Linux Academy

Study Guide

Certified Jenkins Engineer

Contents

Introduction

This study guide is intended to supplement your learning experience for the course in your preparation for the Certified Jenkins Engineer (CJE) 2017 test. To truly prepare for the test, please complete the lessons, exercises and labs, and take advantage of the other supplementary materials.

Study Sheet Conventions

This study sheet follows the syllabus for the course. The content and structure is heavily influenced by the official CJE study guide distributed by CloudBees:

https://www.cloudbees.com/sites/default/files/cje-study-guide-2017.pdf

Continuous Integration/Deployment/Delivery and Jenkins itself are large expansive topics. The study sheet attempts to present these concepts in a succinct manner, while providing enough information for the student to be able to meaningfully apply the information in practice and prove adeptness in the concepts on the certification test.

About the Certification Test

- CloudBees offers two tests: the Certified Jenkins Engineer (CJE) and the Certified CloudBees Jenkins Engineer (CCJE).
- This course prepares you for the CJE; the CloudBees platform will not be covered.
- The test consists of 60 Multiple Choice questions.
- There are 4 basic sections:
    - Key CI/CD/Jenkins Concepts
    - Jenkins Usage
    - Building Continuous Delivery Pipelines
    - CD-as-code Best Practices
- For more information:
    - https://www.cloudbees.com/jenkins/jenkins-certification

Jenkins Introduction

Jenkins is the "leading open source automation server."

- https://jenkins.io/
- As is explained above on their homepage, Jenkins is an open source automation server.
- It's built in Java.
- It's basically a platform for the Software Development Life Cycle (SDLC).
- It's commonly used to implement continuous integration and continuous delivery concepts specifically.
- It's useful to anyone who works on a software project, including:
    1. Application Developers
    2. Web Developers
    3. IT Operations
    4. Systems Engineers
    5. DevOps Engineers

Jenkins is known for being easy to use and highly adaptable. There are thousands of customizations and plugins that make Jenkins suitable for most use cases. It also works in the most common OS environments and is relatively lightweight (doesn't use a lot of resources).

General Terms

Below are some commonly used abbreviations and terms that you'll notice used in the course videos and the rest of this guide:

- **CD:** CD stands for Continuous Delivery unless specified otherwise. It could also be Continuous Deployment.
- **CI:** Continuous Integration
- **DevOps:** It's really an organizational culture. The concept of developers and operations working in harmony. CI/CD is a common strategy used, but DevOps encompasses much more.
- **DSL:** Domain Specific Language
- **GUI:** Graphical User Interface
- **Job:** A task configured in Jenkins; It's synonymous with Project. "Job" is now the deprecated term.
- **JSON:** lightweight data-interchange format that uses Key/Value pairs
- **md5sum:** The digital fingerprint of a file (128-bit MD5 hashes)
- **Pipeline:** The job type created buy the Pipeline plugin; It's used more generically in CI/CD discussion.

- **Project:** A task configured in Jenkins; It's synonymous with Job. "Job" is now the deprecated term.
- **Repo:** Repository
- **REST:** Representational State Transfer, usually leverages HTTP protocol
- **SCM:** source code (or control) management
- **SDLC:** Software development life cycle
- **WebGUI:** Graphical user interface in a web browser
- **XML:** A metalanguage that allows users to define custom markup languages

Key CI / CD / Jenkins Concepts

Continuous Integration/Delivery Concepts

Continuous Integration

A software development practice where contributors are integrating their work very frequently. This results in multiple daily integrations to a mainline. Automated testing (post-commit promotion) is commonly used.

Keys and Assumptions

- People often complain it can't work in their environment. It's easier than they realize. Once adopted, people wonder how they ever lived without it.
- Assumes a high degree of testing
- Unit testing/integration testing

Basic Workflow

- Checkout from an SCM (like git)
- Branch and make local changes
- Add or change tests as necessary
- Trigger automated build locally
- If successful, consider committing
- Update with latest from mainline
- Push changes, build and test on integration machine

Best Practices

- Maintain a single source repository.

    - Have a common mainline Branch (usually master).
- Automate the Build.
    - Minimize potential for user error, automate everything possible
- Make the Build Self-testing.
    - Self-testing code
    - Use Unit Tests for granular functionality.
- Everyone commits frequently (at least daily, preferably).
    - Communication is key! Seems basic, but it's where many orgs fail.
    - Frequent merges will help avoid conflict errors.
    - The working branch should be updated as frequently as possible to help avoid very large diffs while merging.
- Build every commit.
- Prioritize fixing broken builds.
- Keep your builds fast!
- Testing environment should be as close to production as possible
- Make it easy for anyone to get the latest.
- Keep it open; everyone should see what's happening.
- Automate the deployment.

Continuous Delivery

A software development discipline where software is built so that it CAN be released to production at any time.

Team Assumptions

- Software is always deployable throughout software development life cycle (SDLC).
- Not breaking the build is prioritized over adding features.
- Feedback is fast; production readiness is known.
- Push-button deployments of any version of the software
- Close/collaborative working relationship
- Extensive automation

Difference between CI and CD

- Continuous Delivery is the ability to release at any time.
- Continuous Integration is just the practice of integrating code continuously. It doesn't necessarily mean that it can or will be released at any time.
- Not mutually exclusive, but not the same thing

Stages of CI and CD

Build -> Deploy -> Test -> Release

Continuous delivery versus continuous deployment

- Continuous Delivery means the code CAN be released at any time. When the term "Delivery" is used exclusively, the business generally isn't deploying as frequently.
- Continuous Deployment means that code IS released continuously as part of an automated pipeline. Usually associated with many deployments to production every day.

Installing / Configuring Jenkins

Installation Wizard

- Utility in WebGUI that allows for some basic configuration and the installation of common plugins
- After installing the package and starting the service via the CLI, the installation wizard becomes available from the Jenkins WebGUI.
    - http://your-server-fqdn:8080/
- Secret password in `/var/lib/jenkins/secrets/initialAdminPassword`
- Can add an admin user and set credentials

Prerequisite Install

1. Go to: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
2. Download the appropriate Java jdk version from the Oracle website. For this course we use jdk-8u121.
3. Copy the package from your local environment to the target server.
    - Below is an example using `scp`:

```
scp jdk-8u121-linux-x64.rpm user@your-server:/home/user/
```

4. Install the jdk package.

```
rpm -Uvh jdk-8u121-linux-x64.rpm
```

5. Set up Alternatives for Java:

```
alternatives --install /usr/bin/java java /usr/java/latest/bin/java
200000
alternatives --install /usr/bin/javac javac /usr/java/latest/bin/javac
200000
alternatives --install /usr/bin/jar jar /usr/java/latest/bin/jar 200000
```

- Note: Check out Terry's nugget on this for more detail: "Setting local and global java environment variables"

6. Set JAVA_HOME environmental variable in rc.local.

```
vi /etc/rc.local
```

```
export JAVA_HOME="/usr/java/latest"
```

Jenkins Install

1. Add the Jenkins repo to your yum sources on the CentOS node.

```
wget -O /etc/yum.repos.d/jenkins.repo https://pkg.jenkins.io/redhat-
stable/jenkins.repo
```

2. Import the Jenkins rpm signing key.

```
rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io.key
```

3. Install the Jenkins package.

- We will be using version 2.19.4 which is the target version for the certification test.

```
yum install -y jenkins-2.19.4-1.1
```

4. Check for services running on 8080 before starting the Jenkins service.

```
netstat -tulpn | grep 8080
```

5. If nothing is running on 8080, go ahead and start the service via systemctl.

```
systemctl start jenkins
```

6. Also, enable the Jenkins service so it starts on system startup.

```
systemctl enable jenkins
```

7. Check again for services running on port 8080.

    - There will be a slight delay, so we'll use watch to wait for the signal.

```
watch n=1 "netstat -tulpn | grep 8080"
```

    - ctrl-c to break the watch when the service is shown to be running on 8080

8. Visit the WebGUI at: `http://your-server-fqdn:8080/`

9. You'll be prompted for the initialAdminPassword which is located in `/var/lib/jenkins/secrets/initialAdminPassword` on the system being configured. You'll want to `cat` that and copy and paste it into the browser.

```
cat /var/lib/jenkins/secrets/initialAdminPassword
```

10. Paste the password into Install Wizard.

11. Choose "Install Suggested Plugins."

Note:

- I've seen this fail before.
- If you see several plugins that fail to install, and the install hangs at the "Getting Started" page, try the following:

    1. From the terminal on the Jenkins master:

    ```
    systemctl stop jenkins
    systemctl start jenkins
    ```

    2. Then, go back to the WebGUI and start again.

12. Set admin settings, user, password, etc.

13. Press "Enter."

14. Click "Start Using Jenkins."

15. Now you have your Jenkins Master up and running!

Configuring Jenkins

Plugins

Plugins add additional functionality to Jenkins.

