Overview of UrbanCode Deploy security

Before you use this information and the product it supports, read the information in "Notices" on page 26.

Contents

Introduction

This document describes the UrbanCode™ Deploy security features. It highlights core elements and examines the available options. The paper also discusses typical architectural topologies and communication protocols.

Links to related information in the Knowledge Center are provided are provided through the paper.

Overview of UrbanCode Deploy architecture

UrbanCode™ Deploy uses a client-server model with a network of agents and relays that communicate with target systems. The server provides the web-based front-end and core services, such as workflow, agent management, and security. Services can be consumed by clients and agents or other services. Deployments are orchestrated by the server and performed by agents distributed throughout the network. Clients access the server through web browsers, the REST API, or the command-line client.

Server-agent communication is done with Java Message Service (JMS) and HTTPS protocols. JMS agent is an earlier type, where JMS protocol is used for basic commands and information that is exchanged between the server and an agent. HTTPS protocol is used for file transfers between the server and agents, such as uploading or downloading component version artifacts. Introduced with UrbanCode™ Deploy 7.0.0 version, Web agents allows you to use the WebSocket protocol, which is a bi-directional, point-to-point message protocol. WebSocket connections start as HTTP requests and upgrade to WebSocket protocol.

Many services use representational state transfer (REST). REST services are web services that transfer resources over HTTPS. Resources are transferred by a self-describing format such as XML or JSON (JavaScript Object Notation). The XML and JSON representations typically model resource states at the time of agent and client requests. REST-style services achieve statelessness by ensuring that requests include all the data needed by the server to make a coherent response.

The data tier's relational database stores configuration and run-time data. In general, you configure security for the database on the database server, not in UrbanCode Deploy. UrbanCode Deploy communicates with the database by using the Java Database Connectivity (JDBC). The data tier's file store, CodeStation, contains versioned artifacts. Reporting tools can connect directly to the relational database.

Topologies

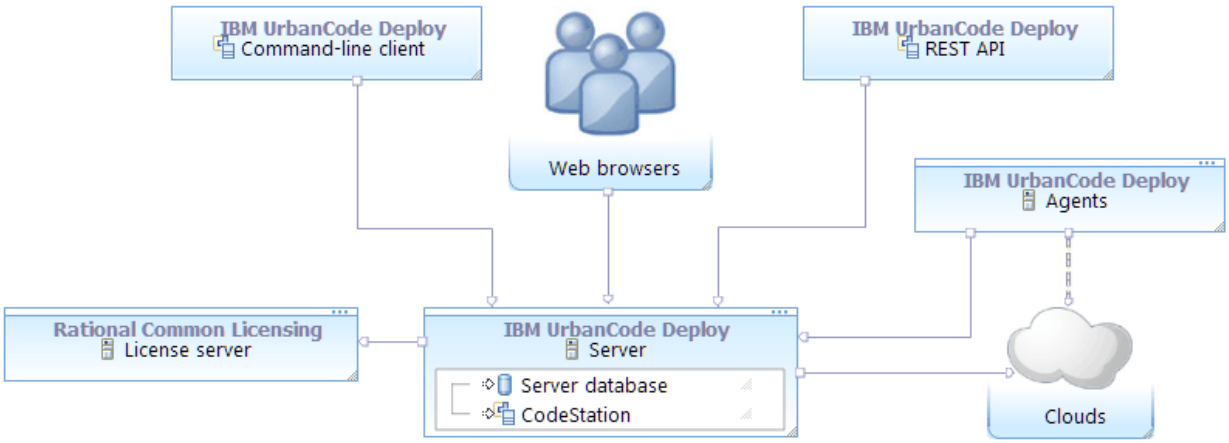
UrbanCode Deploy typically includes a server and one or more agents. You can configure multiple topologies, including ones that use high availability, containerized servers running in IBM Cloud Private, Blueprint Designer, and disaster recovery to meet your needs.

Core topology

The core installation of UrbanCode Deploy includes a server, agents, database, and a license server. Clients access the server through web browsers, the REST API, or the command-line client. Agents can be installed on cloud environments, virtual machines (VMs), containers, or physical systems. With this topology, the server can create environments on clouds that use virtual system patterns (VSPs), such as IBM® Cloud Orchestrator and IBM PureApplication® System. To create environments on other clouds such as Amazon Web Services, SoftLayer®, VMware vCenter, and Microsoft Azure, you must install the blueprint design server and at least one engine, as described in the blueprint design topologies.

Figure 1 illustrates the core topology.

