

**WA2917 Booz Allen Hamilton Tech
Excellence Cloud Engineering
Program - Phase 1**

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Using the GitFlow Workflow.....	3
Lab 2 - Monolith vs Microservices Design.....	15
Lab 3 - Build Script Basics.....	19
Lab 4 - Build Java Project, Management Package Dependencies, and Run Unit Tests with Gradle.....	28
Lab 5 - A Simple RESTful API in Spring Boot.....	37
Lab 6 - Overview of Common Microservices Runtime Issues.....	49
Lab 7 - Configure Tools in Jenkins.....	67
Lab 8 - Create a Jenkins Job.....	71
Lab 9 - Create a Pipeline.....	85
Lab 10 - Advanced Pipeline with Groovy DSL.....	96
Lab 11 - Configure Jenkins Security.....	108

Lab 1 - Using the GitFlow Workflow

In this lab you will explore the GitFlow workflow to create features, releases, hotfixes, and various other types of branches.

Part 1 - Create a directory for storing GitFlow lab content.

In this part you will create a directory for storing GitFlow lab content.

__ 1. Open a command prompt window, and then enter:

```
cd \Workspace
```

Note: If C:\workspace directory doesn't exist, create it manually.
--

__ 2. Create a directory for storing GitFlow lab content:

```
mkdir gitflow_test
```

__ 3. Switch to the created directory:

```
cd gitflow_test
```

Part 2 - Create and setup Remote and Local repositories.

In this part you will create a remote and a local repository.

__ 1. Initialize a new bare repository:

```
git init --bare remote_repo.git
```

__ 2. Verify the remote repository is created:

```
tree remote_repo.git
```

Notice it shows directory tree like this:

```
C:\WORKSPACE\GITFLOW_TEST\REMOTE_REPO.GIT
├── hooks
├── info
├── objects
│   ├── info
│   └── pack
├── refs
│   ├── heads
│   └── tags
```

__3. Clone the bare remote repository to a local working copy:

```
git clone remote_repo.git local_repo
```

Notice it's a blank repository that's why there is warning displayed on the screen.

__4. Verify local repository is created:

```
tree local_repo
```

Notice the directory structure is blank.

__5. Switch to the local working copy:

```
cd local_repo
```

__6. Configure user email for the repository:

```
git config user.email "bob@abc.com"
```

__7. Configure user name for the repository:

```
git config user.name "Bob"
```

Part 3 - Set up the Basic GitFlow branches in the local repository

GitFlow suggests that we should have a 'master' branch that represents our latest release, and a 'develop' branch that holds ongoing work towards the next release. We will create those now.

__1. In the steps above, we created an empty repository and then cloned it. Git won't let us create branches until we have at least one commit in the repository, so we'll do that now, with an empty commit:

```
git commit --allow-empty -m "Initial commit."
```

__2. Create a 'develop' branch and switch to it as shown:

```
git checkout -b develop
```

__3. Run the following command:

```
git flow init
```

__4. Press the **Enter** key to use default values for all options.

__5. View the branch list:

```
git branch
```

Notice it shows following branches:

```
C:\workspace\gitflow_test\local_repo>git branch
* develop
master
```

"develop" is the active branch. In case if you want to activate a different branch, use following command:

```
git checkout <branch>
```

Part 4 - Create a Gitflow feature.

In this part you will create a Gitflow feature.

__1. Create a new feature. Recall that we build features on a feature branch. You could name this anything, but we'll use a common convention of naming the feature branches 'feature/FNAME':

```
git checkout -b feature/HOME_PAGE
```

__2. Get branch list:

```
git branch
```

Notice it shows following branches:

```
C:\workspace\gitflow_test\local_repo>git branch
develop
* feature/HOME_PAGE
master
```

feature/HOME_PAGE feature has been added as a branch.

__3. Get git status:

```
git status
```

Notice **feature/HOME_PAGE** is the active branch and currently there are no files to commit.

Part 5 - Add content as part of the feature.

In this part you will create an HTML file as part of the feature created in the previous part.

__1. Start Notepad, click Yes to create the file:

```
notepad index.html
```

__2. In the Notepad window enter following text:

```
<html>
<body>
</body>
</html>
```

__3. Save the file and close the Notepad window:

__4. Get git status:

```
git status
```

Notice that 'index.html' is listed as 'untracked'.

__5. Add the new file to the index for the next commit:

```
git add index.html
```

__6. Get git status:

```
git status
```

Notice index.html is added the repository but it's not committed yet.

__7. Commit changes:

```
git commit -m "Added index.html"
```

__8. View **feature/HOME_PAGE** branch contents:

```
git ls-tree -r --name-only feature/HOME_PAGE
```

Notice it shows index.html. If you don't use --name-only switch, it will show file size in addition to the file ID.

__9. Switch to the 'develop' branch and merge this feature into it. Since we want a record of this merge, we'll use '--no-ff' to make sure there's a merge commit recorded:

```
git checkout develop
git merge --no-ff feature/HOME_PAGE
```

__10. View **develop** branch contents:

```
git ls-tree -r --name-only develop
```

__11. Switch back to the feature branch:

```
git checkout feature/HOME_PAGE
```

__12. Open index.html file in notepad:

```
notepad index.html
```

__13. Add <h1> tag to the **body** tag:

```
<h1>Hello world!</h1>
```

__14. Save the file and close the Notepad window.

__15. Get git status:

```
git status
```

Notice index.html file is updated on the file system, but it's not updated in the repository. Also notice you are on the develop branch.

__16. Stage the changed index.html:

```
git add index.html
```

__17. Commit changes to the feature branch:

```
git commit -m "Added h1 tag to the index.html"
```

__18. Merge **feature** branch into the **develop** branch without deleting the feature branch:

```
git checkout develop  
git merge --no-ff feature/HOME_PAGE
```

__19. View remote repository content:

```
git ls-remote
```

Notice currently no branches are in the remote repository. All the changes are in the local repository right now.

__20. Push changes from local repository into the remote repository:

```
git push --all
```

__21. View remote repository content:

```
git ls-remote
```

Notice it displays **develop** and **master** branches.

Part 6 - Create a new Release

In this part you will create a Gitflow release.

__1. Create a new release "REL_1.0" based on **develop** branch. We'll use the common convention of calling the release "release/REL_1.0". Enter the following lines, pressing 'return' after each line:

```
git checkout develop
git checkout -b release/REL_1.0
```

In a real situation, you might need to do some commits on the release branch, at the very least to change the version number if it appears in the source code. You might also find things in pre-release testing that need to be fixed. You'd normally commit the required changes on the release branch.

__2. Let's finish the release. As a reminder, the GitFlow workflow suggests that we should merge the release branch into 'master' and then tag the release. Enter the following lines, pressing 'return' after each line:

```
git checkout master
git merge release/REL_1.0
git tag REL_1.0 -m "REL_1.0"
```

Notice that we are using the '-m' option to supply a tag message. This option implies '-a', which causes Git to create an 'annotated' tag. In a real situation, you might want to include more information in the tag message.

__3. Now, we should get rid of the release branch and then update the 'develop' branch to reflect the latest release (as reflected by the state of the 'master' branch). Enter the following lines, pressing 'return' after each line:

```
git branch -d release/REL_1.0
git checkout develop
git merge master
```

Notice that when we merged 'master', we got a message saying that the develop branch is already up-to-date. That's may or may not be the case in real life. In fact if development has continued in parallel with the release testing, you might have to resolve one or two merge conflicts.

__4. Get branch list:

```
git branch
```

Notice it shows branch list like this:

```
C:\workspace\gitflow_test\local_repo>git branch
* develop
  feature/HOME_PAGE
  master
```

__ 5. Push all changes to the remote repository:

```
git push --all
```

__ 6. Switch to the feature/HOME_PAGE branch:

```
git checkout feature/HOME_PAGE
```

__ 7. Open index.html file in Notepad:

```
notepad index.html
```

__ 8. Below h1 tag, add following text:

```
Welcome to ABC Inc.
```

Notice you haven't enclosed the above text in any tag. You are purposefully skipping a tag. You will treat as a bug and fix it later.

__ 9. Save the file and close the Notepad window.

__ 10. Stage the changed index.html:

```
git add index.html
```

__ 11. Commit the changes:

```
git commit -m "Added company name to index.html"
```

__ 12. Merge feature changes into the **develop** branch without deleting the feature branch. Enter the following lines, pressing 'return' after each line:

```
git checkout develop
git merge --no-ff feature/HOME_PAGE
```

__13. Start another release from **develop** branch:

```
git checkout -b release/REL_2.0
```

__14. ...and finish it off by merging it back into master and tagging it:

```
git checkout master
git merge release/REL_2.0
git tag "REL_2.0" -m "REL_2.0"
```

(note that we haven't really accomplished much here - we could have got the same result by merging the feature branch into 'master' and tagging it, but we want to get in the habit of merging the release branches properly. Also, some shops like to use '--no-ff' to make sure the merge is recorded).

__15. Finally, let's merge back to 'develop' and get rid of the release branch:

```
git checkout develop
git merge master
git branch -d release/REL_2.0
```

__16. View tag list:

```
git tag
```

Notice it shows REL_1.0 and REL_2.0.

If you want to checkout a certain release based on tag, you can execute following command:

```
git checkout tags/<tag_name>
```

Part 7 - Working with Hotfixes.

In this part you will create a Gitflow hotfix.