- Can be used to add functionality in many ways
  - "Credentials" manages user Credentials.
  - "Git" and "GitHub" manage Git integrations.
  - "JUnit" handles unit testing reports.
- There are thousands of plugins.
- Plugins (.hpi files) of various releases can be downloaded from:
  - http://updates.jenkins-ci.org/download/plugins

Plugin Manager

The UI to install and configure plugins in the Jenkins WebGUI

Procedure for Installing Plugins

1. From the Dashboard, click "Manage Jenkins."
2. Click "Manage Plugins."
3. Click the "Available" tab.
4. Filter plugins for the targeted plugin with the search box.
5. Click the "Install" button for the desired plugin.

Procedure for Removing Plugins

1. From the Dashboard, click "Manage Jenkins."
2. Click "Manage Plugins."
3. Click the "Installed" tab.
4. Click the "Uninstall" button for the desired plugin.

Procedure for Downgrading Plugins

1. From the Dashboard, click "Manage Jenkins."
2. Click "Manage Plugins."
3. Click the "Installed" tab.
4. Click the "Downgrade to..." button for the desired plugin.

Procedure for Upgrading Plugins

1. From the Dashboard, click "Manage Jenkins."
2. Click "Manage Plugins."
3. Click the "Updates" tab.
4. Check the box next to the desired update.
5. Click "Download now and install after restart."
6. Restart the service jenkins service from the Jenkins master cli, or check the box that will do that for you.

(Advanced) Installing a Plugin from a .hpi File

1. From the Dashboard, click "Manage Jenkins."
2. Click "Manage Plugins."
3. Click the "Advanced" tab.
4. In the "Upload Plugin" section, use the "Choose File" button to navigate to your hpi file.
5. Install it

The Dashboard

Left Panel

- **New Item**
    - This is where you'll add projects, folders and pipelines.
    - The core of Jenkins functionality is here.
- **People**
    - This is where you you can see a list of users and their latest activities.

- **Build History**
    - You'll see an overview display of build history for all projects in graphical form.
- **Manage Jenkins**
    - Where Jenkins is managed
- **My Views**
    - For configuring custom views for projects for the logged in user.
- **Credentials**
    - Lists credentials that have been configured for Jenkins

Build Queue

Jobs waiting for an executor are listed here.

Build Executor Status

- The status of projects associated with Jenkins executors
- An executor runs projects dictated by Jenkins.
- They can run in parallel.
- The default number of executors on the master is 2.

Build History

- Shows the history of builds
- Stable is blue
- Red is broken

Security

Authentication Versus Authorization

Authentication identifies a user, while Authorization dictates what a user is allowed to do.

Security Terms

- **Auditing:** Tracking who did what on your Jenkins server
- **Credentials:** Username/Password for logging into Jenkins

Matrix Security

- Allows you to grant specific permissions to users and groups.
- Most Common Permissions: Overall, Slave, Job, Run, View, and SCM.

Configuration Procedure

1. Go to the Jenkins Dashboard.

   ```
   http://<your-server>:8080
   ```

2. Click "Manage Jenkins."
3. Click "Configure Global Security."
4. Check "Enable Security."
5. Select "Jenkins' own user database."
6. Place a check mark next to "Allow users to sign up."
7. Select "Matrix-based Security."
8. Select the admin user and check all the permissions.
9. Click Save.
10. (Optional) Set "Read" permissions for "Anonymous."
11. (Optional) Set options appropriate for other users.

Roles/Permissions

Categories and Associated Permissions

Overall

- **Administer:**
  - Make system-wide configuration changes.
  - Perform highly sensitive operations that amounts to full local system access (within the scope granted by the underlying OS)
- **Read:**
  - View almost all pages within Jenkins

- **RunScripts:**
    - Run groovy scripts via the groovy console or groovy cli command
- **UploadPlugins:**
    - Upload arbitrary plugins
- **ConfigureUpdateCenter:**
    - Configure update sites and proxy settings

Slave

- **Configure:**
    - Configure existing slaves
- **Delete:**
    - Delete existing slaves
- **Create:**
    - Create new slaves
- **Disconnect:**
    - Disconnect slaves or mark slaves as temporarily offline
- **Connect:**
    - Connect slaves or mark slaves as online

Job

- **Create:**
    - Create a new job
- **Delete:**
    - Delete an existing job
- **Configure:**
    - Update the configuration of an existing job
- **Read:**
    - Grants read-only access to project configurations
- **Discover:**
    - Redirect anonymous users to a login form rather than presenting an error message if they don't

have permission to view jobs

- **Build:**
    - Start a new build and cancel a running build
- **Workspace:**
    - Retrieve the contents of a workspace that Jenkins has checked out for performing a build
- **Cancel:**
    - Cancel a running build

Run

- **Delete:**
    - Delete specific builds from a build's history
- **Update:**
    - Update the description and other properties of a build
    - For example, to leave notes about the cause of a build failure

View

- **Create:**
    - Create new views
- **Delete:**
    - Delete existing views
- **Configure:**
    - Update the configuration of existing views
- **Read:**
    - See any existing views

SCM

- **Tag:** Create a new tag in the source code repository for a given build

Jobs/Projects

- Any executable task in Jenkins
- The term "job" is now deprecated in favor of "project."

Organizing Projects

Folders

- Creates a container with nested items
- It creates a separate namespace for the nested projects.
- Projects can be organized in folders.

Navigation

Left Panel -> New Item -> Folder

Adding Items to Folders

- 3 Options:
  - Hover and click arrow on Existing Item; choose "move"; select target folder.
  - Click existing item; In the item view page, select "Move" in the left panel; select target folder.
  - For new items - Click folder; in the folder view, select new item; it will automatically be added to that folder.

Configuration

- Health Metrics - "Child item with worst health"
  - The item in the folder with the worst "health" dictates the health of the folder.
  - If you set recursive, item's in nested sub-folders are used to calculate this health.

Views

A view is simply a view of items configured in Jenkins.

- The Dashboard has a default view, "All"
- Can be configured for a user (using My Views) or system-wide
- Add a new view by clicking the "+" next to the tabs in the table on the homepage or on the "My Views" page.

Navigation

- System-wide views can be accessed with tabs on the Dashboard home page.
- User views, can be accessed with the "My Views" button in the left panel of the home page.

My View

- Shows all projects that the user has access to

List View

- Shows projects in a simple list format.
- You can arbitrarily filter which jobs show in this view.

Configuration

- Filter Build Queue/Executors
  - You can filter the projects that show in the Build Queue/Executor panels based on the filters set in this view.
- Status filter
  - Filter based on the current status of the job
  - Recursive in sub-folders
    - Allows you to list jobs that are nested in folders
- Jobs
  - Select jobs to show in the view explicitly
- Use a Regular Expression to include jobs into the view
  - Match Project names based on a regular expression
  - Example:

```
.\*Project.\*
```

Parameterized Projects

- These projects accept user parameters. Parameters can be added during job configuration.
- When triggering a build, one can select "Build with Parameters" and specify parameter values.
- Parameters are exposed as environment variables to scripts, and can also be accessed from Project configuration.
- Invocation Example:

```
"${branch}"
```

Configuration

- From Project Configuration, select "This project is parameterized."