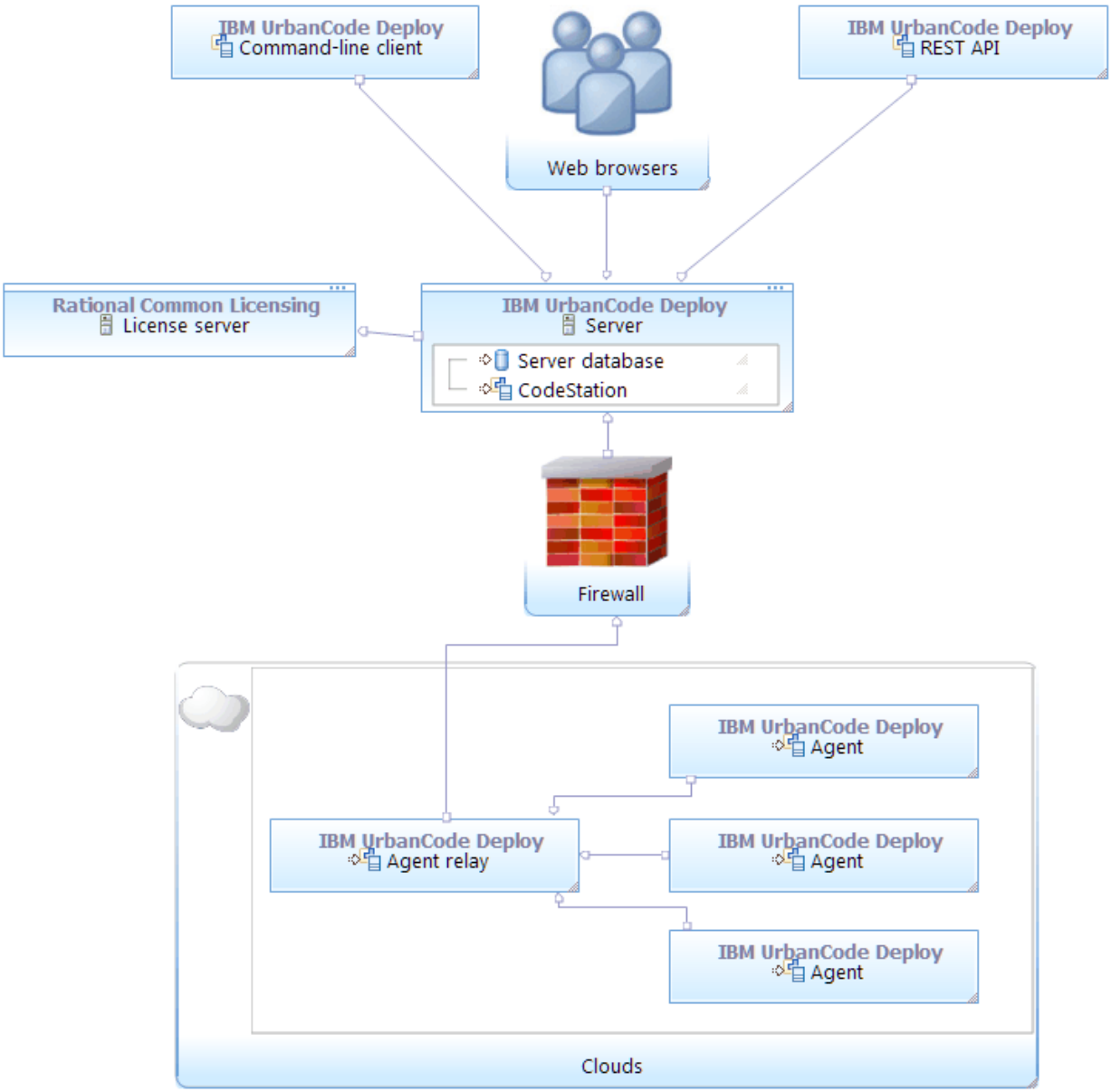
Figure 1. Core topology



Multi-region topology

If your environment has multiple security zones that are divided by firewalls, you can use agent relays to connect agents to the server through the firewalls. You can install an agent relay outside the firewall to allow agents on those environments to connect to the server through the firewall, as shown in Figure 2.

Figure 2. Multi-region topology

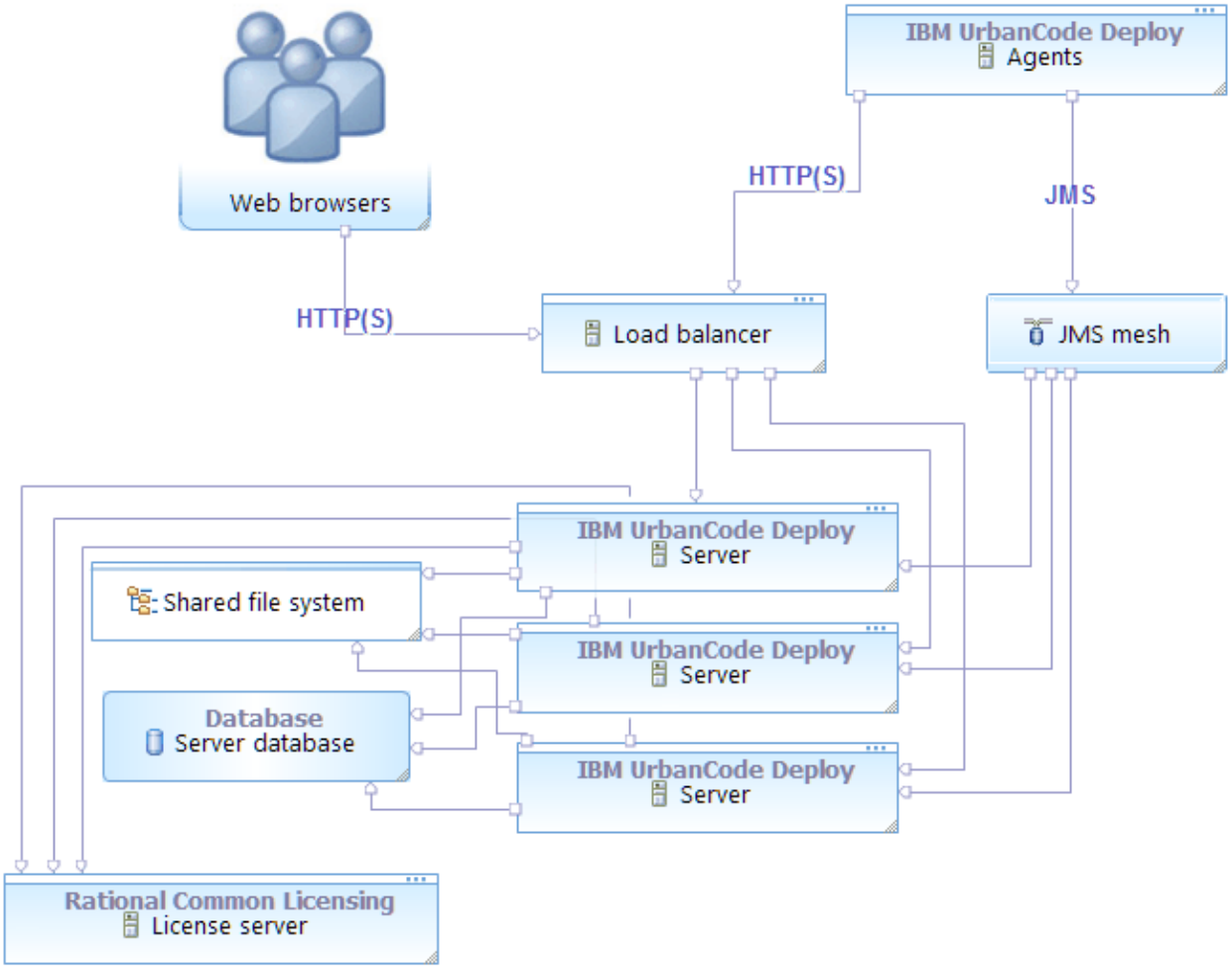


Remote agents open connections to the agent relay. Agents in the same network as the server continue to communicate directly to the server.