__1. Create a new hotfix, based on the 'master' branch (of course you could base it on a different release tag if needed):

```
git checkout master
git checkout -b hotfix/REL_2.0.1
```

Notice hotfix will end up creating a new release.

__2. Open index.html file in the Notepad:

```
notepad index.html
```

__3. Enclose Welcome to ABC Inc. within HTML tag.

__4. Save the file and close the Notepad window.

__5. Stage the changed index.html file:

```
git add index.html
```

__6. Commit changes:

```
git commit -m "Enclosed company name in the bold HTML tag"
```

__7. Finish the hotfix by entering the following commands, pressing 'return' after each line:

```
git checkout master
git merge hotfix/REL_2.0.1
git tag "REL_2.0.1" -m "REL_2.0.1"
git branch -d hotfix/REL_2.0.1
```

__8. ...and, let's bring the change back to 'develop':

```
git checkout develop
git merge master
```

__9. Finally, push all changes into the remote repository:

```
git push --all
```

Part 8 - Verify all releases are created

In this part you will verify all releases are created.

__1. Get tag list:

```
git tag
```

Notice REL_1.0, REL_2.0, and REL_2.0.1 are available.

__2. View current tag name:

```
git describe
```

Notice the prefix is REL_2.0.1

__3. Checkout REL_1.0:

```
git checkout tags/REL_1.0
```

__4. View current tag name:

```
git describe
```

Notice the prefix is REL_1.0

__5. View index.html file contents:

```
type index.html
```

Notice there's no "Welcome to ABC Inc." text.

__6. Checkout REL_2.0:

```
git checkout tags/REL_2.0
```

__7. View current tag name:

```
git describe
```

Notice the prefix is REL_2.0

__8. View index.html file contents:

```
type index.html
```

Notice there's "Welcome to ABC Inc." text, but it's not enclosed in tag.

__9. Checkout REL_2.0.1:

```
git checkout tags/REL_2.0.1
```

__10. View current tag name:

```
git describe
```

Notice the prefix is REL_2.0.1

__11. View index.html file contents:

```
type index.html
```

Notice there's "Welcome to ABC Inc." text enclosed in tag.

__12. Close all.

Part 9 - Review

In this lab you explored Gitflow feature, release, hotfix, and various other options.

Lab 2 - Monolith vs Microservices Design

In this lab, you will be presented with two types of application architecture for a fictitious Java EE application: the monolith and a microservices-based one. You will be required to identify the main differences in their designs and answer some questions. For certain questions there is no single answer; feel free to share your answer(s) with the rest of the group.

Part 1 - Breaking the Monolith

__1. Compare the Monolith Application Architecture 1 (Fig.1) with Microservices Architecture 2 (Fig.2), and name some of the differences.

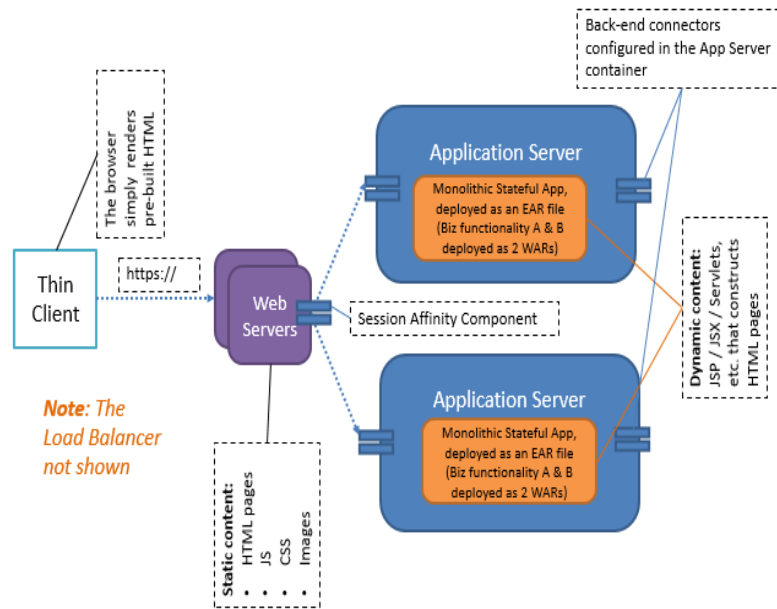


Fig 1. Application Architecture 1: Traditional (Java) Enterprise Application Architecture Example.

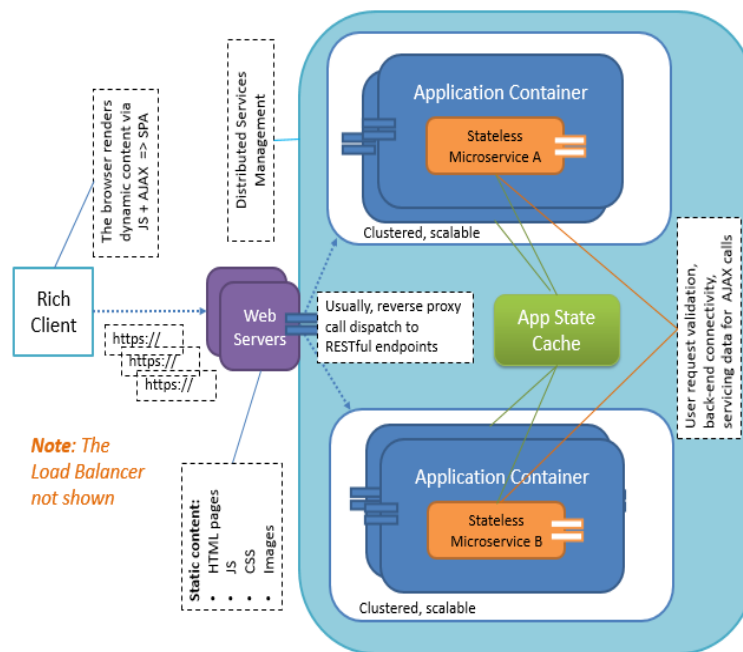


Fig. 2 Application Architecture 2: Microservices-based Architecture Example.

- ___2. What kind of Quality of Service properties we gain / lose by going from Architecture 1 to Architecture 2?
- ___3. Which type of architecture more easily support scalability?
- ___4. What would be involved if you need to change a single Java file (e.g. a servlet)?
- ___5. In which type of architecture you need to have more powerful computers? Is it about CPU, RAM, or both?
- ___6. You need to recycle the machines hosting the applications. In which case you will have a shorter application start up time?

Part 2 - The 12-Factor App

Go through the twelve-factor app principles [<http://12factor.net>] and try to see how, if at all, that methodology is applied in both App Architecture (Fig. 1 and Fig.2).

For example, you can notice that the Application in Fig.1 may be hard to scale through the process principle (VIII) that is broken as all code is bundled together (in one bundle / archive).

Mind you that not all factors can be identified here, though.

Part 3 - Microservices Pros and Cons

Which of the following statements you believe may be related to microservices, and, if so, which ones should be viewed as being an argument in favor or against using microservices. Compare/share your findings with the class, if you wish so.

1. As my app is deployed and executed in its own run-time (often in a container or a VM of sorts), I can get stronger process isolation properties.
2. Change cycles for different services can be more easily decoupled. For example, I can re-deploy Service A without affecting Service B.
3. I have to deal with performance overhead related to inter-process communication.
4. Deployment cycles and developer velocity are faster (due to the smaller footprint of the service).
5. With this type of app, I can do scaling per service / per tier.
6. My app boasts a fantastic performance and fast service interactions; it also has a nice and small security perimeter.
7. Now I have to deal with a considerable operational overhead (the accidental complexity) with more distinct moving parts that require precisely orchestrated deployment and distributed monitoring.
8. My application has all the required services housed in one place and they get deployed using a single archive. The app is designed using the standard SOA

principles: its services are mapped to distinct business capabilities, the services are narrow, composable, and accessible through a well-defined WSDL-based web service interface.

9. Now it is more of a nanoservice, really. Which is really cool, right?
10. Smaller development teams can be set up to develop and maintain the services which leads to faster getting-up-to-speed and development cycles.
11. Now, with this type of design, you must persist any state in a reliable external (e.g. caching) service.

Part 4 - Review

In this lab, we compared two types of application architecture: the monolith and one based on microservices.

Lab 3 - Build Script Basics

Gradle was already configured for you under **C:\Software\gradle-6.6.1** folder and added in the environment variables; Gradle can be downloaded by following this URL: <https://gradle.org/gradle-download/>. **Complete Distribution** contains the binary files, documentation, and source code. If you prefer, you can just download the binary files.

In this lab you will build a Java Project, manage dependencies, and run unit tests with Gradle.

Part 1 - Gradle installation verification

- __ 1. Open a Command Prompt.
- __ 2. Verify Gradle is installed:

```
gradle -version
```

In this lab you will create build scripts and utilize various features.

Part 2 - Create directory structure for storing build script

In this part you will create directory structure for storing build script.

- __ 1. Switch to the Workspace directory:

```
cd C:\Workspace
```

- __ 2. Create the directory structure:

```
mkdir labs\gradle\basics
```

- __ 3. Switch to the newly created directory:

```
cd labs\gradle\basics
```

Part 3 - Create Gradle build script and use shortcut task definition technique for defining a task

In this part you will create a Gradle build script and add a task using shortcut task definition technique.

__1. Create build.gradle file:

```
notepad build.gradle
```

__2. Click Yes to create the file.

__3. Enter following code:

```
// let's add a simple task
task hello1 {
    println 'Hello World!'
}
```

Here you have added a single-line comment and a task which displays a message on the screen.