Parameter Types

- **String Parameter:** The most common, generic parameter; simply a string
- **File Parameter:** A file as a parameter; it should be in the workspace, as part of a source code management checkout
- **Boolean Parameter:** A true/false value
- **Choice Parameter:** dropdown list, first line is the default value
- **Credentials Parameter:** Credential as a parameter; UUID is exposed
- **List Subversion tags:** Tags in a subversion repo
- **Multi-line String Parameter:** Like a string parameter, but multi-line
- **Password Parameter:** A password that can be set as a string value; password is protected when input in "Build with parameters" view. Otherwise, it's accessed like a string parameter. The value is not protected in the console view.
- **Run Parameter:**
- Access information from a specified build of a project
- To use a Run Parameter, the value should be in the format:

```
jobname#buildNumber (eg. "&MyRunParam=foo-job%2399" for foo-job #99)
```

Types of Projects

Freestyle Project

- This is the most versatile type of project.
- You can use any SCM with any build system.
- It can even be leveraged for non-build tasks.

Pipeline

- Formerly known as workflow
- Long running activity orchestration
- Can span multiple slaves
- Jenkins pipeline is a suite of plugins which supports CI/CD pipelines into Jenkins.
- "Pipeline as code" is illustrated by the Jenkinsfile using the Jenkins DSL.

External Job

Allows for management of job execution in another automation system

Multi-Configuration Project

- For projects that have a lot of different configurations
- It can be best if you need to test in multiple different types of environments.
- This is likely more suitable for projects that are comprised of many build steps.

Folder

- Creates a container with nested items
- It creates a separate namespace for the nested projects.
- Projects can be organized in folders.

GitHub Organization

Scans a GitHub organization for repositories and branches

Multibranch Pipeline

Detects branches in an SCM repository

Upstream Versus Downstream Projects

- A project that is triggered by another project is considered to be downstream of that project.
- A project that triggers another project is considered to be upstream of that project.

Builds

A build can describe what is produced from a project, or a single run of a project.

Build Terms

- **Artifact:** Something produced from a build
- **Build Step:** A single action in a build
- **Repository:** The home of the sources code
- **Slave:** An agent or node that runs builds
- **Trigger:** A mechanism that starts a build

Build Steps

- Build steps are added as part of the project configuration.

- They can be specified and configured from a Jenkinsfile in pipeline configurations.

Examples of Build Steps:

- Execute shell - can be a groovy script
- Execute a command in a Docker container
- Execute an Ant/Maven Build
- Running Tests
- Deploying

Jenkins Build Environment Variables

- Environment variables available to all builds
- Accessible via this syntax:

```
# String Interpolation
"${VARIABLE}"
#OR
$VARIABLE
```

- **BUILD_NUMBER:** Just the Jenkins build number; might be useful to interpolate into an artifact name.
- **NODE_NAME:** The node name as defined in the "Manage Nodes" section in "Manage Jenkins." This might be useful when sending alerts, and could help with problem node identification.
- **JOB_NAME:** Name of the job
- **EXECUTOR_NUMBER:** Number of the executor on the node
- **WORKSPACE:** The fully-qualified path to the workspace; can be useful in some scripts

Build Triggers

Pushing versus Pulling

- When Jenkins reaches out to the source to determine what has happened, that's a "pull" scenario.
- When the source notifies Jenkins (like with a Githook), that's a "push" scenario.
- Pulling is inherently passive. There is the potential for latency at the rate of the polling interval.
- Pushing is generally faster and preferred.

SCM Polling

- Jenkins will poll your SCM (like Git or Subversion) on an interval.

- Described as a "pull" mechanism as opposed to "push."
- If there is a change for the specified ref(s) or branch(es), the project will run.
- If using the Jenkins Git plugin with GitHub, you can select this field and leave it empty; the Githook will trigger builds on changes. This is actually using the SCM Polling setting for a "push" scenario.
- The interval setting uses the same cron format as described in the "Time Interval" trigger below.

Build Periodically

- Builds on a pre-defined interval
- Schedule Config Item
- Same cron format is used for "SCM Polling"
- "H" stands for hash.
- The H symbol can be used with a range. For example,

```
H H(0-7) * * *
```

- ...means some time between 12:00 AM (midnight) to 7:59 AM.
- The "H" symbol can be thought of as a random value over a range, but it actually is a hash of the job name, not a random function, so that the value remains stable for any given project.
- Uses the "cron" date format

```
* * * * *
| | | | |
| | | | +---- Day of the Week   (range: 1-7, 1 standing for Monday)
| | | +------ Month of the Year (range: 1-12)
| | +-------- Day of the Month  (range: 1-31)
| +---------- Hour              (range: 0-23)
+------------ Minute            (range: 0-59)
```

Examples:

```
# every fifteen minutes (perhaps at :07, :22, :37, :52)
H/15 * * * *
```

```
# every ten minutes in the first half of every hour (three times,
perhaps at :04, :14, :24)
H(0-29)/10 * * * *
```

```
# once every two hours at 45 minutes past the hour starting at 9:45 AM
and finishing at 3:45 PM every weekday.
```

```
45 9-16/2 * * 1-5


# once in every two hours slot between 9 AM and 5 PM every weekday
(perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)
H H(9-16)/2 * * 1-5


# once a day on the 1st and 15th of every month except December
H H 1,15 1-11 *


# Runs every year
@yearly


# OR
@annually


# Runs every month
@monthly


# Runs every week
@weekly


# Runs every day
@daily


# Runs everyday some time between 12:00am to 12:59am
@midnight


# Runs every hour
@hourly
```

Triggered by Another Project

- This can be set to watch for the completion of another project. (A "pull" scenario)
- Triggers can also be "pushed" by adding a build step "Trigger/call builds on other projects."
- With the "Parameterized Trigger Plugin", you can trigger, and pass parameters to, another project as a "Post-build" step.

Hooks

- Hooks can be configured to trigger jobs via Jenkins' REST API.
- Hooks provide a "push" scenario. The source informs Jenkins when to build.

- Git hooks can be configured in the more traditional way as well, by adding scripts to the ".git/hooks" directory in the repo path.
- Both the GitHub and Git plugins provide a hook inherently.
- Prerequisites for the procedures below:
    - Have SSH keys configured for Jenkins and GitHub
    - Your Jenkins master must be publicly routable, reachable from the central GitHub servers.
    - The Project must be previously configured to monitor the GitHub project already in the "Source Code Management" section.

Procedure for Configuring the GitHub Plugin Git Hook

1. Log in to GitHub.

2. Within GitHub, Click your project.

3. Click "Settings" from the project view.

4. Click "Integrations and Services."

5. Find "Jenkins (GitHub plugin)", via search.

6. Set the following as the "Jenkins hook url":

```
http://<your-server-hostname>.<your-domain>:<port>/github-webhook/
# e.g. http://brandon4231.mylabserver.com:8080/github-webhook/
```

7. On your Jenkins Master Dashboard, log in to Jenkins with Administrator User.

8. Click your project.

9. Click 'Configure' from the left panel.

10. Under build triggers, check "GitHub hook trigger for GITScm polling."

11. Click "Save."

12. (Optional) Make a commit to the repo and wait for the build to trigger to test.

13. (Optional) On GitHub: From the "Jenkins (GitHub plugin)" view in the "Integrations and Services" section, there is an option to test. Click "Test Service" from there and it will attempt to trigger a build from there without committing to the repo.

Procedure for Configuring the Git Plugin Git Hook

1. Log in to GitHub.

2. On GitHub, click your project.

3. Click "Settings" from the project view.

4. Click "Integrations and Services."

5. Find "Jenkins (Git plugin)", via search.

6. Add your Jenkins master url, including port (probably 8080):

```
http://<your-server-hostname>.<your-domain>:<port>/
# e.g. http://brandon4231.mylabserver.com:8080/
```

7. On your Jenkins Master Dashboard, log in to Jenkins with Administrator User.

8. Click your project.

9. Click 'Configure' from the left panel.

10. Under build triggers, check "Poll SCM."

11. Leave the "Schedule" empty.

12. Click "Save."

13. (Optional) Make a commit to the repo and wait for the build to trigger to test.

Artifacts and Fingerprints

Artifacts

Items produced from a build.

- Can download from WebGUI
- Add a post-build step to generate a artifact for a project.