High-availability clustered topologies

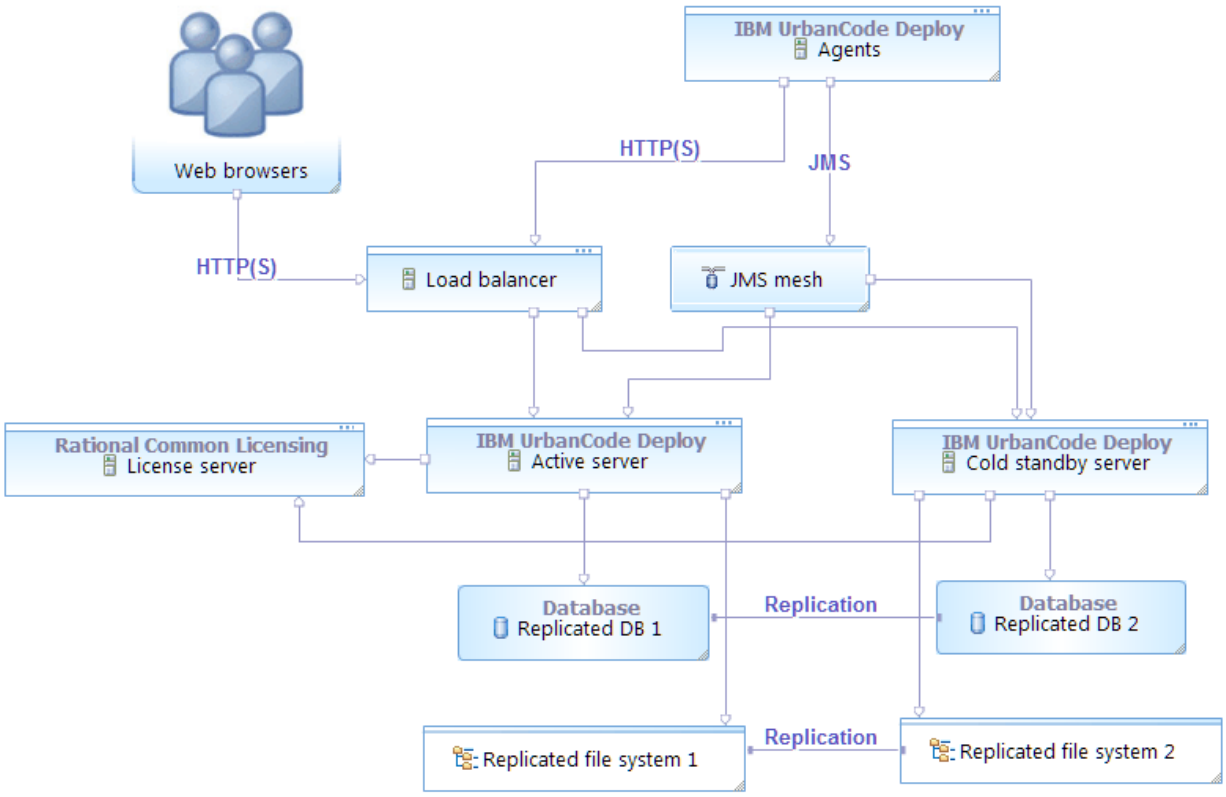
High-availability topologies use multiple servers. These servers can all be running at the same time to share the load (as in a clustered topology), or they can be waiting for another server to fail (as in a cold standby topology).

Typically, customers who employ a clustered topology use a load balancer to distribute work. Users connect directly to the load balancer, which sends them to an active server. When using HTTP or HTTPS, agents connect to the load balancer. When using JMS, agents connect to servers or relays through a JMS mesh. The servers store their files on a shared database and file system.



Disaster recovery topologies

One way to prepare for disaster recovery is to have a cold standby system, including a stopped server and a replicated copy of the database and file system. When the primary system fails, the cold standby is brought online and promoted to primary server. Once online, the standby reestablishes connections with all agents, performs recovery, and proceeds with any queued processes.

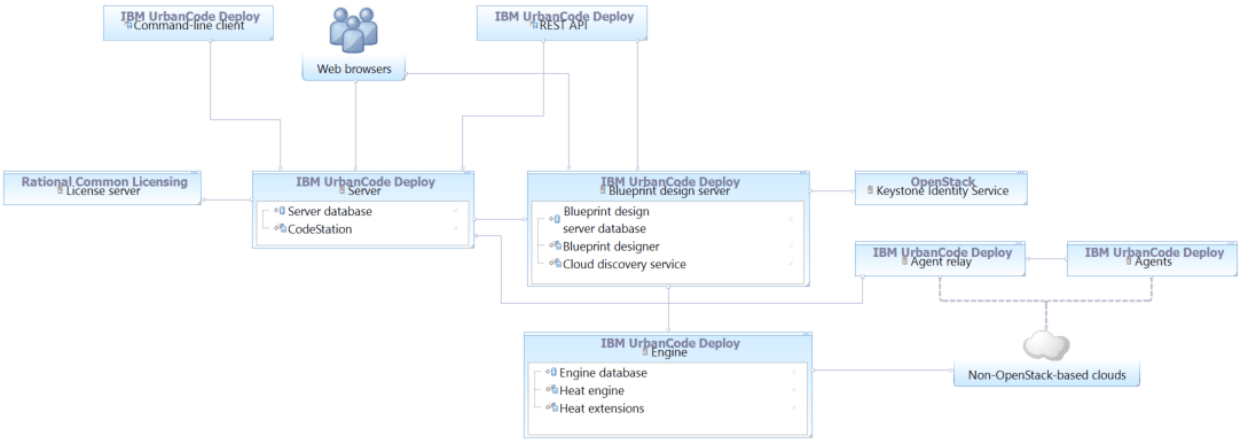


Blueprint design topology

To work with blueprints on clouds using OpenStack Heat, including OpenStack-based clouds, Amazon Web Services, SoftLayer, VMware vCenter, and Microsoft Azure, you use a topology that includes the blueprint design server and engine.