__4. Save the file.

__5. Run Gradle:

```
gradle
```

Notice it's asking us to pass it more parameters.

```
C:\Workspace\labs\gradle\basics>gradle
Starting a Gradle Daemon (subsequent builds will be faster)

> Configure project :
Hello World!

> Task :help

Welcome to Gradle 6.6.1.

To run a build, run gradle <task> ...

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle --help

To see more detail about a task, run gradle help --task <task>

For troubleshooting, visit https://help.gradle.org

BUILD SUCCESSFUL in 8s
1 actionable task: 1 executed
C:\Workspace\labs\gradle\basics>_
```

__ 6. Get task list:

```
gradle tasks --all
```

Notice it lists 'hello1' under "Other tasks" section.

__ 7. Execute hello1 task:

```
gradle hello1
```

Notice it displays "Hello World!" message on the screen, but there are additional messages logged as well.

__ 8. Execute the task by suppressing the log:

```
gradle -q hello1
```

Notice this time it just displays the message.

Part 4 - Create a task with actions

In this part you will create another task which uses actions.

__ 1. In the Notepad window, which contains build.gradle code, append following code:

```
// add another task with actions
task hello2 {
    doFirst {
        print 'This is '
    }
    doLast {
        println 'a test!'
    }
}
```

Note: doFirst and doLast are the predefined actions. As the name suggests, doFirst code is executed before doLast. You don't need to use both together. You can use one or the other.

__ 2. Save the code.

__3. Execute hello2 task:

```
gradle -q hello2
```

It displays "This is a test!" message on the screen.

Note: "Hello World!" will be displayed in every test.

Part 5 - Define Task Dependency

In this part you will define task dependency. A task will make use of another task.

__1. In the Notepad window, which contains build.gradle code, append following code:

```
task welcome(dependsOn: hello1) {  
    doLast {  
        println "Welcome Bob!"  
    }  
}
```

Note: welcome task depends on hello1. It will execute hello1 then welcome task. In this case hello1 task already exists. In case if you define 'welcome' task before defining the dependent hello1 task, then use following syntax.

```
task welcome(dependsOn: 'hello1')
```

Notice here you have used quotes for defining the dependent task. This technique is also called Lazy dependsOn.

__2. Save the code.

__3. Execute welcome task:

```
gradle -q welcome
```

Notice it displays 'Hello Bob!'

Part 6 - Alternative technique for defining task dependency

In this part you will use a different technique for defining task dependency.

__ 1. In the Notepad window, which contains build.gradle code, append following code:

```
task hello {  
}  
  
hello.dependsOn hello1, hello2
```

Note: hello task depends on hello1 and hello2. It will execute hello1, hello2, then if hello task.

__ 2. Save the code.

__ 3. Execute hello task:

```
gradle -q hello
```

Notice it displays 'This is a test!'

Part 7 - Using Task Properties

In this part you will define task properties and reuse them in other tasks.

__ 1. In the Notepad window, which contains build.gradle code, append following code:

```
task myProperties {  
    ext.greeting = "Hello, "  
    ext.userName = "Bob"  
}
```

Note: Here you have created a custom task named myProperties and added 2 properties to it. Task name can be anything, but you must use ext object to add properties to the task.

__ 2. Append follow code to reuse the properties:

```
task useProperties {  
    doLast {  
        println myProperties.greeting + myProperties.userName  
    }  
}
```

Note: Here you have used the properties by specifying the task name.

__3. Save the code.

__4. Execute useProperties task:

```
gradle -q useProperties
```

Notice it displays 'Hello, Bob' message on the screen.

Part 8 - Create a method and reuse it in Gradle build script

In this part you will create a method and then reuse it in the Gradle script.

__1. In the Notepad window, which contains build.gradle code, append following code:

```
String toUpper(String userName) {  
    userName.toUpperCase()  
}
```

Note: This method takes string based input, converts it to upper case, and returns it.

__2. Append following code to reuse the method:

```
task callMethod {  
    doLast {  
        println myProperties.greeting + toUpper(myProperties.userName)  
    }  
}
```

Note: The custom toUpper method is called by the doLast action of custom task.

__3. Save the code.

__4. Execute the callMethod task:

```
gradle -q callMethod
```

Notice it displays 'Hello, BOB' message on the screen.

Part 9 - Defining default tasks

In this part you will set default tasks so they should execute even without specifying them as part of command-line arguments.

__1. In the Notepad window, which contains build.gradle code, append following code:

```
task defaultTask1 {
    doLast {
        println 'Default Task 1!'
    }
}

task defaultTask2 {
    doLast {
        println 'Default Task 2!'
    }
}
```

Note: Here you have defined 2 custom tasks. Next, you will set them as default tasks.

__2. In the beginning of the file, add following code:

```
defaultTasks 'defaultTask1', 'defaultTask2'
```

__3. Save the code.

__4. Execute the build script without specifying any task name:

```
gradle -q
```

Notice it displays 'Default Task 1! Default Task 2!' message on the screen.

Part 10 - Create Tasks Dynamically

In this part you will create tasks dynamically by using a loop.

__1. In the Notepad window, which contains build.gradle code, append following code:

```
3.times { i ->
    task "task$i" {
        doLast {
            println "Task #: $i"
        }
    }
}
```

Note: Here you are using groovy syntax to define a loop which runs 3 times. The current index location is stored in variable i. String interpolation is using to retrieve value of i and append it to "task".

__2. Append following code:

```
task allTasks {  
}  
  
allTasks.dependsOn task0, task1, task2
```

Note: Here you have created allTasks which aggregates task0, task1, and task2.

__3. Save the file.

__4. Execute allTasks:

```
gradle -q allTasks
```

Notice it displays 'Task #0, Task #1, Task #2' message on the screen.

Part 11 - Using list and .each loop

In this part you will create a list, use each loop to go over it, create directories, and delete directories.

__1. In the Notepad window, which contains build.gradle code, append following code:

```
task createDirs {  
    doLast {  
        ext.myDirList = ["dir1", "dir2"]  
        ext.myDirList.each() {  
            new File("${it}").mkdir()  
        }  
    }  
}
```

Note: You have used ext object to create a custom list of string then used each loop to iterate over the items and created directory for each item. `${it}` is a special variable which contains the current item being processed by the each loop.

__2. Save the file.

__3. Execute the task:

```
gradle -q createDirs
```

__ 4. Get directory list:

```
dir
```

Notice dir1 and dir2 are created.

__ 5. In the Notepad window, which contains build.gradle code, append following code:

```
task removeDirs {
    doLast {
        ext.myDirList = ["dir1", "dir2"]
        ext.myDirList.each() {
            new File("${it}").delete()
        }
    }
}
```

Note: In the code you are removing the directories which you created previously.

__ 6. Save the file.

__ 7. Execute the task:

```
gradle -q removeDirs
```

__ 8. Get the directory list:

```
dir
```

Notice dir1 and dir2 are removed.

__ 9. Close all.

Part 12 - Review

In this lab you utilized core features of groovy in build scripts

Lab 4 - Build Java Project, Management Package Dependencies, and Run Unit Tests with Gradle

Part 1 - Create directory structure for storing project files

In this part you will create directory structure for storing Java files. It's the same structure used by Maven as well.

- __ 1. Open a Command Prompt.
- __ 2. Switch to the \Workspace\labs\gradle directory:

```
cd C:\Workspace\labs\gradle
```

- __ 3. Create directory structure for storing Java code:

```
mkdir MyProject\src\main\java\hello
```

MyProject is the name of your project. It's the base directory you will use for storing your project files.

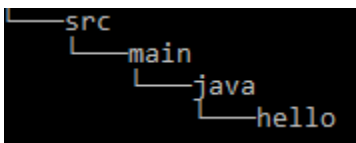
main and java directories must exist with same names. hello is custom Java package name.

- __ 4. Switch to MyProject directory:

```
cd MyProject
```

- __ 5. View directory tree:

```
tree
```



```
├── src
│   ├── main
│   │   ├── java
│   │   │   └── hello
```

Part 2 - Create a Java project

In this part you will write simple Java code. There will be 2 files. One will contain a custom class and second will contain the main function which make use of the custom class.

__1. Create Greeter.java by using Notepad: (Note: Click 'Yes', if prompted to do so)

```
notepad src\main\java\hello\Greeter.java
```

__2. Enter following code:

```
package hello;

public class Greeter {
    public String sayHello() {
        return "Hello world!";
    }
}
```

__3. Save the file and close Notepad window.

__4. Create HelloWorld.java file by using Notepad: (Note: Click 'Yes', if prompted to do so)

```
notepad src\main\java\hello\HelloWorld.java
```

__5. Enter following code:

```
package hello;

public class HelloWorld {
    public static void main(String[] args) {
        Greeter greeter = new Greeter();
        System.out.println(greeter.sayHello());
    }
}
```

__6. Save the file and close Notepad window.

Part 3 - Create Gradle build script

In this part you will create a Gradle build script.

__ 1. Run Gradle, without creating the build script, and get tasks list:

```
gradle tasks
```

Notice there is NO section labeled "Build tasks" containing build, clean, clean, ... tasks.

__ 2. Create build.gradle file: (Note: Click 'Yes', if prompted to do so)

```
notepad build.gradle
```

__ 3. Enter following code:

```
apply plugin: 'java'
```

You have included Java plugin which will make more tasks available to Gradle.