Adding an Artifact to a Freestyle Project

1. In the "Project Configuration" view, go to the "Post-build Actions" section.

2. Click the "Add post-build action" dropdown and select "Archive the artifacts."

3. In the Files to archive field, insert a pattern or path to the file(s) you wish to archive.

Copying Artifacts

Can leverage the "Copy Artifact" Plugin to move artifacts between projects/jobs

Procedure for Copying Artifacts

1. Install the "Copy Artifact" plugin.
2. In the "Project Configuration" view, go to the "Build" section.
3. Click the "Add build step" dropdown, and select "Copy artifacts from another project."
4. Insert the "Project name" of the project you'd like to copy the artifact from.
5. Select the appropriate option for "Which build."
    - "Latest successful build" is the default.
6. Select the "Artifacts to copy" or "Artifacts not to copy."
7. Specify the "Target directory" from within the workspace for the artifacts to land.

Setting Artifact Retention Policy

1. In the "Project Configuration" view, go to the "General" section.
2. Check "Discard old builds."
3. Click "Advanced."
4. Specify "Days to keep artifacts."
5. Specify "Max # of builds to keep with artifacts."

Fingerprints

Tracks md5sum of artifacts that are used between projects.

Viewing Fingerprints of Artifacts

1. In the "Project Build" view, click "See Fingerprints."
2. From there you can see the Artifacts and their original owning projects.
3. If you click "More Details," you can see everywhere where that specific file has been used.
4. You can also see the md5 hash for the file.

Distributed Builds

- Running builds on one or more slaves instead of a master-only configuration.
- The master is the controller, while the slaves run projects.
- Distributing builds is a way to lower the resource load on the master.
- As the scope of a Jenkins configuration expands with more and more projects, it may make sense to incrementally add build slaves.

About Build Slaves

- Synonymous with Build Agent
- Triggered by ssh agent by default
- SSH, JNLP and Cloud are the different agent types.
- Once set up, the distribution of tasks between nodes is fairly automatic.
- Adding slaves doesn't really change how you interact with Jenkins much at all.
- The master will still serve all administrative tasks, http requests.

Executors

Executors dictate how many builds can run on a node at a given time.

- Executors can still exist on the master after you configure build slaves.
- "# of executors" can be set by selecting the configuration icon (gear) in the "Manage Nodes" list view for the node, and then setting the number in the corresponding field.

Assigning Nodes to Projects

Specific build slaves can be selected for projects with expression matching in the "Project Configuration" view, or in the Jenkinsfile for Pipeline applications.

Labels

- Expressions can match on the node's name, which is defined in "Manage Nodes" or by label.
- Labels can be added by selecting the configuration icon (gear) in the "Manage Nodes" list view for the node, and then adding a space separated list of labels to the "Labels" field.
- Examples:

```
CentOS JDK linux
```

Monitoring Build Slaves

- Agents can be monitored from the "Manage Nodes" view.
- A disconnected node will show a small red 'x' on it.
- You can also view the following status items:
    - Architecture
    - Clock Difference ("In sync" is desirable)
    - Free disk space
    - Free swap space
    - Free temp space
    - Response time (Generally, the lower the better)

Procedure for Adding a Jenkins Slave via SSH

- Prerequisites
    - Spin up the node (cloud instance, or hardware).
    - Install and configure Java as described in the Install Pre-requisite section.
    - Generate an RSA key pair for the Jenkins user on the Jenkins Master.
- From the target slave node's console

1. Switch to the "root" user.

```
sudo su
```

2. Add a *jenkins* user with the home `/var/lib/jenkins`.

```
useradd -d /var/lib/jenkins jenkins
```

- From the Jenkins master

3. Copy the id_rsa.pub key from the Jenkins user on the master.

```
cat /var/lib/jenkins/.ssh/id_rsa.pub
```

- From the target slave node's console

4. Create an authorized_keys file for the Jenkins user.

```
mkdir /var/lib/jenkins/.ssh
vi /var/lib/jenkins/.ssh/authorized_keys
```

5. Paste the key in the file vim. Save with ":wq".

- From the Jenkins Dashboard

6. Click "Manage Jenkins" from the left panel.
7. Click "Manage Nodes."
8. Click "Add Node."
9. Set a name for your node (e.g. "Slave 1").
10. Select "Permanent Node."
11. Set "Remote root directory" to `/var/lib/jenkins`.
12. Set "Usage" to "Use this node as much as possible."
13. Set "Launch Method" to "Launch slave agents via SSH."
14. Set "Host" to your node's FQDN (e.g. brandon4232.mylabserver.com).
15. Select "Add" under "Credentials."
16. Set "Kind" to "SSH Username with private key."
17. Set "Username" to "jenkins."
18. Set "Private key" to "From the Jenkins Master."
19. Click "Add."
20. Choose the new credential from the "Credentials" dropdown.
21. Click "Save."
22. The agent should now be available for use.

Build Tools Configuration

Often times, plugins or tools have specific configuration.

Navigation

Dashboard -> Manage Jenkins -> Global Tool Configuration

OR

Dashboard -> Manage Jenkins -> Configure System

Configurable Build Tool Examples

- Ant

- Docker
- Git
- Gradle
- JDK
- Maven

Source Code (Control) Management

Software that manages the interaction and code contributions of a software development team for a project

- There's often a centralized "Origin" server where the SCM lives.
- Developers will also have copies of the repository in their local development environment.
- Tags are often used for releases. Tag names can be derived from data made available from the Git or GitHub plugins.

Common SCM Software

- Git
- Subversion

Cloud-Based SCMs

- GitHub
- AWS CodeCommit
- Atlassian BitBucket

SCM Polling

See the "Build Triggers" section above.

Installing Git

You'll want to install Git on all nodes that will interface with your Git Repository.

1. Become the root user.

```
sudo su
```

2. Install the git client RPM.

```
yum install -y git
```

The Git Plugin

The Git plugin allows the integration of Git functionality within a project.

Environment Variables

The Git plugin sets several environment variables that are accessible from build steps and configuration:

- **GIT_AUTHOR_EMAIL and GIT_COMMITTER_EMAIL:** The email entered if the "Custom user name/e-mail address" behaviour is enabled; falls back to the value entered in the Jenkins system config under "Global Config user.email Value" (if any)
- **GIT_AUTHOR_NAME and GIT_COMMITTER_NAME:** The name entered if the "Custom user name/e-mail address" behaviour is enabled; falls back to the value entered in the Jenkins system config under "Global Config user.name Value" (if any)
- **GIT_BRANCH:** Name of the remote repository (defaults to origin), followed by name of the branch currently being used, e.g. "origin/master" or "origin/foo"
- **GIT_COMMIT:** SHA of the current revision
- **GIT_LOCAL_BRANCH:** Name of the branch on Jenkins. When the "checkout to specific local branch" behavior is configured, the variable is published. If the behavior is configured as null or **, the property will contain the resulting local branch name sans the remote name.
- **GIT_PREVIOUS_COMMIT:** SHA of the previous built commit from the same branch (the current SHA on first build in branch)
- **GIT_PREVIOUS_SUCCESSFUL_COMMIT:** SHA of the previous successfully built commit from the same branch.
- **GIT_URL_N:** Repository remote URLs when there are more than 1 remotes, e.g. GIT_URL_1, GIT_URL_2
- **GIT_URL:** Repository remote URL
- https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin

Jenkins Changelogs

Shows a list of commits for the target repository and branch for the build since the previous build

From Jenkins Dashboard

Project -> Build -> Changes

Incremental Updates Versus Clean Check Out

- An incremental change is usually much faster as the initial environment is already set up and available.
- However, you can't be fully assured that the environment is clean, or has remnants of the previous build job.
- A clean check out ensures that the environment is untouched.

Checking In Code

- Users should check in daily to meet CI best practices.
- The developer should also check for functionality in the local development environment against the most recent mainline code set before checking in changes.

Infrastructure As Code

- Infrastructure is completely defined in code.
- The state of all systems should be defined in code as opposed to local configuration files on the systems themselves.
- Modern config management tools like Puppet, Chef, and Ansible make this possible, along with automated CI/CD tools like Jenkins.

Branch and Merge Strategies

- Developer Branch
- Feature/Fix Branch
- Release Branch
- Merge based on release Scope
- Tags are often used for releases.