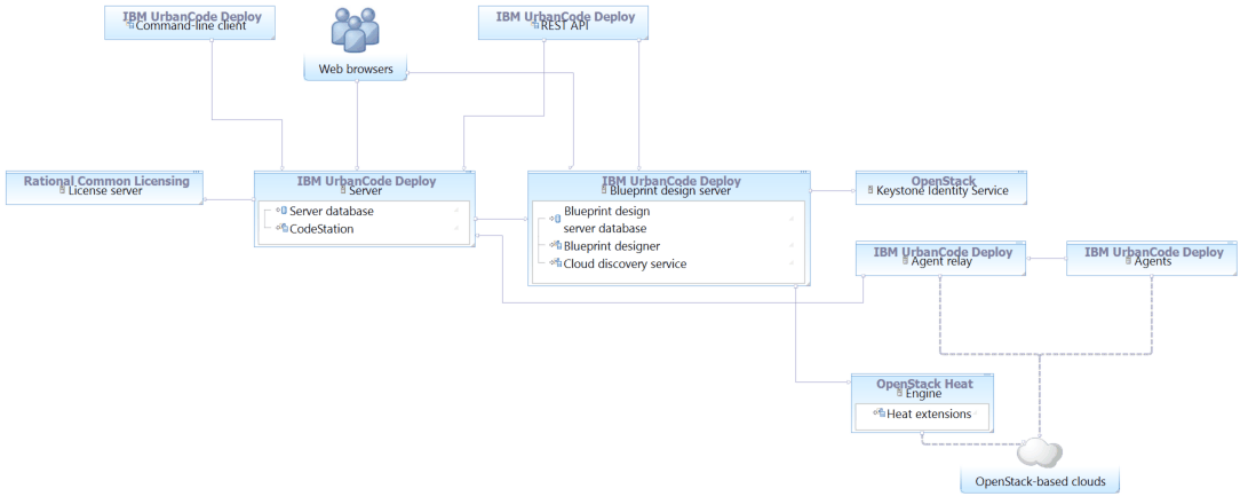
Installing a standalone engine

To deploy environments to non-OpenStack clouds, such as Amazon Web Services, SoftLayer, VMware, Google Cloud, and Microsoft Azure, you install the blueprint design server and Heat engine through UrbanCode Deploy. The following diagram shows a typical topology for this use case:



Extending Heat orchestration engines

To deploy environments to clouds that are based on OpenStack, you install the blueprint design server and extend the Heat engine that is associated to the cloud. In this case, you do not use the Heat engine that comes with UrbanCode Deploy. The following diagram shows a typical topology for this use case:



Server security

UrbanCode Deploy uses several technologies to provide security. Some features cannot be disabled, such as Transport Layer Security (TLS) for JMS communication. Other features can be configured to meet your requirements. Some features are enabled by default, such as end-to-end JMS encryption, while others are disabled by default, such as mutual authentication.

TLS technology

TLS is a standard technology that secures client-server communication by encrypting communication. Data are encrypted before being sent and decrypted by the recipient. Communications cannot be deciphered or modified by third-parties.

TLS technology uses *digital certificates* to secure communications. A digital certificate is a cryptographically signed document that combines a cryptographic key with an organization's fully qualified domain name (FQDN).

Digital certificates use *public key cryptography* which works by using two keys to encrypt and decrypt messages. The *public key* is stored in the certificate and is available publicly. Messages that are encrypted with the public key can only be decrypted with the *private key*.

Digital certificates and private keys are stored separately in the server's key store. Agents and agent relays also maintain their own key stores. See "Key stores and trust stores."

How TLS works

When the server receives a communication request from a user or agent, the server presents its certificate to the requestor. If the requestor trusts the certificate, it uses the certificate's public key to encrypt session messages. The server then uses its private key to decrypt messages encrypted with its public key.

UrbanCode Deploy automatically uses TLS security for JMS-based communications. TLS technology is automatically activated when you install an agent relay or before you connect to the relay. TLS technology cannot be turned off for JMS communication.

During installation, you also have the option to use TLS technology for HTTPS communications. Most users implement this option unless they install UrbanCode Deploy for a trial or demo. If you do not implement TLS technology for HTTPS during installation, you can implement it later.

You can also configure agents to verify the server's identity for HTTPS communication. This option prevents agents from connecting to the server over HTTPS unless the certificate's Common Name (CN) matches the host name of the server. See, "Agent verification of server certificate for HTTPS communication."

TLS mutual authentication

With *mutual authentication*, messages are still encrypted but clients are also required to authenticate themselves by providing digital certificates to the server. In mutual authentication mode, servers, local agents, and agent relays provide digital certificates to one another. UrbanCode Deploy certificates can be self-signed or be from a trusted certificate authority (CA).

In mutual authentication mode, the server stores the certificate from each agent in its trust store and each agent stores the server's certificate in its trust store. If you use agent relays, certificates are swapped between agents, agent relays, and the server. If you use a certificate authority, the certificate exchanges are unnecessary. Messages are rejected if the recipient cannot find the sender's certificate in its trust store. See "Key stores and trust stores."

You can enable mutual authentication for JMS communications for additional security. See, "Mutual authentication."

Other features

In addition to TLS technology, UrbanCode Deploy provides an additional option for JMS-based communications. End-to-end JMS encryption ensures that agents cannot send messages to one another after they are connected to the JMS mesh. In addition, messages cannot be read or modified by other agents, relays, or anything else connected to the JMS mesh. See, "End-to-end JMS encryption."

Role-based access controls are available that determine the actions that users can perform. See "User security model."

Tokens

Tokens are randomly generated passwords. Tokens provide authentication for agents, agent relays, users, and external systems. Agents use tokens when they run process steps and communicate with the server and external services. Users can use tokens with the command-line interface (CLI) client instead of supplying a user name and password in certain situations

Tokens have the same permissions as the users for whom they are created. User accounts can be designed for a particular token. For example, you might create tokens for agent relays with the `agentRelay` user.

Tokens with administrator permissions are used in some situations:

- Some integrations require tokens with administrator permissions. For example, when integrating with UrbanCode Release, the token provided to UrbanCode Release must be created by a user with administrator permissions.
- When a user logs onto the CLI client by using the admin user account, a token is generated that expires after 24 hours and is then deleted.
- When processes run on agents as the admin.

CodeStation

UrbanCode Deploy provides an artifact repository, CodeStation. Artifacts represent deployable items such as files, images, database schemas, configuration materials, or anything else associated with a software project.

CodeStation tracks component versions and maintains a complete history for all components. Without the repository, artifacts would have to be pulled from network shares or some other system, increasing both security risks and the potential for error.

Files are transferred using HTTPS. At deployment time, existing files in the target location can be validated against a SHA256 hash and replaced if modified.

Server-agent communication

The UrbanCode Deploy server sends commands and files to the agents or agent relays distributed around the network. Agents exchange status information with the server. Depending on the type of data exchanged, communication is done by using the HTTPS, WebSocket, or JMS protocols.