__ 4. Save and close the file.

__ 5. Get Gradle tasks list again:

```
gradle tasks
```

Notice there's a new section available with bunch of useful Java related tasks.

```
Build tasks
-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles test classes.
```

Part 4 - Build and Clean the Gradle generated files

In this part you will build the Java project with Gradle, run the generated compiled code, and clean the files.

__1. Build the project using Gradle:

```
gradle build
```

__2. View "build" directory tree:

```
tree build
```

```
C:\Workspace\labs\gradle\MyProject>tree build
Folder PATH listing
Volume serial number is CC19-8684
C:\WORKSPACE\LABS\GRADLE\MYPROJECT\BUILD
classes
├── java
│   └── main
│       └── hello
generated
├── sources
│   ├── annotationProcessor
│   │   └── java
│   │       └── main
│   ├── headers
│   │   └── java
│   │       └── main
libs
tmp
├── compileJava
└── jar
```

Here are some important directories:

* classes: compiled .class files are generated here.

* libs: jar file is stored here.

__3. Run the compiled class in the jar file:

```
java -cp build\libs\MyProject.jar hello.HelloWorld
```

Notice it displays "Hello World!" message on the screen.

__ 4. Clean the build generated files:

```
gradle clean
```

__ 5. Get directory list:

```
dir
```

Notice build directory is gone.

Part 5 - Dependency Management

In this part you will include package dependency and then manage it using the Gradle build script.

__ 1. Edit HelloWorld.java file:

```
notepad src\main\java\hello\HelloWorld.java
```

__ 2. Below "package hello;" add following code:

```
import org.joda.time.LocalDateTime;
```

Note: Although you can use native Java classes for obtaining date and time, but just so you can see dependency management you will utilize a 3rd party package, Joda, for obtaining the date and time.

__ 3. Above "Greeter greeter = new Greeter();" add following code:

```
LocalTime currentTime = new LocalDateTime();  
System.out.println("The current time is: " + currentTime);
```

__ 4. Save and close the file.

__ 5. Try to build the project with Gradle:

```
gradle build
```

Notice the build has failed. It's unable to find LocalDateTime class dependency.

__6. Edit the build script:

```
notepad build.gradle
```

__7. Append following code:

```
repositories {  
    mavenLocal()  
    mavenCentral()  
}
```

Note: Here you are adding local and online Maven Central repository (<http://search.maven.org>) from where the dependencies will be downloaded from. You can also use other repositories, such as Jcenter(<https://bintray.com/bintray/jcenter>).

Custom repositories can also be defined like this:

```
repositories {  
    maven {  
        url "http://repo.mycompany.com/maven2"  
    }  
}
```

__8. Below repository add following code to define source and target Java version:
(Note: this step is optional)

```
sourceCompatibility = 1.7  
targetCompatibility = 1.7
```

__9. Add following code to add Joda time 2.2 package dependency:

```
dependencies {  
    compile "joda-time:joda-time:2.2"  
}
```

__10. Save and close the file.

__11. Run Gradle build:

```
gradle build
```

Notice there are no errors this time.

Part 6 - Using Application Plugin

In this part you will use Application plugin. It makes more tasks available to Gradle. You can use it for executing your application.

__1. Edit the build script file:

```
notepad build.gradle
```

__2. Add following line at the beginning of the file:

```
apply plugin: 'application'
```

__3. Add following code at the end of the file:

```
mainClassName = "hello.HelloWorld"
```

Note: Here you have specified the main class which should be executed.

__4. Save and close the file.

__5. Execute the build script:

```
gradle build
```

__6. Run the application:

```
gradle -q run
```

Notice you have used the run task made available by Application plugin. Also notice it displays current date & time and Hello world! message on the screen.

Part 7 - Running unit tests with Gradle

In this part you will write a JUnit test and run it with Gradle.

__1. Create directory structure for storing unit test:

```
mkdir src\test\java\hello
```

__ 2. Create a unit test: (Note: Click 'Yes', if prompted to do so)

```
notepad src\test\java\hello\TestGreeting.java
```

__ 3. Enter following code:

```
package hello;

import org.junit.Assert;
import org.junit.Test;

import hello.Greeter;

public class TestGreeting {
    @Test
    public void testGreeter() {
        Greeter msg = new Greeter();
        Assert.assertEquals("Hello world!",
msg.sayHello());
    }
}
```

__ 4. Save and close the file.

__ 5. Run the test:

```
gradle test
```

Notice it displays error message that JUnit is not recognized. You need to add JUnit dependency in order to make it work. You will do it in the next step.

__ 6. Open build.gradle file:

```
notepad build.gradle
```

__ 7. In dependencies section, after compile "joda-time:joda-time:2.2", add following line:

```
testCompile "junit:junit:4.12"
```

__ 8. Save and close the file.

__ 9. Run test again:

```
gradle test
```

Notice test is executed successfully and no error is displayed.

__10. View test result:

```
notepad build\test-results\test\TEST-hello.TestGreeting.xml
```

Notice testGreeter test case is listed. It also shows time it took to run the test.

__11. Close the Notepad window.

__12. Close the command prompt:

```
exit
```

Part 8 - Review

In this lab you built a Java Project, managed dependencies, and ran unit tests with Gradle.

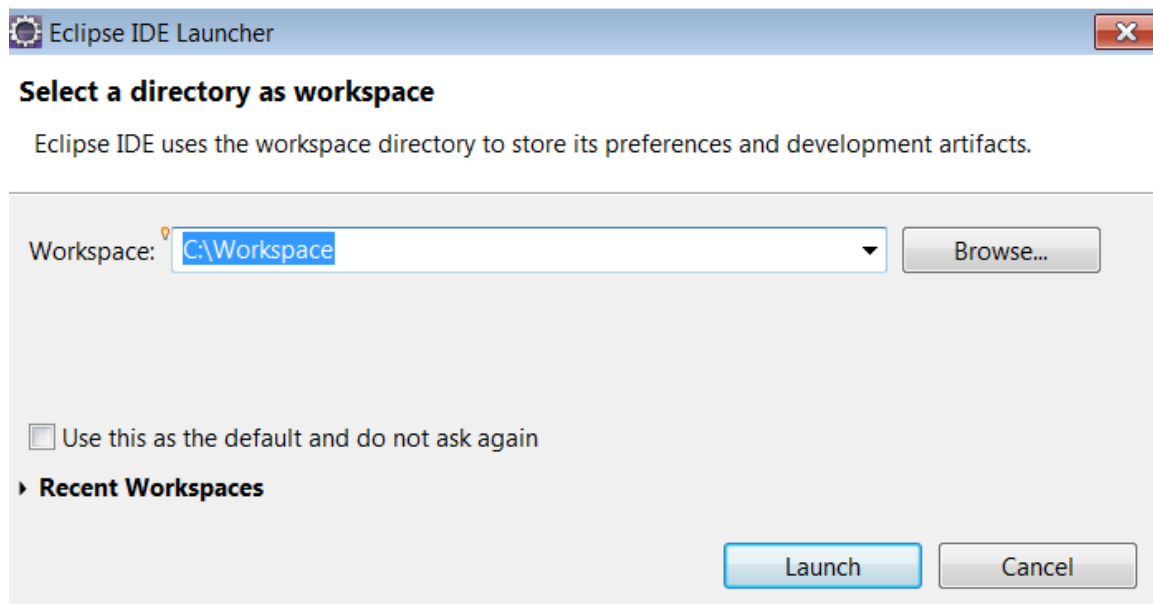
Lab 5 - A Simple RESTful API in Spring Boot

In this lab we're going to build a simple "Hello World" API using Spring Framework and Spring Boot. The API will implement a single resource, "/hello-message" that returns a JSON object that contains a greeting.

Part 1 - Update Gradle in Eclipse

We're going to start from scratch on this project, with an empty Gradle project, and add in the dependencies that will make a Spring Boot project with a core set of capabilities that we can use to implement our "Hello World" API.

- __1. Open Eclipse by navigating to **C:\Software\eclipse** and double-clicking on **eclipse.exe**
- __2. In the **Workspace Launcher** dialog, enter **C:\Workspace** in the **Workspace** field, and then click **Launch**.

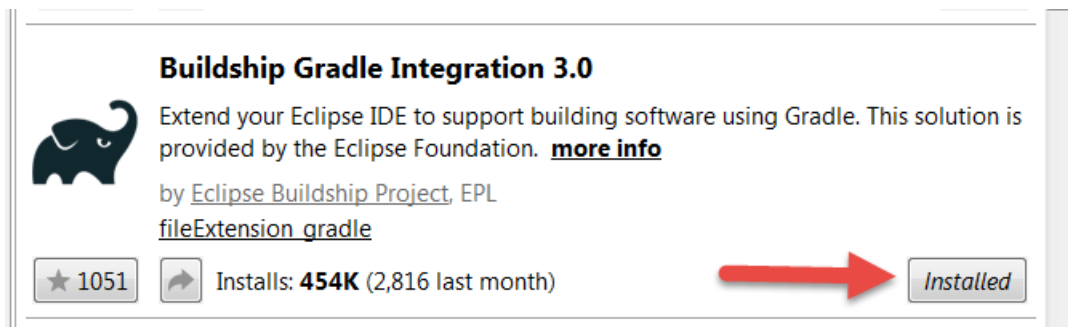


- __3. Close the **Welcome** panel.

