorial, name the set <i>maven</i> .
Symlinks	Accept the default value-- <i>As Link</i> --which means symlinks are sent as symlinks. Note, only available on Unix systems.
Directories	Accept the default value-- <i>Include All</i> --which means all directories will be uploaded, even empty ones.
Permissions	Accept the default value-- <i>None</i> --which means no permissions are sent.

Publish Test Results With JUnit

JUnit is a testing framework for Java. This step runs tests and reports the results. It is configured to always run.

To Configure the JUnit Step

1. On the Maven Build Job page, use the Insert Step After action for the Upload Artifacts step. Select the JUnit step, which is contained in the JUnit plug-in (*Quality > Testing > JUnit*).
2. Complete the dialog's basic fields using the values in the following table (note that the default value for one of the additional options must be changed, which is described in Step. 3).

Table 9.

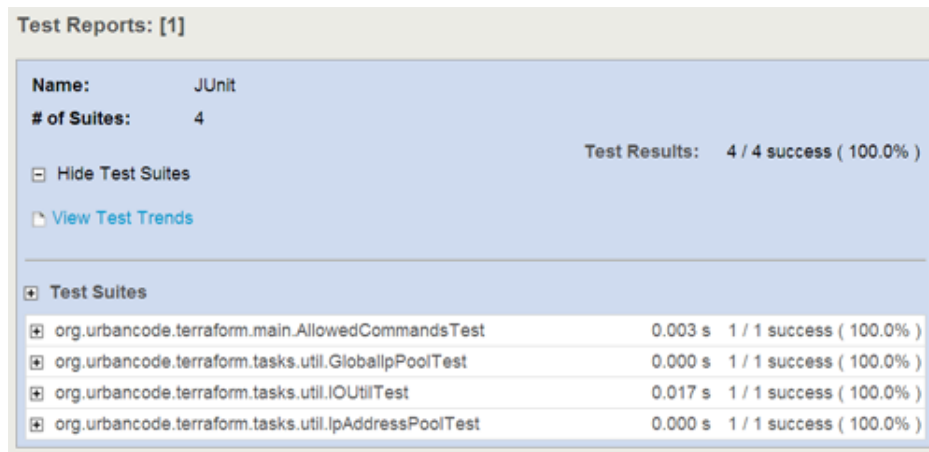
Field	Description
Name	To follow the tutorial, enter <i>Publish JUnit Results</i> .
Report Name	The name given to the report created by JUnit. To follow the tutorial, enter <i>JUnit</i> .
Source Directory	This designates the directory where the reports will be created. To follow the tutorial, enter <i>target/surefire-reports</i> .
Include Patterns	Accept the default value-- <i>TEST*.xml</i> --which means tests matching the pattern will be run.

3. Display the Pre-Condition Script field by clicking the Show Additional Options link. Select *Always* from the Pre-Condition Script drop-down list box. The step will always run which is different from any other step. We want it to run even if the Maven Build step fails, in which case we will still publish and set the failing tests.

To jump ahead a bit, when the project runs, JUnit reports will be displayed on the Tests tab (*Projects > selected project > selected build process > selected build life*), as shown in the following illustration, and also be written to the directory specified when you configured the step. In addition, test files that match the specified pattern will be written to the same directory, for example:

```
TEST-org.urbancode.terraform.main.AllowedCommandsTest.xml
```

Figure 12. JUnit Test Reports



Assign Maven Version

In this step, we use a Groovy script to assign a Maven version number. Scripts can use a `java.util.Properties` variable for output properties to upload to the server called `outProps` and another containing the input properties—`inProps`.

To Configure the Assign Maven Version Step

1. On the Maven Build Job page, insert a step after the Upload Artifacts step. Select the Run Groovy Script step, which is contained in the Groovy plug-in (*Scripting > Groovy*).
2. Enter a name, and the script:

```
println "Setting Maven version to ${p:buildlife/latestStamp}";  
outProps.setProperty("buildlife/maven.version", "${p:buildlife/latestStamp}");
```

Assign Success

This step assigns a status of 'Success' to the entire job. The step will be performed if all preceding steps completed successfully, otherwise the next step will run instead.

To Configure the Assign Success Step

1. On the Maven Build Job page, insert a step after the Assign Maven Version step. Select the Assign Status step, which is contained in the Build Life plug-in.
2. Name the step *Assign Success*, and select *Success* from the Status drop-down list box.

Assign Failure

This step assigns a status of 'Failure' to the entire job. The step will be performed if any of the preceding steps failed. Configure this step in the same way you configured the previous step.

To Configure the Assign Failure Step

1. On the Maven Build Job page, insert a step after the Assign Success step. Select the Assign Status step, which is contained in the Build Life plug-in.

2. Name the step *Assign Failure*, and select *Failure* from the Status drop-down list box.
3. Display the Pre-Condition Script field by clicking the Show Additional Options link. Select *Any Prior Failure* from the Pre-Condition Script drop-down list box. The step will run if any preceding step fails.

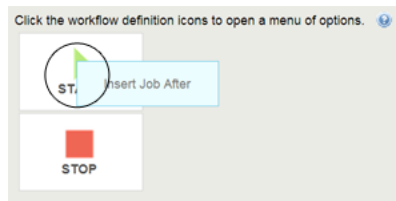
Assign the Job to the Workflow

In order for a job to be used, it must be assigned to the workflow. A workflow can have several jobs and their order in the workflow is determined similarly to the way steps are inserted into jobs.