There are two agent types. Web agents use WebSocket connections and HTTPS for agent-server communication. Web agents were introduced with version 7.0.0. JMS agent use JMS and HTTPS to communicate with the server. You determine agent type when you install the agent.

JMS agents use the JMS channel as the primary control channel. JMS agents use this channel to retrieve commands from the server. However, some JMS agent activities access the web tier with HTTPS. For example, posting logs, transmitting test results, or posting files to CodeStation use HTTPS.

Web agents use WebSocket connects for tracking agent status and notifications, and HTTPS for everything else.

Agent processes are described later, see “Agents.”

The server listens on web sockets for agents, agent relays, and clients. By default, the server listens on only four ports: port 7918 for JMS, 8080 for HTTP, 7919 for WebSocket, and 8443 for HTTPS. Agent relays listen on three ports for agent communication. The default ports used by agent relays are JMS 7916, HTTP proxy 20080, and Codestation uses the HTTP proxy port + 1.

To monitor the health of the agent worker process, the agent monitor process listens for communication from the agent worker process on a port bound to localhost. Agent worker processes do not listen on ports. The worker process uses JMS for system communications, and HTTP REST services when performing plug-in steps or retrieving information from the server. Credentials are provided by the JMS message that initiates execution of a plug-in step.

Because JMS connections are persistent, UrbanCode Deploy does not have to continually open and close ports, which enables the server to communicate with agents at any time while remaining secure and scalable.

Persistent server-agent communication provides significant benefits to performance, security, availability, and disaster recovery.

Default port reference:
http://www.ibm.com/support/knowledgecenter/SS4GSP_7.0.5/com.ibm.udeploy.doc/topics/c_securityover.html

JMS communications

Server-agent JMS communication is subscription-based. The server consumes agent messages from a queue. By default, the server uses port number 7918 for JMS communication. The certificate for TLS communication on the JMS port is stored in the `server.keystore` file. If mutual authentication is disabled, the file is used only to encrypt network traffic. See, "Key stores used for JMS communication."

End-to-end JMS encryption

End-to-end JMS encryption ensures that agents cannot send messages to one another after they are connected to the JMS mesh. It also ensures that JMS clients cannot read messages that are not sent to them. With end-to-end encryption enabled, you do not need mutual authentication. End-to-end encryption requires considerably less configuration than mutual authentication, while ensuring that agents do not accept messages from rogue servers or other malicious entities.

With the end-to-end JMS encryption feature, the server generates an encryption key the first time agents connect to the server. The agent stores the key in its *agent_installation*`/conf/agent/agent-key.properties` file.

Using the UrbanCode Deploy web-based dashboard, you can view agent API keys. If an agent is compromised, you can revoke the API key and prevent the agent from connecting to the server.

Reference:
http://www.ibm.com/support/knowledgecenter/SS4GSP_7.0.5/com.ibm.udeploy.install.doc/topics/ssl_addl_security_2.html

Mutual authentication

When mutual authentication mode is active, UrbanCode Deploy uses it for JMS-based server-agent communication. In this mode, the server provides a digital certificate to each agent and agent relay, and each agent and agent relay provides one to the server. Certificate checks are performed by the JVM according to its own algorithm. Certificates can be issued from a certification authority (CA) if desired, in which instance certificates do not have to be swapped. The integrity of the content is established by the mechanisms built into TLS technology. This mode can be implemented during installation or activated afterward.

Unless you use a certification authority, the server stores the certificate from each agent in its trust store, and each agent stores the server's certificate in its trust store. If you use agent relays, certificates are swapped between agents, agent relays, and the server. Connections are rejected if the recipient cannot find the sender's certificate in its trust store. See "Key stores and trust stores."

Digital certificates are exchanged by using the keytool utility that is provided with the Java developer kit. You export a participant's certificate and then import it into another participant's trust store.

When you use mutual authentication mode, you must turn it on for the server, agents, and agent relays; otherwise, they cannot connect to one another. Mutual authentication mode is not used for HTTPS communication.

Mutual authentication can be implemented for high-availability environments by swapping all agent and agent relay certificates with each server. All agents and agent relays have certificates from all servers, and all servers have certificates from all agents and agent relays. Once again, if you use a certification authority, you do not need to swap certificates. Each server uses the same certificate. For this purpose, load balancers are ignored.

HTTP/HTTPS server-agent communication

While most server-agent communication is done with JMS, some agent activities, such as posting logs, and moving files to CodeStation, use HTTPS. In high-availability topologies, HTTPS server-agent communication is handled by the load balancer and any server can perform validation for any request.

By default, incoming HTTP connections to the server's web interface uses port 8080. For incoming HTTPS connections, use port 8443.

Agent verification of server certificate for HTTPS communication

You can configure agents to verify the identity of the server for communication that uses the HTTPS protocol. When enabled, this option prevents agents from connecting to the server over HTTPS unless the certificate's CN matches the host name to which the agent is connecting. You can implement this feature with self-signed certificates or certificates issued by a certification authority.

When you install UrbanCode Deploy, a private key and self-signed certificate with the alias `server` are stored in the server's `tomcat.keystore` file. This certificate is presented to agents, agent relays, and users that connect to the server using HTTPS. When you enable agent verification of server certificates, the key is modified so that the agent can verify the host name

of the server. Agents are then configured to accept the trusted certificate and to also verify the host name of the server.

This option requires that the location of the Java developer kit (JDK) **`keytool`** utility is in the system path.

Agent relays can be configured to verify the identity of the server. If you do not enable this option, then only agents need to validate server certificates. See "Agent relay artifact caching."

Reference:
http://www.ibm.com/support/knowledgecenter/SS4GSP_7.0.5/com.ibm.udeploy.install.doc/topics/ssl_addl_security.html

Securing user-created properties

Secure properties are not displayed in the user interface but are instead redacted in the form of "*****". Agents are passed property values but not names. Secure properties, such as passwords, are encrypted with an Advanced Encryption Standard (AES) algorithm and a 128-bit key.