Testing

- Automated testing with Jenkins is fast.
- Shortening the feedback loop makes developers more productive.

Unit Tests

- Tests a small part of the code set
- Usually associated with an individual class method if applicable

- Unit test failures can be set to trigger a Build Breakage or not.

JUnit Example

Ensures that the "Rectangle" instance function "getArea" returns '6'

```
// A JUnit Test
Rectangle myRectangle = new Rectangle(2,3);
...
@Test
public void testGetArea() {
assertEquals(myRectangle.getArea(), 6);
}
```

Reporting

- You can publish reports for many unit testing frameworks, JUnit for example.
- The Ant build tool has a function to generate JUnit result xml files.

Health Report Amplification factor

- "Health report amplification factor" is a factor that multiplies the percent of failures.
- Examples:
  - Setting to "2" when the actual failure percentage is 1% results in a 98% health score.
  - Setting to "10" when the actual failure percentage is 3% results in a 70% health score.

Procedure for Adding JUnit Reports to Projects

Freestyle (Traditional) Reports

From the Project Configuration Page

1. Click the "Add post-build action" dropdown.

2. Select "Publish JUnit test result report."

3. Set the relative path from the workspace to the result xml file.

Example from the course exercises:

```
//path
reports/result.xml
```

Ant Buid File Configuration Snippet to generate the test report:

```
<target name="test" depends="clean, makedir, compile">
  <junit printsummary="yes" fork="true">
    <classpath refid="class.path" />
    <test name="RectangleTest" todir="${report.dir}" outfile="result">
      <formatter type="xml" />
    </test>
  </junit>
</target>
```

4. Set a "Health report amplification factor" (described above).

5. Check "Allow empty results" if desired.

6. Click "Save."

Pipeline Reports

In the Jenkinsfile (declarative):

```
steps {
  sh 'ant -f test.xml -v'
  // here's where you put the path to the result file
  junit 'reports/result.xml'
}
```

Testing Code Coverage

Code coverage is a measure of the degree of testing on your code set.

Cobertura

- There's a code coverage plugin for Java, called Cobertura, that is commonly used.
- In Cobertura, you can specify percentages for Sunny/Stormy/Unstable Coverage Metric Target statuses.
- Must have JUnit test report to run on
- Procedure for Use

  1. Install the Cobertura plugin (via Manage Jenkins -> Manage Plugins) (See the "Plugin" Section for more detail)

  2. Configure your project's build script to generate Cobertura XML reports.

  3. Select "Publish Cobertura Coverage Report" from the "Add post-build action" dropdown in your project's configuration view.

  4. Enable the "Publish Cobertura Coverage Report" publisher.

5. Specify the path where the Cobertura report is generated.
6. Click Advanced; Configure the "Coverage Metric Targets" to reflect your goals.
    - Metric targets can be set for Sunny/Stormy/Unstable for the following metrics:
        - Methods
        - Lines
        - Conditionals
        - Packages
        - Files
        - Classes

Verification/Functional/Smoke Test

- Also known as sanity testing, or Build Verification
- It's a set of testing that verifies that the software's primary functionality still works.

Integration Testing

- Tests to make sure everything still works together.
- Common in environments with multiple modules or components, and/or multiple contributing teams.
- Systems Test

Acceptance Test

- The software is tested for usage scenarios or meeting the requirements specified for the project.
- These tests typically run later than unit tests.

Notifications

- Important for reenforcing CI/CD ideals
- Prioritizing fixing the build when it fails is a central tenant of CI.
- Ensure contributors are aware of problems in a fast, distributed and automated fashion.
- Configured in the post build action section of configuration of a project

Alarming

Jenkins notifications can be enhanced with post-build actions in project configuration.

Failure Notifications

- If the build breaks, there should be a notification.
- At that point, the failure is expected without action, so Jenkins will not continue to send notifications.
- This is important to prevent "Alert Fatigue."

Integrations

- Email
    - Set a static recipient for alerts.
- Editable Email Notification
    - Email alerts with more options for notification content
- Slack
- HipChat
- Jabber
- IRC

Pipelines

- Formerly known as workflow
- Long running activity orchestration
- Can span multiple slaves
- Jenkins pipeline is a suite of plugins which supports CI/CD pipelines into Jenkins.
- "Pipeline as code" is illustrated by the Jenkinsfile using the Jenkins DSL.

Setting Up Docker

Run this on all nodes where you plan to leverage docker (probably the master and agents).

1. Switch to the root user.

```
sudo su
```

2. Ensure there are no docker packages on the system already.

```
yum remove docker docker-common container-selinux
```

3. Ensure "yum-utils" is installed.

```
yum install -y yum-utils
```

4. Add the docker repo.

```
yum-config-manager --add-repo https://download.docker.com/linux/centos/
docker-ce.repo
```

5. Clear the Yum Cache.

```
yum clean all
```

6. Install the docker-ce package.

```
yum install docker-ce-17.03.0.ce-1.el7.centos
```

7. Add the jenkins user to the docker group.

```
gpasswd -a jenkins docker
```

8. Now, start the docker service.

```
systemctl start docker
```

9. Enable the docker service so it starts at system start time.

```
systemctl enable docker
```

10. Refresh the Jenkins service.

```
systemctl restart jenkins
```

Ant

- A command line tool for compiling, building and testing java projects

- Unlike Maven, it doesn't require project layout conventions.

Ant Installation

- You'll want to install Ant on all nodes that may build via an Ant file.
- If you're following a long with the course, you need to install Ant on the master and the slave node.

1. Switch to the root user.

```
sudo su
```

2. Download Ant.

```
wget http://www.us.apache.org/dist/ant/binaries/apache-ant-1.10.1-bin.
tar.gz
```

3. Unpack the tar file.

```
tar xvfvz apache-ant-1.10.1-bin.tar.gz -C /opt
```

4. Symlink `/opt/apache-ant-1.10.1` to `/opt/ant`

```
ln -s /opt/apache-ant-1.10.1/ /opt/ant
```

5. Add the ANT_HOME environment variable.

```
sh -c 'echo ANT_HOME=/opt/ant >> /etc/environment'
```

6. Symlink `/opt/ant/bin/ant` to `/usr/bin/ant`

```
ln -s /opt/ant/bin/ant /usr/bin/ant
```

Ant Build Files

- Ant build files typically live in the root of your repository.
- These builds would be executed with the following syntax:

```
ant -f <file>.xml -v
```

build.xml

```
<?xml version="1.0"?>
<project name="Rectangle-Build" default="main" basedir=".">
```

```
  <property environment="env"/>
  <!-- Sets variables which can later be used. -->
  <!-- The value of a property is accessed via ${} -->
  <property name="src.dir" location="src" />
  <property name="build.dir" location="bin" />
  <property name="dist.dir" location="dist" />
  <path id="class.path">
    <pathelement location="./lib/junit-4.10.jar" />
    <pathelement location="./bin" />
  </path>
  <!-- Deletes the existing build, docs and dist directory-->
  <target name="clean">
    <delete dir="${build.dir}" />
    <delete dir="${dist.dir}" />
  </target>
  <!-- Creates the  build, docs and dist directory-->
  <target name="makedir">
    <mkdir dir="${build.dir}" />
    <mkdir dir="${dist.dir}" />
  </target>
  <!-- Compiles the java code (including the usage of library for JUnit
-->
  <target name="compile" depends="clean, makedir">
    <javac srcdir="${src.dir}" destdir="${build.dir}">
      <classpath refid="class.path" />
    </javac>
  </target>
  <!--Creates the deployable jar file  -->
  <target name="jar" depends="compile">
    <jar destfile="${dist.dir}\rectangle.jar" basedir="${build.dir}">
      <manifest>
        <attribute name="Main-Class" value="Rectangulator" />
      </manifest>
    </jar>
  </target>
  <target name="main" depends="compile, jar">
    <description>Main target</description>
  </target>
</project>
```

test.xml

```
<?xml version="1.0"?>
<project name="Rectangle-Test" default="main" basedir=".">
  <property environment="env"/>
  <property name="src.dir" location="src" />
  <property name="build.dir" location="bin" />
  <property name="report.dir" location="reports" />
  <path id="class.path">
    <pathelement location="./lib/junit-4.10.jar" />
    <pathelement location="${build.dir}" />
  </path>
  <target name="clean">
    <delete dir="${build.dir}"  />
    <delete dir="${report.dir}" />
  </target>
  <target name="makedir" depends="clean">
    <mkdir dir="${build.dir}" />
    <mkdir dir="${report.dir}" />
  </target>
  <target name="compile" depends="makedir">
    <javac srcdir="${src.dir}" destdir="${build.dir}">
      <classpath refid="class.path" />
    </javac>
  </target>
  <target name="test" depends="clean, makedir, compile">
    <junit printsummary="yes" fork="true">
      <classpath refid="class.path" />
      <test name="RectangleTest" todir="${report.dir}" outfile="result">
        <formatter type="xml" />
      </test>
    </junit>
  </target>
  <target name="main" depends="test">
    <description>Main target</description>
  </target>
</project>
```

The Jenkinsfile

- Defines your continuous delivery pipeline
- Lives with your source code
- 2 Styles: Declarative and Scripted
- Written in Groovy
- Use Groovy plugins/extensions for text editors.
- For Atom: https://atom.io/packages/language-groovy
- Pay attention to your curly braces!