To Insert a Job Into the Workflow

1. On the Maven Build Workflow page, click on the Start Action icon, and select the *Start Job After* action from the context menu. The Insert Start Job dialog is displayed.

Figure 13. Start Action Icon



2. Complete the dialog's fields using the values in the following table.

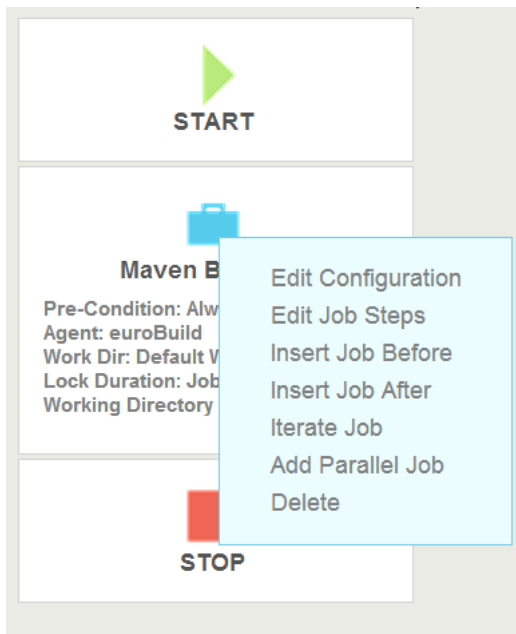
Table 10.

Field	Description
Job	From the Job drop-down list box, select the job you created earlier: <i>Maven Build Job</i> .
Pre-Condition	From the Pre-Condition drop-down list box, select <i>Always</i> , which means the job will always run.
Agent Selection	From the Agent Select drop-down list box, select one of the agents you installed earlier. This agent will perform the actual work of the job's steps.
Working Directory	From the Working Directory drop-down list box, select <i>Default Workflow Directory</i> .

3. Finish by using the Insert Job button.

Once finished, the job is listed in the Jobs list. You can edit it or perform other actions, such as inserting another job after it, by using the Job icon, as shown in the following illustration.

Figure 14. Maven Build Job Icon



Create a Template

A template defines a generic configuration by defining properties that are given values by the projects created from them. For instance, a template might always invoke a particular type of build script but the name or location of the script can vary by project.

To Create a Template

1. Display the Create a Project Template dialog (*Configuration > Templates tab > Create button*).
2. To follow the tutorial, name the template *Maven Build Template*.
3. Select a team to manage the template.
4. Define the properties which were referenced by the Maven Build step you created earlier. Click the Properties tab and use the values in this table to define the properties:

5. **Table 11.**

Name	Label	Default Value
maven-flags	Maven Flags	default
maven-goals	Maven Goals	package

Both properties are of display type *Text*, and both are required.

The following sections describe how to complete the template by defining a source and a build process.

Source Templates

A source template associates a source with the template. Any project built with the template will have the source available to it.

To Create a Source Template

1. On the Maven Build Template page, click the Create Source Templates button. The New Source Template page is displayed.
2. Select *Git* from the Source Type drop-down list box. All installed source-type plug-ins are listed. The fields on this page are tailored to the source type you select; once a source is selected, the type-specific fields are displayed.
3. Use the values in this table to complete the source template, and save your work when you are done.

Table 12.

Field	Description
Name	To follow the tutorial, enter <i>GitHub</i> .
Working Directory Script	From the drop-down list box, select <i>Default Workflow Directory</i> .
Remote URL	This is used for the Git project name. Enter: <i>\${p:project}</i> . The property's value will be supplied when you configure the project.
Branch	This identifies the Git project. Enter: <i>\${p:branch}</i> . The property's value will be supplied when you configure the project.

The Remote URL and Branch fields will be defined by properties, which you define next. Anyone using this template will supply values for the properties when they configure a project, or accept the default values you define, if any.

To Configure Source Template Properties

1. Display the Properties pane for the GitHub source template.
2. Use the Create button to create the first property. Use the values in this table to define the two properties:.

Table 13.

Name	Label	Default Value
<i>branch</i>	<i>Branch</i>	<i>master</i>
<i>project</i>	<i>Project</i>	Leave blank.

Both properties are of display type *Text*, and both are required.

Configuring Build Process Templates

A build process template associates a workflow with the template. Any project built with the template will have the workflow available to it.

To Create a Build Process Template

1. On the Maven Build Template page, click the Create Build Process Templates button to display the New Build Process Template dialog.
2. Provide a name for the process. To follow along with the tutorial, enter *Maven Build Process*.

3. From the Workflow drop-down list select the workflow you created earlier—*Maven Build Workflow*.

To Configure Build Process Template Properties

1. Display the Properties pane for the Maven Build Process template.
2. Use the Create button to create the first property. Use the values in this table to define the two properties:

Table 14.

Name	Label	Default Value
<i>maven.artifactId</i>	<i>Maven Artifact Id</i>	Leave blank.
<i>maven.groupId</i>	<i>Maven Group Id</i>	Leave blank.

Both properties are of display type *Text*, and both are required.