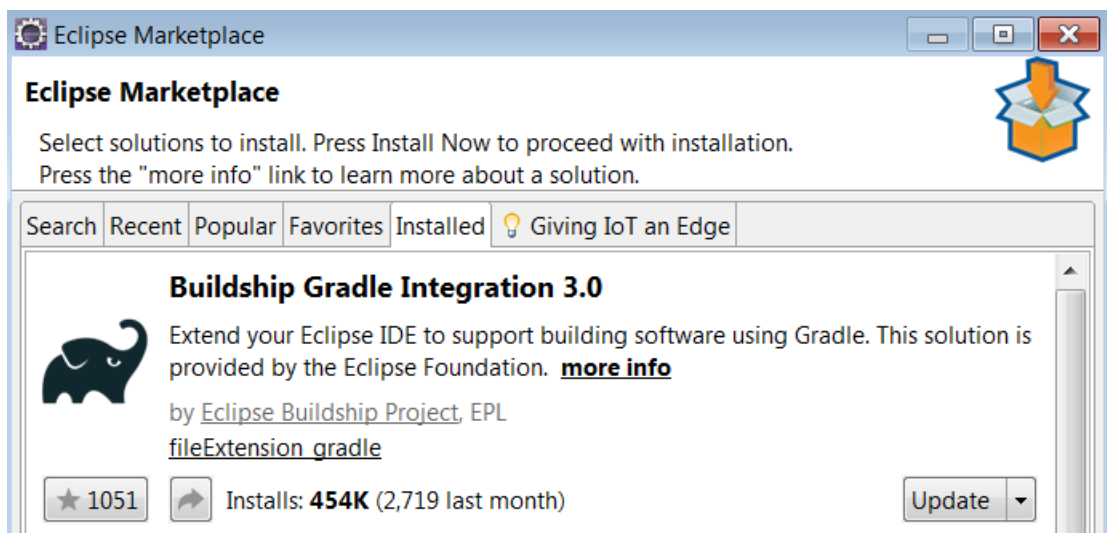
We need to update gradle.

- __4. From the menu, select **Help > Eclipse Marketplace**.
- __5. In the Find box, type **gradle** and hit enter.

__6. Locate **Buildship Gradle Integration 3.0** and click the *Installed* button.



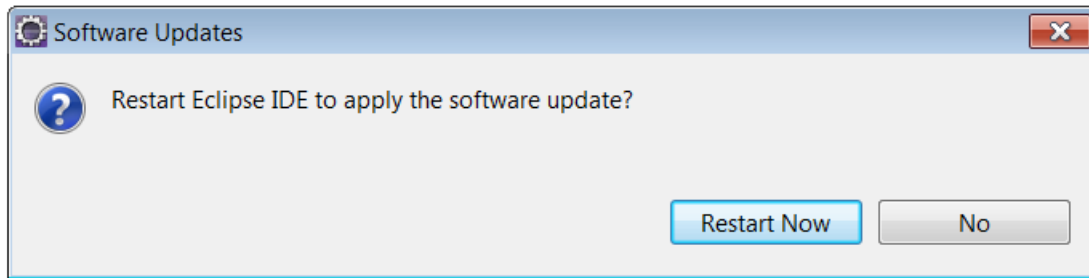
__7. This will take you to the **Installed** tab, locate **Buildship Gradle Integration 3.0** and click the **Update** button.



__8. Accept the license agreement and click **Finish**.

Wait for the installation to be completed.

__9. A dialog will open when the updates are installed, click **Restart Now**.



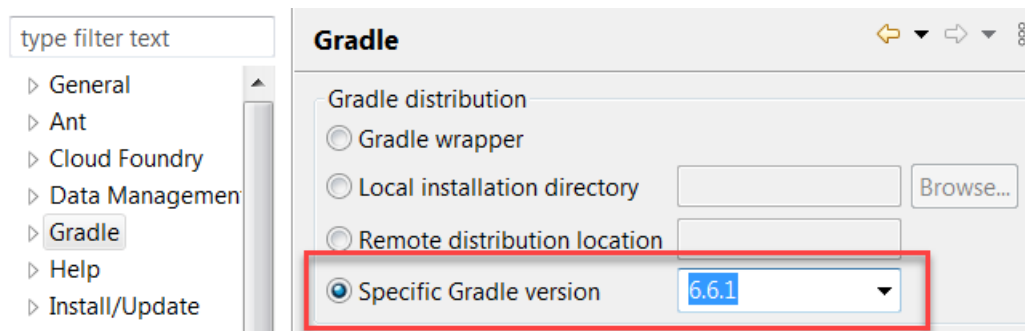
We are ready to work with Gradle.

Part 2 - Create a Gradle Project

__1. From the main menu, select **Windows** → **Preferences**.

__2. Select **Gradle** from the left menu.

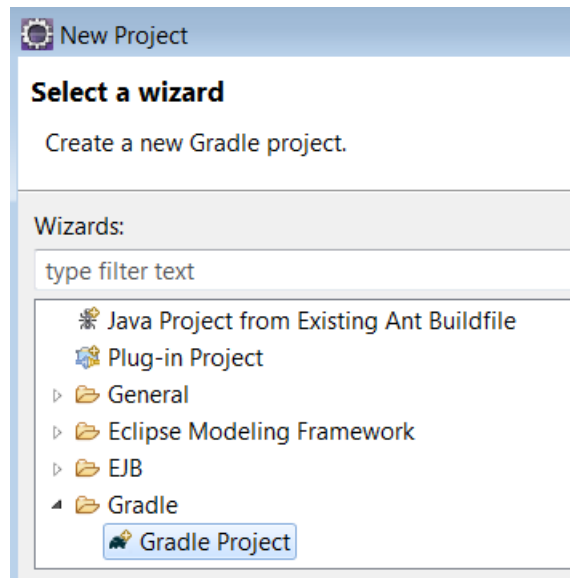
__3. Select **Specific Gradle version** and from the drop down select **6.6.1** version.



__4. Click **Apply and Close**.

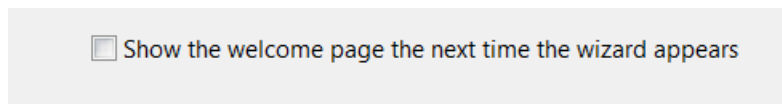
__5. From the main menu, select **File** → **New** → **Project...**

__6. In the **New Project** dialog, select **Gradle** → **Gradle Project**.

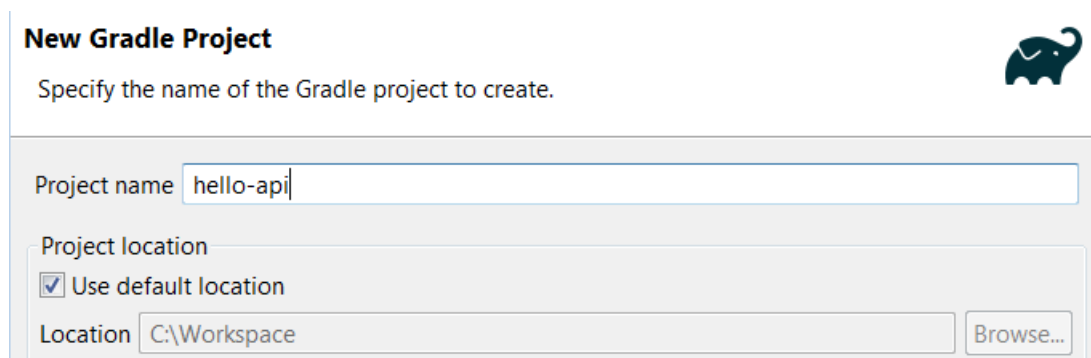


__7. Click **Next**.

__8. If it shows you a Welcome dialog, uncheck the checkbox **Show the welcome page the next time the wizard appears** and click **Next**.



__9. In the **New Gradle Project** dialog, enter **hello-api** as **Project name** and then click **Finish**.

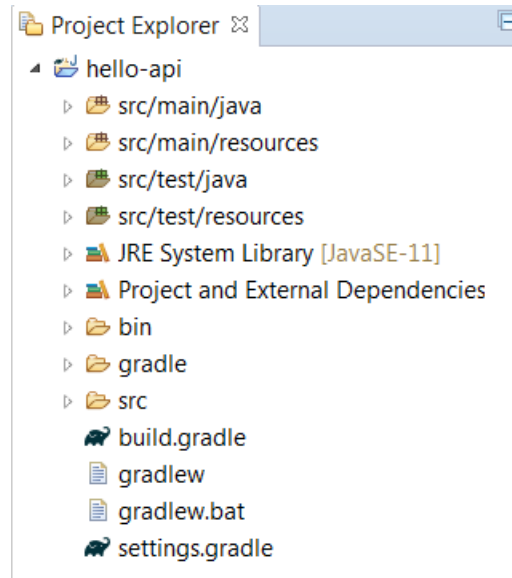


Wait for Eclipse to finish any background process.

Part 3 - Configure the Project as a Spring Boot Project

The steps so far have created a basic Gradle project. Now we'll add the dependencies to make a Spring Boot project.