Basic Structure (Declarative)

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

The Pipeline Directive

- It's required for the declarative format of the Jenkinsfile.
- It just says, "here's my pipeline definition."

```
pipeline {
...
}
```

The Agent Directive

- Where you specify which agent to run the pipeline, or a stage in the pipeline
- Can specify multiple agents for a pipeline

```
...
agent any
...
```

Agent Label Matching

- Expression matching works just like it would from the Project Configuration view.

```
// Matches our 'Slave 1' agent
agent {
label 'Slave 1'
}
// Matches agents with the CentOS label
agent {
label 'CentOS'
}
// Matches our 'master'
agent {
label 'master'
}
```

The Docker Agent

- You can utilize Docker for agents.

- It will reach out to the Docker Hub if the container isn't available locally.
- It locates the container image based on the string you pass the directive.
- You must have Docker configured on the agents/master that in order to utilize it as an agent.

```
// utilized the openjdk:8u121-jre image from the Docker Hub
stage('Test on Debian') {
  agent {
    docker 'openjdk:8u121-jre'
  }
  steps {
    sh "wget http://brandon4232.mylabserver.com/rectangles/all/
rectangle_${env.BUILD_NUMBER}.jar"
    sh "java -jar rectangle_${env.BUILD_NUMBER}.jar 2 3"
  }
}
```

Agent Any and None

- If you declare an agent in the stage scope, you must declare "agent none" at the top scope.

```
// Used when agents are defined in the "stage" scope
...
agent none
stages {
  stage('Unit Testing') {
    agent {
      label 'Slave 1'
    }
  }
}
...
// Use "any" when any agent will work
...
agent any
stages {
  stage('SO Simple') {
    steps {
      sh 'echo "I can run on any agent, except for Windows ;-)"'
    }
  }
}
```

The Stages and Stage Directives

- "Stages" just says "here are my stages."
- A stage typically describes some step in the SDLC.
  - Build
  - Test

- Deploy

- Stages are not at all limited by that scope though.

```
stages {
  // 'Build' is simply setting the name of the stage
  stage('Build') {
    ...
  }
  stage('Test') {
    ...
  }
  stage('Deploy') {
    ...
  }
}
```

The Steps Directive

- You define the actual steps in the stage here.

- This would be analogous to a "Build Step" in the classic Project Configuration view.

```
steps {
  sh 'printenv'
  sh 'ant -f test.xml -v'
  junit "reports/${env.BUILD_NUMBER}_result.xml"
}
```

Common "Steps"

- **sh:** runs a shell script

- **echo:** prints a string

- **junit:** processes a junit report

- There are many more associated with various plugins.

The Post Directive

- Contains "Post-build" steps

```
post {
  // This will always run, whether or not the previous steps
succeeded
  always {
    // Archiving the jar file, tracking fingerprint
    archiveArtifacts artifacts: 'dist/*.jar', fingerprint: true
  }
}
```

Conditionals

- **always:**
    - Run regardless of the completion status of the Pipeline run.
- **changed:**
    - Only run if the current Pipeline run has a different status from the previously completed Pipeline.
- **failure:**
    - Only run if the current Pipeline has a "failed" status, typically denoted in the web UI with a red indication.
- **success:**
    - Only run if the current Pipeline has a "success" status, typically denoted in the web UI with a blue or green indication.
- **unstable:**
    - Only run if the current Pipeline has an "unstable" status, usually caused by test failures, code violations, etc.
    - Typically denoted in the web UI with a yellow indication.

The Environment Directive

- You can set custom environment variables with this directive.
- Can be set in the stage and pipeline scopes
- Accessed like any other environment variable

```
environment {
  ENV_VAR = "my value"
}
```

The Triggers Directive

- You can specify a cron or pollSCM trigger with this.
- It's currently limited to those options, so it may not always make sense to use this for your project.
- It may be preferable to use the Project Configuration view for configuring triggers.

```
...
// Using the cron trigger
triggers {
    cron('H 4/* 0 0 1-5')
}
```

```
...
// Using the pollSCM trigger
triggers {
  pollSCM('H 4/* 0 0 1-5')
}
```

The Parameters Directive

- Use this directive to set parameters for your project
- Currently only allows for string and boolean parameters

```
parameters {
  string(name: 'PERSON', defaultValue: 'Mr Awesome', description:
'Who's the best?')
  booleanParam(name: 'IS_JENKINS_AWESOME', defaultValue: true,
description: "Jenkins Awesome?" )
}
```

Global Libraries

- You can extend the functionality of Pipelines with a shared library.
- Scripts are written in Groovy.
- Can live in their own repositories

Repo Layout

```
(root)
+- src                     # Groovy source files
|   +- org
|       +- foo
|           +- Bar.groovy  # for org.foo.Bar class
+- vars
|   +- foo.groovy          # for global 'foo' variable
|   +- foo.txt             # help for 'foo' variable
+- resources               # resource files (external libraries only)
|   +- org
|       +- foo
|           +- bar.json    # static helper data for org.foo.Bar
```

Configuration Navigation

Dashboard -> Manage Jenkins -> Configure System -> Global Pipeline Libraries

Example

1. Set up an empty Git repo on GitHub.

2. Make a directory called "vars".

3. Create a file in the "vars" directory called "sayHello.groovy."

4. Populate the file with the following:

```
// vars/sayHello.groovy
def call(String name = 'human') {
    // Any valid steps can be called from this code, just like in other
    // Scripted Pipeline
    echo "Hello, ${name}."
}
```

5. Commit and Push to your origin.

6. Go to the Jenkins Dashboard.

7. Click "Manage Jenkins," then "Configure System."

8. Go to the "Global Pipeline" and click "Add."

9. Add a Name for the library, like "My Pipeline Library."

10. Set "master" for the "Default version." This can be any Git ref available from the origin (branch, tag, commit, etc...).