To Configure Build Process Template Artifacts

Generated artifacts must be assigned to an artifact set. An artifact set is a label for a collection of build artifacts. uBuild enables you to create as many sets as you want, and then associate the build artifacts to a particular set.

1. Display the Artifacts pane for the Maven Build Process template.
2. Use the New Artifact Config button to display the Artifact Configurations pane, and use the values in this table to define property:

Note

Reminder: the tutorial assumes you followed its suggestion by configuring a 'maven' artifact set.

Table 15.

Artifact Set	Base Directory	Include Artifacts
<i>maven</i>	<i>target</i>	*.jar

Create a Project

Projects represent the components and applications that can be built individually for release or distribution. Projects often correspond to how development is grouped in your source control management system. Projects can be composites of many other projects using a dependency management system.

Projects are built from templates. In this tutorial, you will use the template you created earlier to create your project, and you will supply final definition to the template's source and process configurations.

To Create a Project

1. Display the New Project dialog (*Projects > Create button*).
2. Select the template you created earlier--*Maven Build Template*. Fields tailored to the selected template are displayed. The Maven Flags and Maven Goals fields are derived from the properties you defined when you created the template.

Figure 15. New Project Dialog

The 'New Project' dialog box contains the following fields and values:

- Name:** Terraform (with a red asterisk indicating it is a required field)
- Description:** (empty text box)
- Template:** Maven Build Template
- Team:** euroBuild Team (dropdown menu)
- Maven Flags:** default (with a red asterisk indicating it is a required field)
- Maven Goals:** package (with a red asterisk indicating it is a required field)

Buttons at the bottom: Save, Cancel.

3. Enter a name for the project, to follow the tutorial, enter *Terraform*.
4. In the Maven Flags field, enter *-P ubuild-dist*. This is the Maven activate profile option.
5. Enter *verify* in the Maven Goals field. *Verify* is a basic Maven phase and will cause Maven to run validation checks.

To Configure a Project Build Process

A project must be associated with a build process, which is done by assigning a build process template to the project (remember, a template might have any number of build process templates associated with it).

1. Display the New Build Process dialog (*Project > Terraform > Configuration > New Build Process button*).
2. From the Template drop-down list box, select template you created earlier--*Maven Build Process*. Fields tailored to the selected template are displayed. The Maven Artifact ID and Maven Group ID fields are derived from the properties you defined when you created the template.
3. Use the values from the following table to complete the dialog.

Table 16.

Field	Description
Name	To follow the tutorial, enter <i>Maven Build Process</i>
Maven Artifact ID	<i>terraform</i>
Maven Group ID	<i>Acom.urbancode.terraform</i>

To Configure a Project Build Source

A project build process must be associated with a source, which is done by assigning a source template to the process (remember, a template might have any number of source templates associated with it).

1. On the Configuration pane, use the Create Source Configuration button (*Project > selected_project > selected_build_process > Configuration*).
2. From the Source Template drop-down list box, select the source template you created earlier--*GitHub*. Fields tailored to the selected template are displayed. The Branch and Project fields are derived from the properties you defined when you created the template.

3. Use the values from the following table to complete the dialog.

Table 17.

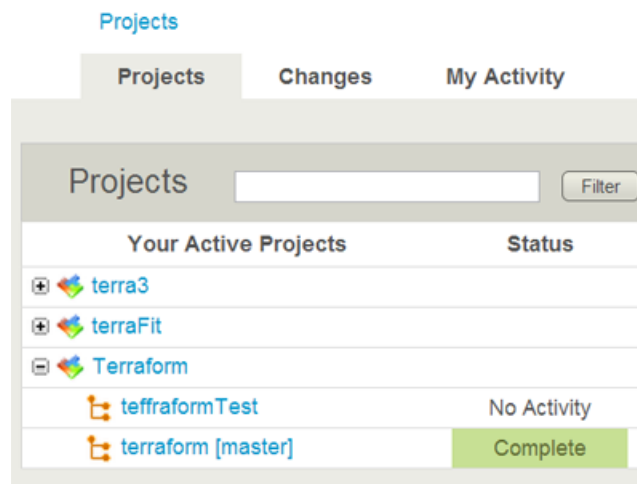
Field	Description
Source Name	<i>GitHub</i>
Repository	<i>GitHub</i>
Branch	<i>master</i>
Project	<i>UrbanCode/terraform.git</i>

Run the Build

To Run a Build

1. On the Projects page, expand your project to display the build process you created, as shown in the following illustration. A project can have any number of build processes.