An outline of how secure properties are handled is provided in the following steps:

1. The server loads all the secure properties and decrypts them as necessary for input properties.
2. The server uses JMS to send a list of all input properties and secure values to the agent over TLS.
3. If the plug-in supports an encrypted property format, an encrypted temporary file is written to the file system. The secret for the encryption is generated on a per step basis and passed to the plug-in step through `stdin`. If the plug-in does not support an encrypted property format, a plain text file is written.
4. The agent starts the plug-in step.
5. If the plug-in supports an encrypted property format, the temporary file is read and decrypted with the secret read from `stdin`. Otherwise, the plain text file is read.
6. Before the logs are uploaded to the server, the server values found in the file are obfuscated.
7. The logs and output properties are uploaded once the step is complete. If the plug-in supports an encrypted property format, stored properties are encrypted
8. The logs, output properties, and input properties files are removed from the agent.

The secret key that is used to encrypt and decrypt secure properties is stored in the `encryption.keystore` file on the server. The secret key in the file is randomly generated during installation. Applications and components can use secure properties from other servers after the servers exchange the contents of the keystores.

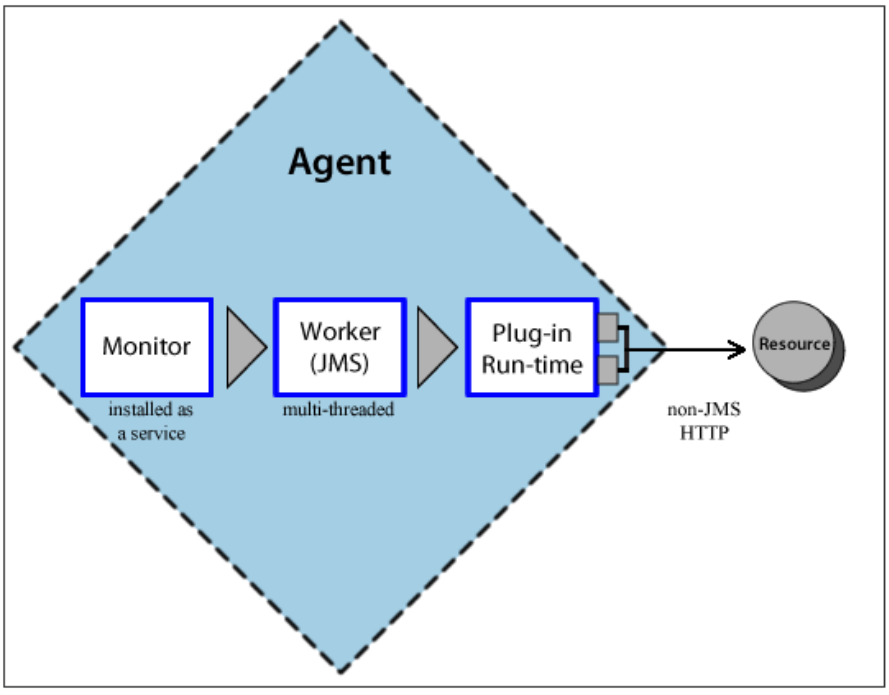
Agents

An agent is a lightweight Java process that runs on a host and communicates with the UrbanCode Deploy server. Agents can be installed on cloud environments, virtual machines (VMs), containers, or physical systems Agents perform the actual work of deployment, which relieves the server from the task. Agents are platform-agnostic.

An agent opens a socket connection to the server when it starts. JMS agent communicates between server and agents and uses Java Message Service (JMS) protocol and HTTPS. Web agent communication start as HTTP requests and upgrade to WebSocket protocol. The connection is always secured by TLS with the optional mutual key-based authentication for each endpoint. This communication protocol is resilient to network outages.

An agent consists of a worker process and a monitor process. The worker is a multithreaded process that performs the actual deployment work after receiving commands from the server. The worker process handles all communications with the server. Work commands come from plug-in steps. The monitor is a service that manages the worker process: starting and stopping, handling restarts, and upgrades. Agents can be managed from the UrbanCode Deploy web-based dashboard.

Figure 3 Agent processes



Plug-ins

Plug-ins consist of distinct parameterized processes called *steps*. Each step consists of some properties, a command that runs the step, and post-processing instructions. Post-processing typically ensures that expected results occur. Step properties can serve various purposes, from providing input to the command, to supplying the actual command itself. Plug-in properties are set at design time.

At deployment time, component processes are run by the server. Process plug-in steps are run by agents that are installed in the target environment. For a plug-in step to complete successfully, the agent must have access to all resources, tools, and files that are required by the plug-in steps in the process.

Minimum agent permissions

Typically, a user account is dedicated to the agent on the server where the agent is installed. This account is managed by the operating system where the user is installed, not by the UrbanCode Deploy team-based security system.

This user must have access to all working directories that are used in the processes that this agent runs. In addition, each agent must have these permissions:

- Open JMS and HTTPS connections. The agent uses JMS and HTTPS connections to communicate with the server's JMS and HTTPS ports or the HTTP proxy.
- Open an HTTPS connection to CodeStation. The agent must be able to connect to download artifacts from the CodeStation repository.
- Access the file system. Agents need read/write permissions to items on the file system.

Agent user impersonation

Agent accounts are managed by the operating system where the agent is installed. Unless you use impersonation, agent processes use the account that was used to start the agent. Agents can employ user impersonation when required to perform tasks for which they would not otherwise have permission. To run a database update script, for example, an agent might need to be the "oracle" user; but to update the application, the agent might need to be the "websphere" user. By using impersonation, the same agent can run the script and update the application, which enables you to combine these steps into a single process.

Impersonation on Linux/Unix

UrbanCode Deploy uses the `su/sudo` commands for user impersonation on Linux/Unix. The `su` command enables a user to start a shell as another user (process steps can be considered

individual shells). When you configure a process step, you can use impersonation for the step. By default, `su` is used but you can use `sudo` instead. To configure impersonation, you supply the username required by the target host. When the impersonation-configured process step runs, the `su` or `sudo` command runs the step as the impersonated user. You can configure default impersonation settings for agents and resources.

For sudo, impersonation privileges are defined in the `/etc/sudoers` file. To use su, agents must run as the root user. You configure the impersonating user to not need a TTY or a password.

Reference:
http://www.ibm.com/support/knowledgecenter/en/SS4GSP_7.0.5/com.ibm.udeploy.doc/topics/arch_appx_sudo_using.html

Impersonation on Windows

For agents running on Windows platforms, UrbanCode Deploy provides a program that handles impersonation. You implement impersonation for Windows-based agents the same way you do for Unix- or Linux-based agents. When you configure a process step, you specify the credentials that are used when the step is processed.