___ 1. Expand the **hello-api** project in the **Project Explorer**.



___ 2. Expand **src/main/java** tree node.

___ 3. Delete everything that you see inside this folder.

___ 4. Expand **src/test/java** tree node.

___ 5. Delete everything that you see inside this folder.

___ 6. Double-click on **build.gradle** to open it.

___ 7. Select and delete all the existing content of **build.gradle** file.

__ 8. Enter the following content to the file:

```
apply plugin: 'java'
apply plugin: 'maven'

group = 'com.webage.spring.samples'
version = '0.0.1-SNAPSHOT'

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {

    mavenCentral()
}

dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-
starter-web', version: '2.1.8.RELEASE'
}
```

The entries above call out the Spring Boot dependencies.
--

__ 9. Save the file by pressing **Ctrl-S** or selecting **File → Save** from the main menu.

__ 10. Right-click the project and click **Gradle > Refresh Gradle Project**.

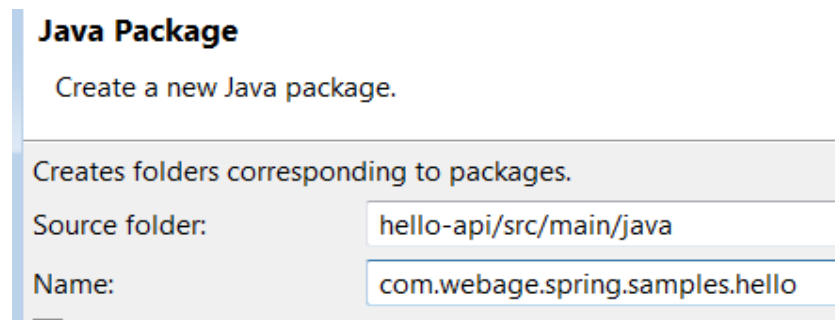
__ 11. Make sure the **Markers** tab doesn't show any error.

Part 4 - Create an Application Class

Spring Boot uses a 'Main' class to startup the application and hold the configuration for the application. In this section, we'll create the main class.

__ 1. In the **Project Explorer**, right-click on **src/main/java** and then select **New → Package**.

__2. Enter **com.webage.spring.samples.hello** in the **Name** field, and then click **Finish**.



Java Package
Create a new Java package.

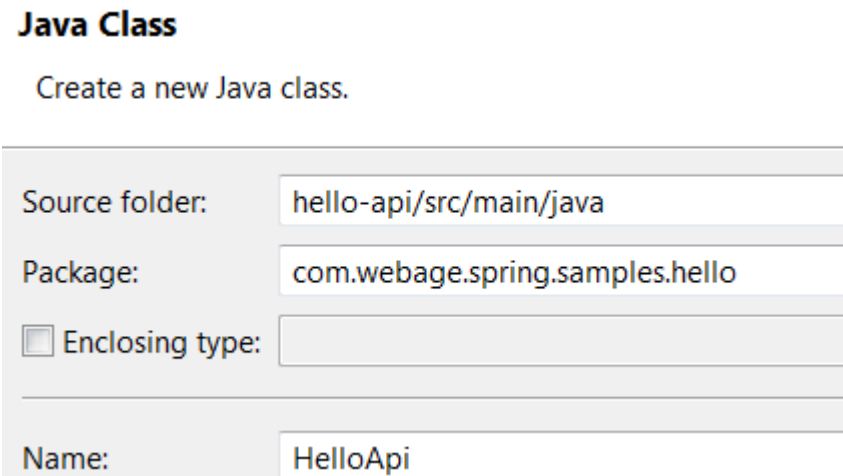
Creates folders corresponding to packages.

Source folder: hello-api/src/main/java

Name: com.webage.spring.samples.hello

__3. In the **Project Explorer**, right-click on the newly-created package and then select **New** → **Class**.

__4. In the **New Java Class** dialog, enter **HelloApi** as the **Name**, and then click **Finish**.



Java Class
Create a new Java class.

Source folder: hello-api/src/main/java

Package: com.webage.spring.samples.hello

☐ Enclosing type:

Name: HelloApi

__5. Add the **@SpringBootApplication** annotation to the class, so it appears like:

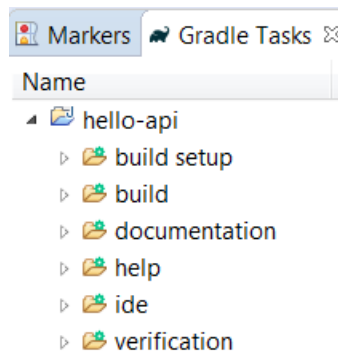
```
@SpringBootApplication  
public class HelloApi {
```

__6. Add the following 'main' method inside the class:

```
    public static void main(String[] args) {  
        SpringApplication.run(HelloApi.class, args);  
    }
```

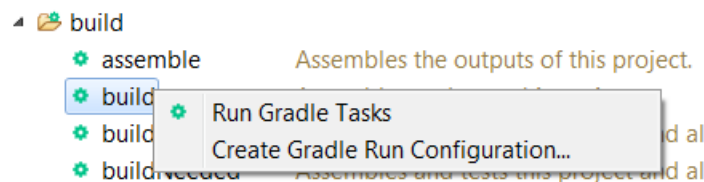
__7. The editor is probably showing errors due to missing 'import' statements. Press **Ctrl-Shift-O** to organize the imports.

- __ 8. Save the file.
- __ 9. Switch to **Gradle Tasks** pane and expand **hello-api** project.



Note: If Gradle Tasks pane isn't visible, in Eclipse, click Window > Show View > Other... > Gradle > Gradle Tasks and then click OK

- __ 10. Expand **build**, then right-click **build** and select **Run Gradle Tasks**.



- __ 11. Switch to **Console** pane and ensure the build was performed successfully.
- Now all we need to do is add a resource class and a response class.

Part 5 - Implement the RESTful Service

In this part of the lab, we will create a response class and a RESTful resource class.

- __ 1. In the **Project Explorer**, right-click on **src/main/java** and then select **New** → **Package**.
- __ 2. Enter **com.webage.spring.samples.hello.api** in the **Name** field, and then click **Finish**.

___3. In the **Project Explorer**, right-click on the newly-created package and then select **New → Class**.

___4. In the **New Java Class** dialog, enter **HelloResponse** as the **Name**, and then click **Finish**.

___5. Edit the body of the class so it reads as follows:

```
package com.webage.spring.samples.hello.api;

public class HelloResponse {
    String message;

    public HelloResponse(String message) {
        super();
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

___6. Save the file.

___7. In the **Project Explorer**, right-click on the **com.webage.spring.samples.hello.api** package and then select **New → Class**.

___8. In the **New Java Class** dialog, enter **HelloResource** as the **Name**, and then click **Finish**.

___9. Add the following 'getMessage' method inside the new class:

```
public HelloResponse getMessage() {
    return new HelloResponse("Hello!");
}
```

Spring Boot recognizes and configures the RESTful resource components by the annotations that we're about to place on the resource class that we just created.

__10. Add the '@RestController' annotation to HelloResource, so it looks like:

```
@RestController
public class HelloResource {
```

__11. Add the '@GetMapping' annotation to the 'getMessage' method, so it looks like:

```
@GetMapping("/hello-message")
public HelloResponse getMessage() {
```

__12. Organize the imports by pressing **Ctrl-Shift-O**.

__13. Save all files by pressing **Ctrl-Shift-S**.

__14. Switch to **Gradle Tasks** pane and expand **hello-api** project.

__15. Expand **build**, then right-click **build** and select **Run Gradle Tasks**.

__16. Switch to **Console** pane and ensure the build was performed successfully.

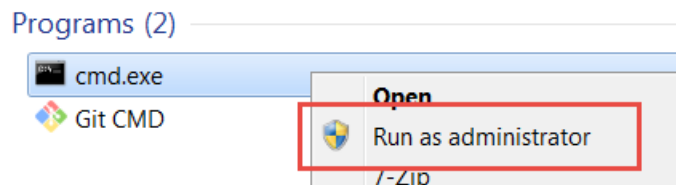
Note. If fails building try again and the second time should works.

The console should show a successful build.

Part 6 - Run and Test

Before running the application we need to make sure there are no other applications using port 8080.

__1. Open a command prompt window as an administrator.

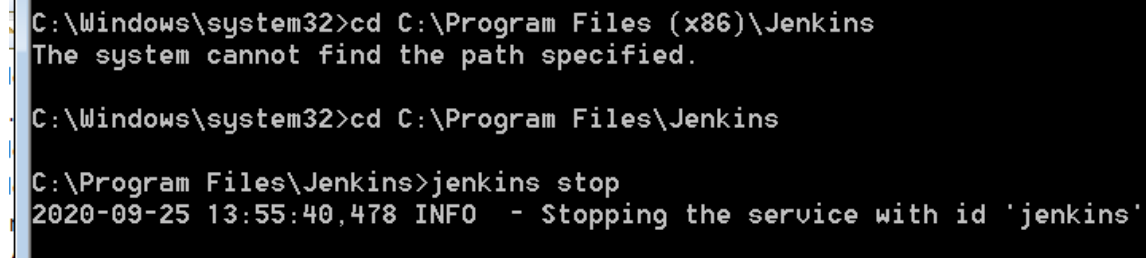


__2. If jenkins was installed then change to Jenkins installation folder (verify Jenkins path):

```
cd C:\Program Files\Jenkins
```

__3. Shut down Jenkins:

```
jenkins stop
```



```
C:\Windows\system32>cd C:\Program Files (x86)\Jenkins
The system cannot find the path specified.