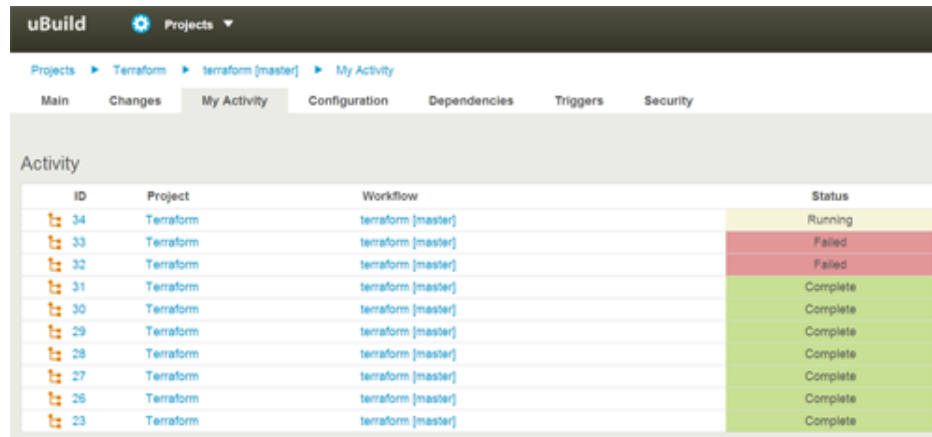
Figure 16. Project Listing



2. Select the process you created for the tutorial.
3. On the build process page, use the Build button.

When a process begins running, the My Activity pane is displayed which provides status information about the process.

Figure 17. My Activity Tab



The screenshot shows the uBuild interface with the 'My Activity' tab selected. The breadcrumb trail is 'Projects > Terraform > terraform [master] > My Activity'. Below the breadcrumb, there are tabs for 'Main', 'Changes', 'My Activity' (active), 'Configuration', 'Dependencies', 'Triggers', and 'Security'. The 'Activity' section contains a table with the following data:

ID	Project	Workflow	Status
34	Terraform	terraform [master]	Running
33	Terraform	terraform [master]	Failed
32	Terraform	terraform [master]	Failed
31	Terraform	terraform [master]	Complete
30	Terraform	terraform [master]	Complete
29	Terraform	terraform [master]	Complete
28	Terraform	terraform [master]	Complete
27	Terraform	terraform [master]	Complete
26	Terraform	terraform [master]	Complete
23	Terraform	terraform [master]	Complete

Refresh the window until the status changes to *Failed* or *Complete*.

Review




















































In this walk through, you:

1. Created a Git repository.
2. Created a workflow and associated an agent pool to it.
3. Created a job that checked-out the Terraform source from GitHub and built Terraform with Maven, creating a Maven repository.
4. Assigned the job to the workflow.
5. Created a template that defined the *maven-flag* and *maven-goals* properties.
6. Using the template's source configuration, you defined the *branch* and *project* properties.
7. Using the template's build process configuration, you defined the *maven.artifactid* and *maven.groupid* properties.
8. Also using the template's build process configuration, you defined the *maven* artifact set.
9. Next, you created a project based on the template and supplied values for the *maven-flag* and *maven-goals* properties.
10. Supplied values for the *Maven Artifact ID* and *Maven Group ID* properties when you configured the project's build process.
11. Supplied values for the *branch* and *project* properties when you configured the project's source.
12. Finally, you ran a successful build.

Here is a screen shot of the finished job:

Figure 18. Finished Maven Build Job

Name: Maven Build Job
Description: Quickstart demo job.

Name	Type	Pre-Condition	Actions
Create Stamp	Stamp	All Prior Success	     
Clean Workspace	Cleanup Source Plugin Meta	All Prior Success	     
Populate Workspace	Populate Workspace Source Plugin Meta	All Prior Success	     
Get Changelog	Get Changelog Source Plugin Meta	All Prior Success	     
Maven Build	Maven - Maven Build	All Prior Success	     
Upload Artifacts	CodeStation - Upload Artifacts	All Prior Success	     
Publish JUnit Results	JUnit - JUnit Report	Always	     
Assign Maven Version	Groovy - Run Groovy Script	All Prior Success	     
Assign Success	Assign Status	All Prior Success	     
Assign Failure	Assign Status	Any Prior Failure	     

Reference

uBuild Plug-ins

Overview of uBuild Plug-ins

uBuild plug-ins provide tools for creating build processes. Plug-ins consist of configurable *steps* which can be thought of as distinct pieces of automation. By combining steps in the uBuild web application, you can create fully-automated build processes.

A plug-in consists of a number of steps, which varies from plug-in to plug-in. A step has a variable number of properties, a command that performs the function associated with the step, and post-processing instructions (typically used to ensure that expected results occurred). Step properties can serve a wide variety of purposes, from providing information required by the step's command, to supplying some or all of the actual command itself. Property values can be supplied when defining a workflow configuration or parameterized to be entered at process run-time.

Plug-ins at Run-time

Build processes are run by agents installed in the target environment. For a process to run successfully, the agent must have access to all resources, tools, and files required by the plug-in steps used in the process. When installing an agent, ensure that:

- The agent running the process has the necessary user permissions to execute commands and access any required resources. This typically entails granting permissions if an external tool is installed as a different user; installing the agent as a service; or impersonating the appropriate user.
- Any external tools required by plug-in steps are installed in the target environment.
- The required minimum version of any external tool is installed.

Creating Plug-ins

A plug-in consists of two mandatory XML files, `plugin.xml` and `upgrade.xml`, along with any supporting script files required by the plug-in. The `plugin.xml` file defines the steps comprising the plug-in. Each step is an independently configurable entity in the uBuild web application. The `plugin.xml` file also contains a variable number of property groups depending on the plug-in type.