For agent impersonation purposes, an agent can impersonate a user whose username and password are stored on the target computer and who is part of the administration group.

z/OS agents

UrbanCode Deploy agents can be installed on the IBM® z/OS® platform. The z/OS deployment tools are installed on all agents that deploys z/OS components. The z/OS deployment tools use authentication tokens to communicate with the UrbanCode Deploy server.

Agent relays

Agent relays allow agents outside the firewall to communicate with the server. Agent relays simplify firewall rules and can reduce server load by reducing the number of connections. For example, if your server is on an internal network and your agents are on a public cloud, you install the agent relay on the cloud and have the agents connect to the agent relay. Then, the agent relay connects through the firewall to the internal network.

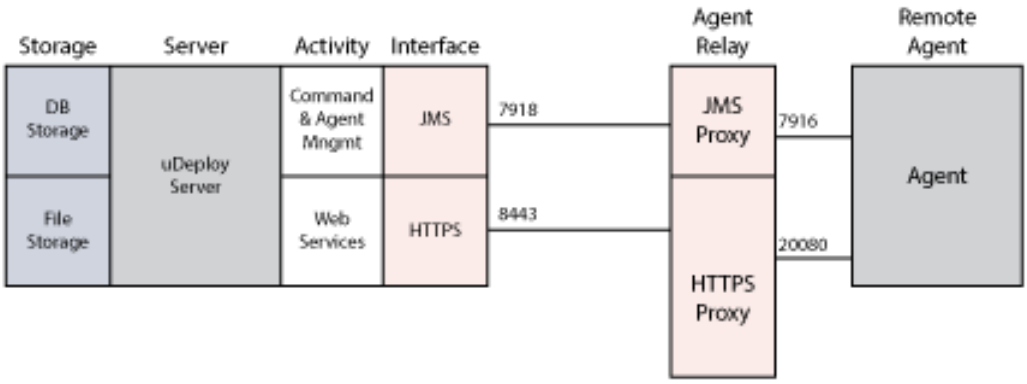
The agent relay communicates with the server and the agents by using the same combination of JMS, and HTTPS communication protocols that the agents use.

In networking terms, the agents establish long-lived TCP connections with the relay and the relay establishes long-lived TCP connections with the central server. These long-lived

connections are used for JMS communication. The server communicates back over these persistent connections to issue instructions.

In configurations with relay agents, agents local to the server can continue to communicate with the server.

Figure 4 Agent relays



Agent relays are assigned an authentication token. The token has the permissions granted to the user who created the token. Typically, a user account is created specifically for the agent relay.

The mechanics of a typical remote communication is described in the following steps:

1. A remote agent starts and establishes a connection to the agent relay using JMS, which then notifies the server using JMS that the remote agent is online.
2. The server sends, say, an artifact download command to the relay using JMS, and the relay delivers the message to the remote agent using JMS.
3. The agent requests the artifacts from the relay.
4. If the artifacts are in the relay cache, the relay begins streaming them directly to the agent over the agent-relay HTTPS connection.
5. If the artifacts are not in the relay cache, the server locates the artifacts, and uploads them to the relay over HTTPS. The relay then begins streaming the artifacts to the agent.
6. Once the remote agent completes the work, it informs the server using JMS.

Relays can be chained. If it is appropriate to communicate from network A, to network B, to network C, agent relays can point at each other to aggregate-hop networks.

Agent relay artifact caching

To aid network performance, agent relays can cache CodeStation artifacts. When you enable artifact caching, the agent relay maintains a local CodeStation repository. By default, the first

time an agent requests an artifact, the agent relay retrieves the artifact, caches it in its CodeStation repository, and provides it to the agent. You can also configure agent relays to prepopulate the cache with all artifacts with a specified status.

When using artifact caching, the user account assigned to the relay must have the `Read Artifact Set List` permission. This enables the relay to access the GUID assigned to the component version.

Agent relays and firewalls

You install the agent relay on the same network and the same side of the firewall as the agents. If your agents connect to the server through an agent relay, configure your networks and firewalls to allow the following communication:

- Agents must be able to open network connections on the agent relay JMS port. The default agent relay JMS port is 7916.
- Agents must be able to open network connections on the agent relay HTTP proxy port. The default agent relay HTTP proxy port is 20080.
- Agents must be able to open a network connection to the Agent Relay CodeStation proxy port (HTTP_proxy + 1, by default 20081).
- Agent relays must be able to connect to the server HTTPS and JMS ports. The default server ports are: 7918 for JMS, 8080 for HTTP, and 8443 for HTTPS.

Reference: http://tpm-kc15.rtp.raleigh.ibm.com:9090/kc/SS4GSP_7.0.5/com.ibm.udeploy.install.doc/topics/agent_firewalls.html

Clients

Web browsers are the most typical client, but other clients can be used to access the web services. Clients are deployed locally or remotely and communicate with the server using HTTP or HTTPS. The web-based GUI is a Rich Internet Application (RIA) that maintains much of its functionality in the browser.

Command-line client (CLI)

CLI is a command-line interface that provides access to the server. It can be used to find or set properties and to run numerous functions. The command-line client contains commands that match the REST commands for the server. CLI commands are not available for the REST commands for the blueprint design server.

REST API

UrbanCode Deploy provides a REST API that enables external systems to interact with the server. Administrators for those products authorize integrations and configure user access by issuing authorization tokens. You can also use your user name and password to access the API. Finally, you can use basic user authentication to access the REST API.

Many different programs can run REST commands. To run the command, you call a method on a REST resource and pass parameters or a request in JSON format.

Authenticating for REST commands

REST commands require the same permissions required to use the web interface. The way that you authenticate to run REST commands depends on how the server is set up and the tool that you are using to run the commands. The simplest way to authenticate for REST commands is to use basic user authentication.

Reference:
http://www.ibm.com/support/knowledgecenter/en/SS4GSP_7.0.5/com.ibm.udeploy.reference.doc/topics/rest_api_ref_authenticating.html

User security model

The team- and role-based security system manages user interactions and secures product features. Roles control virtually every product area, including the objects that users can create and who can modify the security system itself.

User-created objects are managed by teams. Team members can only access objects, such as applications, managed by their team. Team members interact with team-managed objects according to the permissions granted to their role. Some team members can create and edit environments, for example, while others might have only view access for environments. Team membership determines who receives email notifications and data on their UrbanCode Release and Deploy mobile app.

The team- and role-based security system does not affect the accounts used by agents to run processes.

An agent that runs plug-in steps with HTTP call backs to the server are given an authentication token for the user who requests the process. The agent is restricted to the permissions that requesting-user has.