11. Check "Load implicitly."

12. Check "Modern SCM."

13. Check "GitHub."

14. Input your GitHub username in the "Owner" field.

15. Select your repository from the "Repository" dropdown list.

16. Click "Save."

17. You can now invoke the "sayHello" function from a Jenkinsfile.

In the Jenkinsfile

```
stage('Unit Tests') {
  steps {
    sh 'ant -f test.xml -v'
    junit 'reports/result.xml'
    //here it is
    sayHello 'Pinehead'
  }
}
```

Console Output

```
...
Recording test results
[Pipeline] echo
Hello, Pinehead.
[Pipeline] }
...
```

Adding Your First Pipeline

Prerequisites

- You have a Git repository with a project you'd like to set up for a Pipeline.
- You have a Jenkins setup and configured per the steps in the video/guide.
- Appropriate keys are set up for GitHub.
- Use the below example if you just want to learn the process:

Sample Repo Setup Procedure

1. Create a new repo on GitHub.
    - Sign up for a new account if necessary.
    - Call the repo "first-jenkins-pipeline."
2. Clone to your dev environment.
3. From the terminal, set the git global config items if necessary:
    - `git config --global user.name "Your Name"`
    - `git config --global user.email you@example.com`
4. Create a Jenkinsfile in the root path.
    - `vi Jenkinsfile`
    - Add the following in a Jenkinsfile in the root path of the git repo:

```
pipeline {
    agent { docker 'ruby' }
    stages {
        stage('build') {
            steps {
                sh 'ruby --version'
            }
        }
    }
}
```

5. Add the file to the repo's index.

    - `git add .`

6. Commit the change.

    - `git commit -am "initial commit with Jenkinsfile"`

7. Push the change to origin

    - `git push origin master`

Pipeline Configuration Procedure

1. Head over to your Jenkins Web GUI and log in: `http://your-server.mylabserver.com:8080/login`

2. Click "New Item" on the left navigation menu.

3. Set the name to "My First Pipeline" under "Enter an Item Name"

4. Then, select "Multibranch Pipeline" and click "OK".

5. Under "Branch Sources" in the configuration menu, select "GitHub" from the dropdown.

6. Place your GitHub username in the "Owner" field.

7. Then select your repository from the repository dropdown.

8. Click "Save"

9. If all went well, you'll see the job make an initial run and pass.

Jenkins REST API

- REST calls in XML or JSON format
- There are language specific APIs available.
- You'd use the REST API for programatic calls to Jenkins, like a post/pre-commit hook.

- Using REST API to trigger jobs remotely, access job status, create/delete/copy jobs
- There are Ruby/Python/XML/JSON APIs.

Disabling "Prevent Cross Site Request Forgery exploits"

- You man have trouble authenticating via the REST api for some commands with some versions of Jenkins.
- Disabling "Prevent Cross Site Request Forgery exploits" can resolve this.
- Uncheck the box from "Manage Jenkins" -> "Configure Global Security"

Generating a Token

- You can generate a crumb to authenticate with the Jenkins server
- You should use this, and have $CRUMB available via Environment Variables, before attempting the other commands below.

```
CRUMB=$(curl -s 'http://USER:PASS@<your-server>:8080/crumbIssuer/api/
xml?xpath=concat(//crumbRequestField,":",//crumb)')
```

Setting Up a Token

1. From the Project Configuration view, go to the "Build Triggers" section.
2. Check "Trigger builds remotely (e.g., from scripts)"
3. Input some string for an Authentication Token. (e.g., "build")

Submitting Jobs

Jobs Without Parameters

- HTTP POST on `JENKINS_URL/job/JOBNAME/build?token=TOKEN`
- `%20` for spaces in the URL
- Example:

```
// The token is "build" in this example.
curl -X POST -H "$CRUMB" http://<your-server>.com:8080/job/My%20
Freestyle%20Project/build?token=build
```

Jobs with Parameters

- HTTP POST on 'JENKINS_URL/job/JOBNAME/build?token=TOKEN'
- '%20' for spaces in the URL
- Example:

  ```
  curl -X POST -H "$CRUMB" http://<your-server>.com:8080/job/My%20
  Freestyle%20Project/build \
  +-data token=build \
  +-data-urlencode json='{"parameter": [{"name":"branch",
  "value":"development"}]}'
  ```

JSON Format

```
{
  "parameter": [
    {
      "name":"first_param",
      "value":"first_value"
    },
    {
      "name":"second_param",
      "value":"second_value"
    }
  ]
}
```

Other Common API Calls

Retrieve or Update the Config

- You can retrieve and update the project configuration via XML or JSON.
- You can programmatically update the xml and "PUT" it back to change config.

  ```
  //xml format
  curl -H "$CRUMB" http://<your-server>.com:8080/job/Freestyles/job/
  My%20Freestyle%20Project/config.xml
  //json format
  curl -H "$CRUMB" http://<your-server>.com:8080/job/Freestyles/job/
  My%20Freestyle%20Project/api/json
  ```

config.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<project>
  <actions/>
  <description>My first project</description>
  <keepDependencies>false</keepDependencies>
  <properties>
    <com.coravy.hudson.plugins.github.GithubProjectProperty
```

```
plugin="github@1.26.0">
      <projectUrl>https://github.com/labmac/jenkins-test/</projectUrl>
      <displayName></displayName>
    </com.coravy.hudson.plugins.github.GithubProjectProperty>
    <hudson.model.ParametersDefinitionProperty>
      <parameterDefinitions>
        <hudson.model.StringParameterDefinition>
          <name>BRANCH</name>
          <description></description>
          <defaultValue>master</defaultValue>
        </hudson.model.StringParameterDefinition>
      </parameterDefinitions>
    </hudson.model.ParametersDefinitionProperty>
  </properties>
  <scm class="hudson.plugins.git.GitSCM" plugin="git@3.0.5">
    <configVersion>2</configVersion>
    <userRemoteConfigs>
      <hudson.plugins.git.UserRemoteConfig>
        <url>git@github.com:labmac/jenkins-test.git</url>
        <credentialsId>redacted</credentialsId>
      </hudson.plugins.git.UserRemoteConfig>
    </userRemoteConfigs>
    <branches>
      <hudson.plugins.git.BranchSpec>
        <name>*/$BRANCH</name>
      </hudson.plugins.git.BranchSpec>
    </branches>
    <doGenerateSubmoduleConfigurations>false</
doGenerateSubmoduleConfigurations>
    <browser class="hudson.plugins.git.browser.GithubWeb">
      <url>https://github.com/labmac/jenkins-test/</url>
    </browser>
    <submoduleCfg class="list"/>
    <extensions/>
  </scm>
  <assignedNode>Linux &amp;&amp; CentOS</assignedNode>
  <canRoam>false</canRoam>
  <disabled>false</disabled>
  <blockBuildWhenDownstreamBuilding>false</
blockBuildWhenDownstreamBuilding>
  <blockBuildWhenUpstreamBuilding>false</blockBuildWhenUpstreamBuilding>
  <triggers>
    <com.cloudbees.jenkins.GitHubPushTrigger plugin="github@1.26.0">
      <spec></spec>
    </com.cloudbees.jenkins.GitHubPushTrigger>
  </triggers>
  <concurrentBuild>false</concurrentBuild>
  <builders>
    <hudson.tasks.Shell>
      <command>git log</command>
    </hudson.tasks.Shell>
    <hudson.plugins.parameterizedtrigger.TriggerBuilder
plugin="parameterized-trigger@2.33">
      <configs>
        <hudson.plugins.parameterizedtrigger.
BlockableBuildTriggerConfig>
```

```
              <configs>
                <hudson.plugins.parameterizedtrigger.
    PredefinedBuildParameters>
                  <properties>IMPORTANT_PARAMETER=$BUILD_NUMBER</properties>
                </hudson.plugins.parameterizedtrigger.
    PredefinedBuildParameters>
              </configs>
              <projects>Downstream Project</projects>
              <condition>ALWAYS</condition>
              <triggerWithNoParameters>false</triggerWithNoParameters>
              <triggerFromChildProjects>false</triggerFromChildProjects>
              <buildAllNodesWithLabel>false</buildAllNodesWithLabel>
            </hudson.plugins.parameterizedtrigger.
    BlockableBuildTriggerConfig>
          </configs>
        </hudson.plugins.parameterizedtrigger.TriggerBuilder>
        <hudson.tasks.Shell>
          <command>docker run hello-world</command>
        </hudson.tasks.Shell>
        <hudson.plugins.copyartifact.CopyArtifact
    plugin="copyartifact@1.38.1">
          <project>My Java Project</project>
          <filter>dist/rectangle.jar</filter>
          <target></target>
          <excludes></excludes>
          <selector class="hudson.plugins.copyartifact.
    StatusBuildSelector"/>
          <doNotFingerprintArtifacts>false</doNotFingerprintArtifacts>
        </hudson.plugins.copyartifact.CopyArtifact>
        <hudson.tasks.Shell>
          <command>java -jar dist/rectangle.jar 4 5</command>
        </hudson.tasks.Shell>
      </builders>
      <publishers/>
      <buildWrappers/>
    </project>
```

Deleting a Job

- Make sure you're sure before using this one.