A step is defined by a `<step-type>` element which contains: one `<properties>` element, one `<command>` element, and one `<post-processing>` element. The `<properties>` element is a container for a variable number of `<property>` child elements. The `<property>` elements define user-configurable properties that will be passed to the step, as well as how they are presented in the uBuild editor. Property values can be supplied at design-time in the uBuild editor or run-time. The `<post-processing>` element provides error-handling capabilities and sets property values that can be used by other steps. The `<command>` element performs the step's function. The function can be defined completely by the element, or be constructed in part or entirely from the step's properties at design-time or run-time.

In addition to a step's own properties, a command has access to other properties: those set earlier by other steps within the process, to properties related to the configuration of the process, as well as to those of the agent executing the step.

The `upgrade.xml` file is used to upgrade the plug-in to a new version. Optionally, you can include an `info.xml` file which contains a version ID to be shown in the UI as well as other information used by the UrbanCode plug-in page. Although optional, UrbanCode recommends the use of the `info.xml` file.

Plug-in steps are performed by an agent installed in the target environment, which means that plug-ins can be written in any scripting language as long as the agent can access the required scripting tools on the host. Once a plug-in is created, upload it into uBuild to make it available to users. To upload a plug-in, create a ZIP archive that contains the XML files (plugin.xml and upgrade.xml) along with any scripts required by the plug-in, then import the ZIP file.

The plugin.xml File

The plugin.xml file conforms to the PluginXMLSchema_v1.xsd schema definition

The structure of the plugin.xml file consists of a <header> element and one or more <step-type> elements. The <header> identifies the plug-in. Underneath the header element you will find two important child elements: <identifier> and <server:plugin-type>. The <identifier> element has an id attribute that specifies an unchanging value which identifies the plug-in; <server:plugin-type> specifies the plug-in type, either Source or Automation. The id attribute on the <identifier> element and the <server:plugin-type> element cannot be changed across plug-in versions. Each <step-type> element defines a step; steps are available to users in the uBuild process editor and are used to construct processes.

A plug-in also has a varying number of <property-group> elements. A Source type plug-in has both a Source and Repository <property-group> element. The Automation type may have an Automation <property-group>.

After the document type declaration, the plugin root element identifies the XML schema type, PluginXMLSchema_v1.xsd, which is used by all plug-ins. The following presents the basic structure of plugin.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://www.urbancode.com/PluginXMLSchema_v1"
        xmlns:server="http://www.urbancode.com/PluginServerXMLSchema_v1"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <header>
    <identifier id="{plugin_id}" version="{version}" name="{plug-in name}"/>
    <description/>
    <tag>Plugin_type/Plugin_subtype/Plugin_name</tag>
    <server:plugin-type>{Source|Automation}</server:plugin-type>
  </header>
  <step-type name="{Step_Name}">
    <description/>
    <properties>
      <server:property name="{property_name}" required="{true|false}">
        <property-ui type="{type}" label="{label_displayed_in_UI}"
          description="{description}"
          default-value="{default}"
          hidden="{true|false}"/>
      </property>
    </properties>
    <post-processing><![CDATA[
      if (properties.get("exitCode") != 0) {
        properties.put("Status", "Failure");
      }
      else {
        properties.put("Status", "Success");
      }
    ]]></post-processing>
  </step-type>
</plugin>
```

```

    }
  ]]></post-processing>

  <command program="{path_to_tool}">
    <arg value="{parameters_passed_to_tool}"/>
    <arg path="{path_to_required_files}"/>
    <arg file="{command_to_run}"/>
    <arg file="{PLUGIN_INPUT_PROPS}"/>
    <arg file="{PLUGIN_OUTPUT_PROPS}"/>
  </command>
  <server:type>{server type}</server:type>
  <server:validation language="javascript"/>
</step-type>
{one or more of <server:property-group>}
</plugin>

```

Note

After the `<step-type>` elements, a plug-in may have one or more `<server:property-group>` elements. See the section called “The `<server:property-group>` Element”.

The `<header>` Element

The mandatory header element identifies the plug-in and contains three child elements:

Table 18. Plugin - plugin.xml `<header>` Element

<code><header></code> Child Elements	Description
<code><identifier></code>	<p>This element's three attributes identify the plug-in:</p> <ul style="list-style-type: none"> <i>version</i> API version (the version number used for upgrading plug-ins is defined in the info.xml file). <i>id</i> Identifies the plug-in. <i>name</i> The plug-in name appears on uBuild's web application and on the urbancode.com plug-in page. <p>All values must be enclosed within single-quotes or double-quotes.</p>
<code><description></code>	Describes the plug-in; appears on uBuild's web application and on the urbancode.com plug-in page.
<code><tag></code>	Defines where the plug-in is listed within the uBuild hierarchy of available plug-ins steps. The location is defined by a string path separated by slashes. For example, the Ant plug-in tag is: Build/Build Tools/Ant. The Ant steps will be listed beneath the Ant item, which is in a Build Tools item, which is in the Build item.
<code><server:plugin-type></code>	uBuild supports two plug-in types:

<header> Child Elements	Description
	<p>Automation: General purpose plugin type that is used for integrations with build tools, issue trackers, testing frameworks, and many others.</p> <p>Source: Plugin type specifically for integrating with source control management (SCM) systems.</p>

The following is a sample header definition:

```
<header>
  <identifier id="com.urbanocode.plugin.accurev" name="Accurev" version="1"/>
  <description>
    The Accurev plugin automates populating an Accurev workspace,
    creating a tag, and publishing source changes to the
    Changes tab of the Build Life.
  </description>
  <tag>SCM/Accurev</tag>
  <server:plugin-type>Source</server:plugin-type>
</header>
```

Plug-in Steps--the <step-type> Element

Plug-in steps are defined with the <step-type> element; each <step-type> represents a single step in the uBuild web application. A <step-type> element has a name attribute and several child elements: <description>, <properties>, <command>, <post-processing>, <server:type>, and <server:validation>.

The mandatory name attribute identifies the step. The description and name appear in uBuild's web application and on the urbanocode.com plug-in page.

Step Properties--the <properties> Element

Each step's single <properties> element serves as a container for properties which are defined with the <property> tag. A <properties> element can contain any number of <property> child elements.

A <property> tag has a mandatory name attribute, optional required attribute, and two child elements, <server:property-ui> and <server:value>, which are defined in the following table.

Table 19. Plugin - plugin.xml <property> Element

<property> Child Elements	Description
<server:property-ui>	<p>Defines how the property is presented to users in the uBuild web application. This element has several attributes:</p> <ul style="list-style-type: none"> <i>label</i> Identifies the property in the web application. <i>description</i>

<property> Child Elements	Description
	<p>Text displayed to users.</p> <ul style="list-style-type: none"> • <i>default-value</i> <p>Property value displayed in the uBuild editor; used if unchanged.</p> <ul style="list-style-type: none"> • <i>hidden</i> <p>Hides the property from the UI while still allowing access to the property in the plugin scripts.</p> <ul style="list-style-type: none"> • <i>type</i> <p>Identifies the type of widget displayed to users. Possible values are:</p> <ul style="list-style-type: none"> • <i>textBox</i> <p>Enables users to enter an arbitrary amount of text, limited to 4064 characters.</p> <ul style="list-style-type: none"> • <i>textAreaBox</i> <p>Enables users to enter an arbitrary amount of text (larger input area than <i>textBox</i>), limited to limited to 4064 characters.</p> <ul style="list-style-type: none"> • <i>secureBox</i> <p>Used for passwords. Similar to <i>textBox</i> except values are redacted.</p> <ul style="list-style-type: none"> • <i>checkBox</i> <p>Displays a check box. If checked, a value of <i>true</i> will be used; otherwise the property is not set.</p> <ul style="list-style-type: none"> • <i>selectBox</i> <p>Requires a list of one or more values which will be displayed in a drop-down list box. Configuring a value is described below.</p> <ul style="list-style-type: none"> • <i>multiSelectBox</i> <p>Similar to <i>selectBox</i> but allows multiple selections.</p> <ul style="list-style-type: none"> • <i>sourcePropSheetRef</i> <p>Enables steps to access configured source configurations. This is required for all Source plugin steps and must always be hidden.</p> <ul style="list-style-type: none"> • <i>automationPropSheetRef</i> <p>Enables steps to access configured automation tools. This is required if an Automation plugin has an Automation property group and must always be hidden.</p>

<property> Child Elements	Description
<code><server:value></code>	Used to specify values for a selectBox or multiSelectBox. Each value has a mandatory label attribute which is displayed to users, and a value used by the property when selected. Values are displayed in the order they are defined.

Here is a sample `<property>` definition:

```
<property name="onerror" required="true">
  <server:property-ui type="selectBox"
    default-value="abort"
    description="Action to perform when statement fails: continue, stop, abort."
    label="Error Handling"/>
  <server:value label="Abort">abort</server:value>
  <server:value label="Continue">continue</server:value>
  <server:value label="Stop">stop</server:value>
</property>
```

The `<post-processing>` Element

When a plug-in step's `<command>` element finishes processing, the step's mandatory `<post-processing>` element is executed. The `<post-processing>` element sets the step's output properties and provides error handling. The `<post-processing>` element can contain any valid JavaScript script (unlike the `<command>` element, `<post-processing>` scripts must be written in JavaScript). Users can also provide their own scripts when defining the step in the uBuild editor. Although not required, it's recommended that scripts be wrapped in a CDATA element.

The `<post-processing>` element can examine the exit code of the tool invoked by the `<command>` element or the step's output log, and take actions based on the result. For example, uBuild sometimes uses a scanner object to scan the step's output on the agent (scanning occurs on the agent) and take actions dependent on scan results. In the following code fragment, `scanner.register()` registers a string with a regular expression engine, then takes an action if the string is found. Once all strings are registered, it calls `scanner.scan()` on the step's output log.

```
<![CDATA[
  properties.put("Status", "Success");
  if (properties.get("exitCode") != 0) {
    properties.put("Status", "Failure");
  }
  else {
    scanner.register("(?i)ERROR at line", function(lineNumber, line) {
      var errors = properties.get("Error");
      if (errors == null) {
        errors = new java.util.ArrayList();
      }
      errors.add(line);
      properties.put("Error", errors);

      properties.put("Status", "Failure");
    });
  }
}]
```

```
.
.
scanner.scan();

var errors = properties.get("Error");
if (errors == null) {
    errors = new java.util.ArrayList();
}
properties.put("Error", errors.toString());
}
]]
```

The <command> Element

Steps are executed by invoking the scripting tool or interpreter specified by the <command> element. The <command> element's `program` attribute defines the location of the tool that will perform the command. It bears repeating that the tool must be located on the host and the agent invoking the tool must have access to it. In the following example, the location of the tool that will perform the command--the Java-based scripting tool *groovy* in this instance--is defined.