The user security model does not affect the admin user. The admin user is the system's superuser and is outside the role-based model.

The blueprint design server has a separate layer of security.

Permissions and security resource types

Permissions control access to product areas and user-created resources. Most permissions refer to activities such as create, read, update and delete. *Security resource types* allow permissions to be categorized. Each security type has a set of permissions that affect how users interact with it. For example, the agent security type has permissions that affect a user's ability to interact with agents. You might create a 'production agent' type with default permissions different from those of the stand agent type.

Roles

A *role* is a set of granted permissions. A developer-type role might have permissions that enable users to create components but not create users or manage teams. A role by itself does not impart its permissions to any actual user. Like permissions, roles define what can be done, not who can do them. Roles and their associated permissions are applied to users when they are assigned to teams.

Teams

A *team* is a collection of users and groups. When a user is added to a team, that person is assigned to a role. Role membership automatically grants all permissions that are defined for the role. When groups are added to roles, all group members are automatically granted the permissions that are defined for the role.

Security resources

Security resources are user-created objects such as components and applications. Security resources are managed by teams. To secure an application, for example, a team is associated with it. User interactions with team-managed resources are determined by their role.

Authorization and authentication realms

You use authentication and authorization realms to manage user accounts. *Authentication realms* manage users; *authorization realms* manage user groups. Authentication realms manage users and determine user identity within authorization realms

UrbanCode Deploy provides the following authorization and authentication realms:

- Internal Storage uses internal role management. Users can be created manually for internal storage realms.
- Lightweight Directory Access Protocol (LDAP) including Active Directory (AD).
- Single sign-on authorization (SSO).

Most customers import users from external LDAP realms. When you import users, you can filter LDAP account data for the information you need, such as user IDs, email addresses, and passwords. Groups that the imported users belong to are also imported. Assigning a group to a team is a quick way to complete a team.

When LDAP is enabled, if an unknown user attempts to log on, LDAP authentication realms are searched, and if the user is found, a corresponding record is created. Imported or first-time users have limited read-only permissions until they are assigned to teams.

Security reports

Several security reports are available. *Security reports* provide information about user-created resources and provide information about user roles and privileges. The User Security Report provides information about the authentication realms and groups that are defined for users. Each report row represents an individual user. When selected, the report runs automatically for all users.

Appendices

Key stores and trust stores

In Java, a key store is where you store private keys and the digital certificates that servers use to authenticate themselves to remote users. A trust store is a key store that is used to store trusted digital certificates from remote parties, such as agents, users, other servers, and agent relays. Sometimes a single key store can be used for both purposes, and other times separate key stores are used.

Key stores used for JMS communication

When you install UrbanCode Deploy, a private key and self-signed certificate are stored in a key store located in the *server_installation_directory*`/appdata/conf/server.keystore file`. This certificate is presented to agents, agent relays, and users for JMS communication. You can replace the self-signed certificate with one from a CA.

When you turn on mutual authentication for JMS-based communication, the certificate in the `server.keystore` is imported into the trust stores of the agents and agent relays distributed around the system. Similarly, the certificates from the key stores of the agents and agent relays are imported into the server's trust store. The server trust store is the same as the server key store, `server.keystore`. The agent key store is located in the *`agent_installation`*`/conf/agent.keystore` file.

Key stores used for HTTPS communication

When you install UrbanCode Deploy, a private key and self-signed certificate are stored in a key store located in the *`server_installation_directory`*`/server/opt/tomcat/conf/tomcat.keystore` file. This certificate is presented to agents, agent relays, and users for HTTPS communications. You can replace the self-signed certificate with one from a CA.

If you turn on the server identity option, you modify the certificate and store a copy in the `$JAVA_HOME/lib/security/cacerts` trust store.

Sharing secured properties among servers

Applications with secured properties can be shared among UrbanCode Deploy servers. To enable application sharing, server keys are swapped among the participating servers' `encryption.keystore` key stores.

Reference:
http://www.ibm.com/support/knowledgecenter/en/SS4GSP_7.0.5/com.ibm.udeploy.install.doc/topics/ssl_mutual_authServers.html

Supported TLS protocols and ciphers

UrbanCode Deploy TLS certificates use the Java KeyStore (JKS) format. Certificates are generated by an RSA key with a 2048-bit length and are signed by a SHA256 with RSA algorithm. UrbanCode Deploy supports TLS v1, TLS v1.1, and TLS v1.2 protocols. The SSL v3 protocol was used by agents in early product versions.

Reference:
http://www.ibm.com/support/knowledgecenter/en/SS4GSP_7.0.5/com.ibm.udeploy.install.doc/topics/ssl_compatibility.html

Description of systems

The UrbanCode Deploy core architecture consists of the following elements:

UrbanCode Deploy server

The server stores components, processes, and other elements used to model deployments. You run automated deployments from the server.

Web browsers

Web browsers are the main way that users interact with the server and blueprint designer. The UrbanCode Deploy browser-based GUI is a Rich Internet Application (RIA) that maintains much of its functionality in the browser. Clients interact with REST services on the server as needed.

Command-line client

The command-line client provides access to the server through the command line. It can automate functions on the server, such as creating components and applications, and it provides most of the features that are found in the browser-based GUI. The command-line client is built on top of REST services.

REST API

The REST API provides access to the server through HTTP. Like the command-line client, it can automate functions on the server, such as creating components and applications.

The server and the blueprint design server have separate REST APIs. Each command in the server REST API has an equivalent command in the command-line client; however, commands in the blueprint design server API do not have command-line equivalents.

Agents

Agents run processes on target systems. Agents can run on physical computers, virtual systems, or cloud systems.

Agent relays

Agent relays are communication proxies for agents that are located behind firewalls or in other network locations.

Rational® Common Licensing license server

The license server provides licenses to the server.

Clouds

Clouds host virtual resources. When you create environments with blueprints, the server or engine provisions resources on the target clouds.

CodeStation

CodeStation stores component versions and artifacts. It is part of the UrbanCode Deploy server.

Blueprint design server

This server hosts the blueprint designer and controls access to blueprints, the files that describe network topology for applications that you provision to different clouds.

Heat engine

The Heat engine is an installation of the OpenStack Heat orchestration engine with IBM extensions. The engine manages cloud infrastructure, provisioning resources from clouds, updating those resources, and deleting them.

Index

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785 US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT

LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Trademarks