  ```
  curl -H "$CRUMB" http://<your-server>.com:8080/job/Freestyles/job/
  My%20Freestyle%20Project/doDelete
  ```

Disable/Enable a Job

```
//to Disable
curl -H "$CRUMB" http://<your-server>.com:8080/job/Freestyles/job/My%20
Freestyle%20Project/disable


//to Enable
curl -H "$CRUMB" http://<your-server>.com:8080/job/Freestyles/job/My%20
```

```
Freestyle%20Project/disable
```

Update description

```
curl -H "$CRUMB" http://<your-server>.com:8080/job/Freestyles/job/My%20
Freestyle%20Project/description
```

Jenkins CLI

Set Up a Pub Key for Your User

1. Generate a rsa key pair for your user.
2. Go to: http://yourserver.com/me/configure
3. Copy the contents of your id_rsa.pub file into the "SSH Public Keys" field.

Download the CLI Jar file

```
wget -P /var/lib/jenkins/ http://<your-server>:8080/jnlpJars/jenkins-
cli.jar
```

Executing Commands

```
java -jar /var/lib/jenkins/jenkins-cli.jar -s http://<your-server>:8080/
<command>
```

(Optional) Setting Up Your Environment

- You can use an alias and environment variables to make the syntax a bit simpler.
- The instructions below assume you're running from the Jenkins Master and from a CentOS system.
- Make changes as necessary if that's not the case.

1. Set the JENKINS_URL environment variable.

```
echo "JENKINS_URL='http://localhost:8080'" >> /etc/environment
```

2. Set an alias for your bash shell.

```
echo "alias jenkins-cli='java -jar /var/lib/jenkins/jenkins-cli.jar'" >>
```

```
~/.bashrc
```

3. The Jenkins CLI can now be executed via `jenkins-cli`

Common Commands

```
// Returns user
[root@brandon4231 ~]# jenkins-cli who-am-i
Authenticated as: brandon
Authorities:
authenticated


// Returns Overall Help
[root@brandon4231 ~]# jenkins-cli help
add-job-to-view
Adds jobs to view.
build
Builds a job, and optionally waits until its completion.
...


// Triggers a build on a project
[root@brandon4231 ~]# jenkins-cli build "Freestyles/My Freestyle
Project"


// Returns the Jenkins version
[root@brandon4231 ~]# jenkins-cli version
2.19.4


// Shuts down Jenkins
[root@brandon4231 ~]# jenkins-cli shutdown


//Safely shuts down Jenkins (waits for builds to finish)
[root@brandon4231 ~]# jenkins-cli safe-shutdown


// Restarts Jenkins
[root@brandon4231 ~]# jenkins-cli restart


//Safely restarts Jenkins
[root@brandon4231 ~]# jenkins-cli safe-restart


// Installs Plugins
[root@brandon4231 ~]# jenkins-cli install-plugin thinBackup -restart
Installing thinBackup from update center
```

```
// Returns console output from a build
[root@brandon4231 ~]# jenkins-cli console "Freestyles/My Freestyle
Project" 51
Started from command line by
ha:AAAAmR+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT+vOD8nVc83PyU1x6OyILUoJz
Mv2y+/JJUBAhiZGBgqihhk0NSjKDWzXb3RdlLBUSYGJk8GtpzUvPSSDB8G5tKinB
IGIZ+sxLJE/ZzEvHT94JKizLx0a6BxUmjGOUNodHsLgAz2EgZe/dLi1CL9pKLEvJT8P
ACqY1WJwgAAAA==brandon


Building remotely on ha:AAAAnR+LCAAAAAAAAP9b85aBtbiIQTGjNKU4P08vOT+vO
D8nVc83PyU1x6OyILUoJzMv2y+/JJUBAhiZGBgqihhk0NSjKDWzXb3RdlLBUSYGJk
8GtpzUvPSSDB8G5tKinBIGIZ+sxLJE/ZzEvHT94JKizLx0a6BxUmjGOUNodHsLgAz2EgZB/
eT83ILSktQi/eCcxLJUBUMALBjAR8YAAAA=Slave 1 (Linux CentOS) in workspace /
var/lib/jenkins/workspace/Freestyles/My Freestyle Project
> git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
...
```

Jenkins Groovy Script Console

- Allows you to run arbitrary scripts from the Dashboard
- Can be used for administration purposes, or just for information
- Checkout the following URL for a ton of scripts:
    - https://github.com/jenkinsci/jenkins-scripts/tree/master/scriptler

Navigation

Dashboard -> Manage Jenkins -> Script console

Examples

- Workspace Cleaner
    - https://github.com/jenkinsci/jenkins-scripts/blob/master/scriptler/workspace-cleaner.groovy

```
/*** BEGIN META {
  "name" : "Workspace Cleaner",
  "comment" : "This script will go through all workspaces for any/all
jobs and remove them.",
  "parameters" : [],
  "core": "1.300",
  "authors" : [
    { name : "EJ Ciramella" }
  ]
} END META**/
import hudson.FilePath;
```

```
for (slave in hudson.model.Hudson.instance.slaves)
{
  FilePath fp = slave.createPath(slave.getRootPath().toString() +
File.separator + "workspace");
  fp.deleteRecursive();
```

- Plugin Printer
  - https://github.com/jenkinsci/jenkins-scripts/blob/master/scriptler/pluglist_print.groovy

```
/*** BEGIN META {
  "name" : "plugins lister",
  "comment" : "print list of plugins (optionally set build.
displayName)",
  "parameters" : [],
  "core": "0.601",
  "authors" : [
    { name : "Mark Hudson" }
  ]
} END META**/
def pcount = 0 ; def pstr = ''
def plist = jenkins.model.Jenkins.instance.pluginManager.plugins
plist.sort{it}  // plugins list it defaults to shortName
plist.each {
  pcount = pcount + 1
  pname = (pcount + ' ' + it).replaceAll("Plugin:", '')
  pstr = pstr + ' ' + pname + ' ' + it.getVersion() + "\n"  // +
"<br>"
}
println pstr  // Console Output
if ( "executable" in Thread.currentThread().getProperties() ) {
  // print Thread.currentThread().getProperties()
  def manager_build = Thread.currentThread().executable ; assert
manager_build  // non-Postbuild context
  manager_build.displayName =  "#" + manager_build.number + " had " +
pcount + " plugins"
  } else { // Pipeline Workflow DSL
    currentBuild.displayName = "#" + currentBuild.number + " had " +
pcount + " plugins"
  }
return
```

- Get Next Build Numbers
  - https://github.com/jenkinsci/jenkins-scripts/blob/master/scriptler/getNextBuildNumbers.groovy

```
/*** BEGIN META {
 "name" : "Print next build numbers for all jobs",
 "comment" : "This script ouputs a JSON-formatted listing of the next
build numbers for all jobs recursively. If you are interested in only
a subset of the jobs, please specify the root folder explicitly. The
output JSON can be captured in a file, copied over to another Jenkins
server and used with the setNextBuildNumbers.groovy script. The idea is
to ensure that build numbers do not get reset to 1 when migrating jobs
from one Jenkins server to another.",
 "parameters" : ['rootItem'],
 "core": "1.625",
 "authors" : [
 { name : "Amit Modak" }
 ]
 } END META**/
import jenkins.model.*
def getBuildNumber(def item, def node) {
  if(item instanceof com.cloudbees.hudson.plugins.folder.Folder) {
    node[item.getName()] = [:]
    item.getItems().each {
      getBuildNumber(it, node[item.getName()])
    }
  } else {
    node[item.getName()] = item.nextBuildNumber
  }
}
//
// main
//
def root = [:]
def node = root
if(rootItem) {
  if(Jenkins.instance.getItemByFullName(rootItem) && (Jenkins.instance.
getItemByFullName(rootItem) instanceof com.cloudbees.hudson.plugins.
folder.Folder)) {
    rootItem.split('/').each {
      node[it] = [:]
      node = node[it]
    }
    Jenkins.instance.getItemByFullName(rootItem).getItems().each {
      getBuildNumber(it, node)
    }
  } else {
    println "Error: '" + rootItem + "' does not exist or is not a
folder"
    return
  }
} else {
  Jenkins.instance.getItems().each {
    getBuildNumber(it, node)
  }
}
def output = groovy.json.JsonOutput.toJson(root)
println groovy.json.JsonOutput.prettyPrint(output)
```