```
<command program='${GROOVY_HOME}/bin/groovy'>
```

The actual command and any parameters it requires are passed to the tool by the <command> element's <arg> child element. Any number of <arg> elements can be used. The <arg> element has several attributes:

Table 20. Plugin - plugin.xml <arg> Attributes

Attribute	Description
<value>	Specifies a parameter passed to the tool. Format is tool-specific; must be enclosed by single-quotes or double-quotes.
<path>	Path to files or classes required by the tool. Must be enclosed by single-quotes or double-quotes.
<file>	Specifies the path to any files or classes required by the tool. Format is tool-specific; must be enclosed by single-quotes or double-quotes.

Because <arg> elements are processed in the order they are defined, ensure the order conforms to that expected by the tool.

```
<command program='${GROOVY_HOME}/bin/groovy'>
  <arg value='-cp' />
  <arg path='classes:${sdkJar}:lib/commons-codec.jar:
    lib/activation-1.1.1.jar:
    lib/commons-logging.jar:lib/httpclient-cache.jar:
    lib/httpclient.jar:lib/httpcore.jar:
    lib/httpmime.jar:lib/javamail-1.4.1.jar' />
  <arg file='registerInstancesWithLB.groovy' />
  <arg file='${PLUGIN_INPUT_PROPS}' />
  <arg file='${PLUGIN_OUTPUT_PROPS}' />
```

</command>

The `<arg file='${PLUGIN_INPUT_PROPS}' />` specifies the location of the tool-supplied properties file. The `<arg file='${PLUGIN_OUTPUT_PROPS}' />` specifies the location of the file that will contain the step-generated properties.

Note: new lines are not supported by the `<arg>` element and are shown in this example only for presentation.

The `<server:type>` and `<server:validation>` Element

Each Source type `<step-type>` element must have a sub-type. Sub-types are defined with the `<server:type>` element. Sub-types are:

- **SCM_Populate** Must have a `<property name="date">` property which is used to provide a date for checkout.
- **SCM_Cleanup** has no specific requirements.
- **SCM_Changelog** Must have `<property name="startDate">` and `<property name="endDate">` properties used to define the interval for which a changelog will be produced.
- **SCM_Label** Must have `<property name="label">` and `<property name="message">` properties used for the label and message during the step action.
- **SCM_QuietPeriod** Must have `<property name="startDate">` and `<property name="endDate">` properties used to define the interval for examining changes.

All SCM_ sub-types must have a `<property name="source">` property with a `hidden="true"` attribute.

The `<server:validation>` element has a `language` attribute that denotes the language of the validation script. The validation script is used to perform both validation at configuration time and at run-time.

The `<server:property-group>` Element

The `<server:property-group>`, like the `<properties>` element, defines the user-configured properties passed to the step. The `<server:property-group>` element is a container for properties which are defined with the `<server:property>` tag. A `<server:property-group>` element can contain any number of `<server:property>` child elements.

The `<server:description>` appears in uBuild's web application and on the urbancode.com plug-in page.

The `<server:property-group>` element's `type` attribute specifies the group's type. Supported values are: Source, Repository, and Automation. A Source type plug-in must have one Source and one Repository type element and a Source type element must have a `repo` property that corresponds to the Repository type element. An Automation type may have an Automation type element.

A `<server:property>` tag has a mandatory `name` attribute, optional `required` attribute, and two child elements, `<server:property-ui>` and `<server:value>`, which are defined in the following table.

Table 21. Plugin - plugin.xml <server:property> Element

<server:property> Child Elements	Description
<server:property-ui>	<p>Defines how the property is presented to users in the uBuild web application. This element has several attributes:</p> <ul style="list-style-type: none"> <i>label</i> Identifies the property in the web application. <i>description</i> Text displayed to users. <i>default-value</i> Property value displayed in the uBuild editor; used if unchanged. <i>hidden</i> Hides the property from the UI while still allowing access to the property in the plugin scripts. <i>type</i> Identifies the type of widget displayed to users. Possible values are: <ul style="list-style-type: none"> <i>textBox</i> Enables users to enter an arbitrary amount of text, limited to 4064 characters. <i>textAreaBox</i> Enables users to enter an arbitrary amount of text (larger input area than <i>textBox</i>), limited to limited to 4064 characters. <i>secureBox</i> Used for passwords. Similar to <i>textBox</i> except values are redacted. <i>checkBox</i> Displays a check box. If checked, a value of <i>true</i> will be used; otherwise the property is not set. <i>selectBox</i> Requires a list of one or more values which will be displayed in a drop-down list box. Configuring a value is described below. <i>multiSelectBox</i> Similar to <i>selectBox</i> but allows multiple selections.

<code><server:property></code> Child Elements	Description
	<ul style="list-style-type: none"> <code>repoPropSheetRef</code> <p>Enables steps to access configured repositories. This only applies to Source property groups in a Source plugin. Properties of this type should have a name attribute of "repo."</p>
<code><server:value></code>	Used to specify values for a selectBox and multiSelectBox. Each value has a mandatory <code>label</code> attribute which is displayed to users, and a <code>value</code> used by the property when selected. Values are displayed in the order they are defined.