C:\Windows\system32>cd C:\Program Files\Jenkins

C:\Program Files\Jenkins>jenkins stop
2020-09-25 13:55:40,478 INFO - Stopping the service with id 'jenkins'
```

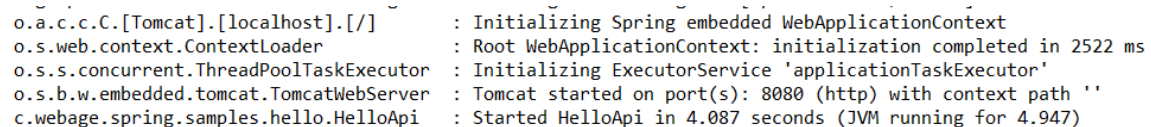
__4. Close the command prompt window.

That's all the components required to create a simple RESTful API with Spring Boot. Now let's fire it up and test it!

__5. In the **Project Explorer**, right-click on the **HelloApi** class and select **Run as** → **Java Application**.

__6. If the **Windows Security Alert** window pops up, click on **Allow Access**.

__7. Watch the **Console** panel. At the bottom of it, you should see a message indicating that the **HelloApi** program has started successfully:

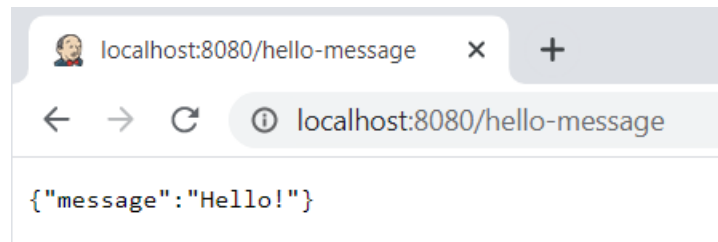


```
o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
o.s.web.context.ContextLoader       : Root WebApplicationContext: initialization completed in 2522 ms
o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
c.webage.spring.samples.hello.HelloApi : Started HelloApi in 4.087 seconds (JVM running for 4.947)
```

__8. Open the **Chrome** browser and enter the following URL in the location bar:

```
http://localhost:8080/hello-message
```

__9. You should see the following response:

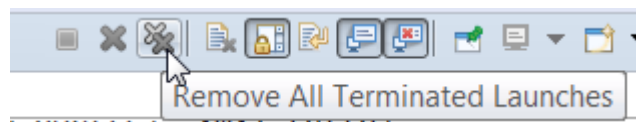


Notice that the response is in the form of a JSON object whose structure matches the 'HelloResponse' class contents.

__10. Close the browser.

__11. Click on the red 'Stop' button on the **Console** panel to stop the application.

__12. Click **Remove All Terminated Launches** a couple of times.



__13. Close all open files.

Part 7 - Review

In this lab, we setup a rudimentary Spring Boot application. There are a few things you should notice:

- There was really very little code and configuration required to implement the very simple RESTful API.
- The resulting application runs in a standalone configuration without requiring a web or application server. It opens its own port on 8080 (we'll see later how to configure this port to any value you want).
- Although the Eclipse IDE is providing some nice features, like type-ahead support and automatic imports, the only tool we really need is a build tool that does dependency management (e.g. Gradle).

Lab 6 - Overview of Common Microservices Runtime Issues

In this lab, we're going to review some of the common runtime errors of Java-based microservices (web services), including memory leaks, out-of-memory errors, etc.; these errors can also occur in systems written in other programming languages as well as stand-alone applications.

To generate runtime errors, we will use a simple microservice with a RESTful API written using the *Java Spark* microservice framework (<https://sparkjava.com/>) – make sure you do not confuse this Java framework with the Apache Spark scalable Big Data processing platform! The *Java Spark* framework uses the embedded Jetty web server (<https://www.eclipse.org/jetty/>) that only needs your application packaged as a JAR file – our application is packaged as **lab_server-1.0.jar**. In this way, you are not forced to create Java WAR or EAR files – more complex deployment artifacts – for packaging and deploying your Java web service code. This approach for building and deploying microservices is also used in the Spring Boot project (<https://spring.io/>).

In this lab, for monitoring runtime issues, we will be using the VisualVM tool (<https://visualvm.github.io/download.html>) – a free Java profiler and performance analysis tool that has a number of proprietary counterparts normally bundled with enterprise applications servers, like those offered by IBM and Oracle.

Part 1 - Logging

For logging the runtime events generated by our microservice, we will use *log4j* (<https://logging.apache.org/log4j/2.x/>), a popular enterprise-grade Java-based logging system.

Some of the important properties of a logging system include:

- ✓ Performance that can be achieved without impacting the logging application (your web server or microservice)
 - log4j version 2 (log4j 2) is a garbage-free system in stand-alone applications and low-GC in web applications
- ✓ Integration with other logging system APIs
- ✓ Automatic reloading of configurations at runtime upon modification
- ✓ Support for log message filtering based on context, markers, regular expressions, and other components in the log event stream

Log4j offers developers configurable logging levels (in order of severity): TRACE, DEBUG, INFO, WARN, ERROR, FATAL, that allow developers to log only messages at the set level and above (e.g. setting the current logging level at the INFO level, will stop logging all the messages below it: TRACE and DEBUG, only logging messages at the INFO level and up: WARN, ERROR, and FATAL).

The log4j 2 configuration file we will use is shown below for your reference:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n"/>
    </Console>
  </Appenders>
  <Loggers>
    <Root level="debug">
      <AppenderRef ref="Console"/>
    </Root>
  </Loggers>
</Configuration>
```

The configuration file sets the **root** level at **debug** (can be overwritten in specific loggers); the output target for logging messages is set to `SYSTEM_OUT` (console); usually, logging is done to a local file, or a remote system over a TCP/IP link.

We have configured the logging messages to follow this pattern:

```
<PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} -
%msg%n"/>
```

which sets

- the timestamp of the logging message,
- the thread name (%t),
- the logging level,
- the logger name, and
- the logging message text terminated with a line-feed symbol (%n).

Here is a sample logging message:

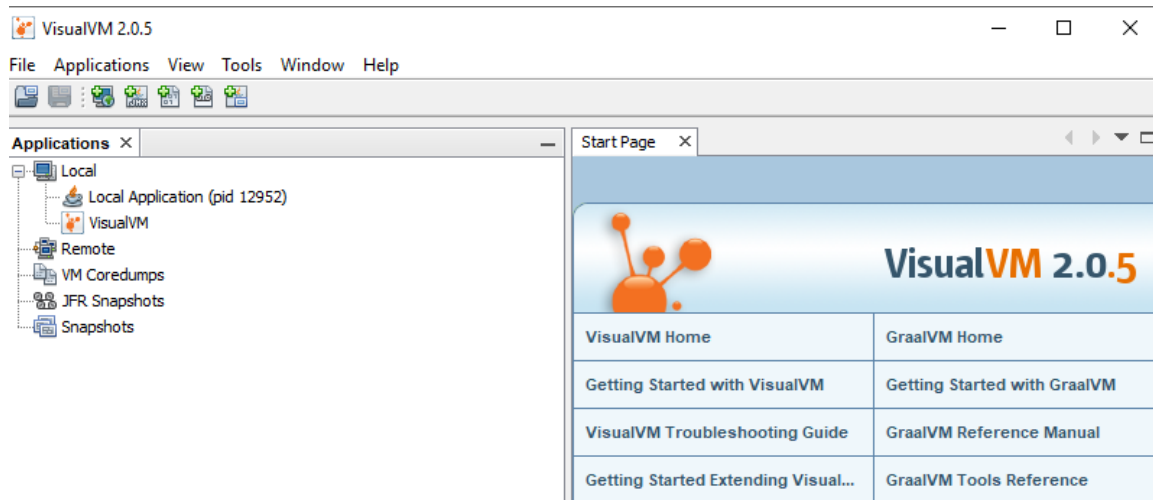
```
16:32:00.939 [qtp534947286-19] INFO  webage.LabServer - Received a
request to stop...
```

Log files are usually analyzed in specialized systems like Splunk and Datadog.

Part 2 - Setting Up the Working Environment

- ___ 1. Locate the `C:\LabFiles\VisualVM_2.05\visualvm_205.zip` archive and unzip it.
- ___ 2. In the unzipped archive, locate and run `visualvm\bin\visualvm.exe` file (if you have any issues starting the application, read **VisualVM README.txt** in the `C:\LabFiles\VisualVM_2.05\` directory).
- ___ 3. Click **I accept** in the License agreement window.

After a successful application start-up, you should see the following Visual VM UI:



- ___ 4. Open a Command Prompt, change directory to `C:\LabFiles\`, and enter the following command:

```
title CURL
```

We will be referring to this terminal window as the *CURL* terminal.

In this terminal, we will be running the `curl.exe` tool (*cURL*) to submit web requests to our application in order to generate inside it runtime exceptions that will be logged to console.

Note: There are 4 operating modes that our application can enter that you can control with the web requests submitted by curl:

- ✓ Default mode – invoked by a GET web request submitted by `curl.exe` – the app sends back the current system time in response
- ✓ Simulation of a regular service load that consists of 30 consecutive requests for large chunks of the heap as coded in the `regularProcessing()` function; this mode is entered into by the app in response to the `task=1` query parameter of the curl web request

- ✓ Simulation of a *java.lang.StackOverflowError* error as coded in the *causeStackOverflow()* function; this mode is entered by the app in response to the **task=2** query parameter of the curl web request
- ✓ Simulation of a *java.lang.OutOfMemoryError* error as coded in the *generateOOM()* function; this mode is entered by the app in response to the **task=3** query parameter of the curl web request

__5. Open a new Command Prompt, change directory to **C:\LabFiles**, and enter the following command:

```
title LabServer
```

We will be referring to this terminal window as the *LabServer* terminal.