```

<server:property-group type="{Repository|Source|Automation}" name="{name}">
  <server:description>{description}</server:description>
  <server:property name="{name}" required="{true|false}">
    <server:property-ui type="{type}"
      label="{label}"
      description="{description}"
      default-value="{value}"/>
  </server:property>
  .
  .
  .
</server:property-group>

```

Upgrading Plug-ins

To create an upgrade, first, increment the number of the `version` attribute of the `<identifier>` element in `plugin.xml`. Next, create a `<migrate>` element in `upgrade.xml` with a `to-version` attribute containing the new number. Finally, place the property and step-type elements that match the updated `plugin.xml` file within this element, as shown in the following example.

```

<?xml version="1.0" encoding="UTF-8"?>
<plugin-upgrade
  xmlns="http://www.urbancode.com/UpgradeXMLSchema_v1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <migrate to-version="2">
    <migrate-command name="Run SLQPlus script"/>
  </migrate>
  <migrate to-version="3">
    <migrate-command name="Run SQLPlus Script" old="Run SLQPlus script">
      <migrate-properties>
        <migrate-property name="sqlFiles" old="sqlFile" default="*.sql"/>
      </migrate-properties>
    </migrate-command>
  </migrate>
</plugin-upgrade>

```

```
</migrate>
<migrate to-version="4">
  <migrate-command name="Run SQLPlus Script">
    <remove-properties>
      <remove-property name="sqlFiles"/>
    </remove-properties>
  </migrate-command>
</migrate>
<migrate to-version="5">
  <remove-command name="Run SQLPlus Script"/>
</migrate>
</plugin-upgrade>
```

Of course, you can also make a script-only upgrade, that is, an upgrade that contains changes to the step's associated scripts and files but does not change plugin.xml. This mechanism can be useful for plug-in development and for minor bug-fixes/updates.

The info.xml File

Use the optional info.xml file to describe the plug-in and provide release notes to users. The file's <release-version> element can be used for version releases and is the version displayed in the UI.

System Settings

Installing Plug-ins

The Plugins page (*System > Plugins*) enables you to install or delete plug-ins, and provides information about the installed plug-ins. Plug-ins can be installed at any time. Download plug-ins from UrbanCode's plug-in page:

<http://plugins.urbancode.com>

To install a plug-in:

1. Download the plug-in from the UrbanCode plug-in page. Plug-ins are provided in compressed format. There is no need to decompress the file.

You can also load your own plug-ins. For information about creating plug-ins, see *uBuild Plug-ins*.

2. Enter the path to the downloaded file, or use the Choose File button to select it.
3. Once the file is specified, use the Load button to load it.

If the plug-in loaded successfully, it will be displayed in the Loaded Plug-ins list as soon as the process finishes. Once installed, plug-in functionality is available immediately.

uBuild Scripting

Scripting documentation is available in the UI of the installed uBuild server by clicking on the tools link on the upper-right corner of any page.

Index

A

- additional options for steps, 34
- agent
 - installing, 23
- agent pool, 29
- Artifact Configurations pane, 42
- Assign Status step, 38

B

- Build button, 44
- build life
 - build lives, 5
- build process
 - build processes, 4
- Build Process Templates button, 41

C

- CodeStation
 - file storage, 9
- CodeStation plug-in, 36
- concepts
 - overview, 4
- Create a Project Template dialog, 40
- Create Source Configuration button, 43
- Create Source Templates button, 41

D

- dependency
 - dependencies, 5
- digital certificates, 14

G

- Get Changelog Source Plugin Meta dialog, 35
- Git, 29
- Groovy, 29
- Groovy plug-in, 38

I

- Insert Job button, 39
- Insert Start Job dialog, 39
- installing agents, 23
- installing plug-ins, 59

J

- java.util.Properties, 38
- JavaScript Object Notation, 7
- JMS communication, 7
- Job icon, 39
- JSON, 7

- JUnit, 37

K

- keystore, 26, 26
- keytool, 26

L

- LOCAL SYSTEM account, 24

M

- Maven, 29
- Maven plug-in, 36
- mutual authentication, 14, 26
- My Activity pane, 44

N

- New Artifact Config button, 42
- New Build Process dialog, 43
- New Build Process Template dialog, 41
- New Project dialog, 42

O

- Oracle
 - installing, 19
 - supported editions, 18

P

- plug-in
 - installing, 59
- plugins.urbancode.com, 59
- Populate Workspace Source Plugin Meta dialog, 35
- Pre-Condition Script field, 37, 39
- project
 - projects, 4

R

- relocating CodeStation
 - relocating file storage, 10
- repository
 - repositories, 5
- REST-based user interface, 8

S

- secondary process
 - secondary processes, 6
- secure socket layer, 25
- server
 - user account, 16
- source config
 - source configurations, 4
- source type, 41
- SSL configuration, 25

SSL mutual key-based authentication, 14
standard out, 21
Start Action icon, 39
system settings, 59
 installing plug-ins, 59
System Team, 30

T

template, 40
 templates, 5
Terraform, 29
testing jobs, 37

U

UrbanCode Plug-in Page, 59

W

workflow
 workflows, 5