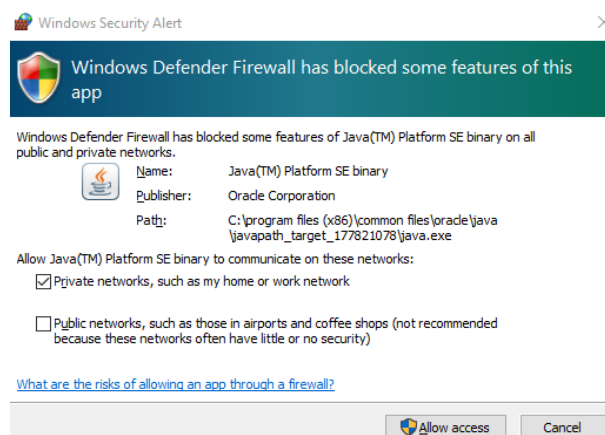
Our RESTful service for generating runtime errors that we are going to use in the subsequent lab parts is bundled as the **lab_server-1.0.jar** file; the app's code is shown at the end of this lab for your reference.

Part 3 - Run the App

__1. In the *LabServer* terminal, enter the following command to bootstrap the app using the embedded Jetty web server:

```
java -cp lab_server-1.0.jar -Xmx128m webage.LabServer 8080
```

If and when you see a *Windows Security Alert* dialog like the one shown below, click **Allow access** or provide a similar response to proceed.



Note: The command-line arguments used in the above command are as follows:

- cp** - the JVM CLI flag for setting up the CLASSPATH system variable pointing to the location of the application's Java classes to be loaded and used
- Xmx128m** - limiting the Java process' heap at 128 MB; you could use this configuration option to request more memory from the underlying OS so you can gain more time in case of a memory leak
- webage.LabServer** - the fully qualified name of the app
- 8080** - the port for the Jetty web server to start listening on

When the app starts, you should see the following messages printed in the terminal:

```
WARNING: sun.reflect.Reflection.getCallerClass is not supported. This will impact
performance.
10:23:04.460 [main] INFO  webage.LabServer - About to begin processing the command line
parameters.
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
10:23:04.491 [main] INFO  webage.LabServer - The server is listening on port: 8080
10:23:04.506 [main] INFO  webage.LabServer - All Spark routers loaded ...
```

There are two logging channels printing messages to console:

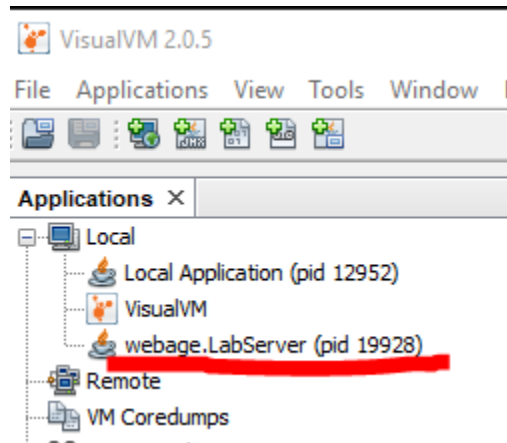
- The *Spark* system's warning and diagnostic messages, which you can ignore
- Our app's messages, formatted as per log4j.xml

Messages at the INFO level are logged using the log4j logger object's *info* method:

```
logger.info("your message");
```

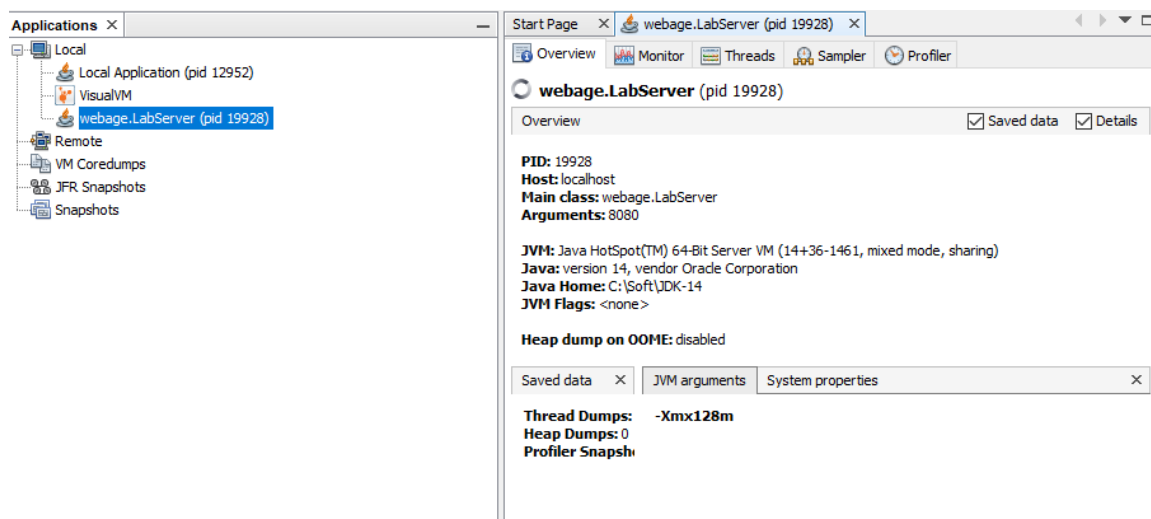
2. Arrange the *VisualVM* window, the *CURL* and *LabServer* terminals side-by-side without overlapping on the desktop so you can easily switch between them, enter commands, and observe the logging messages printed in the console.

3. In the *VisualVM* app, you should see that the *webage.LabServer* Java process has been identified and registered (your process id will be different):

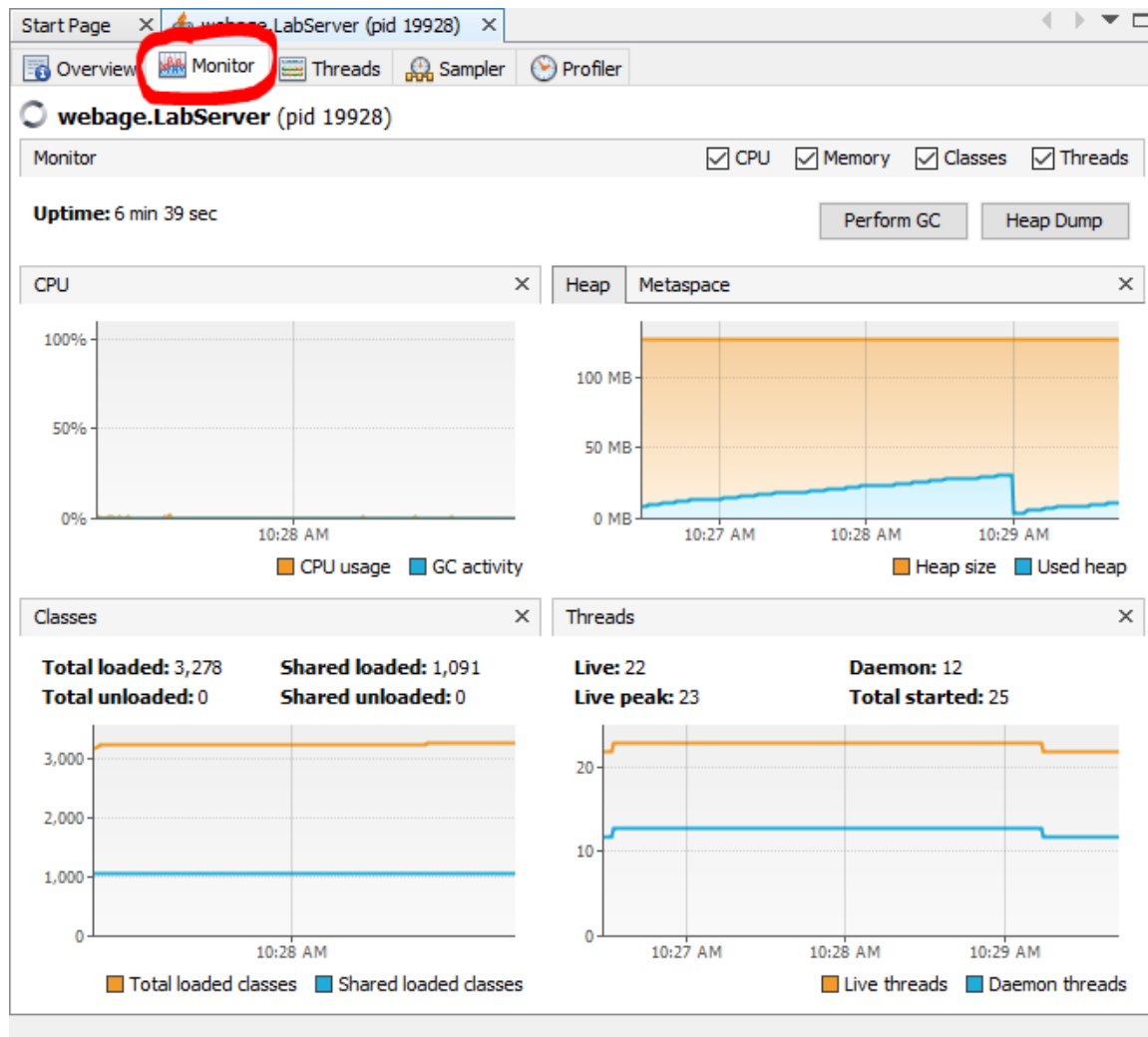


4. Double click **webage.LabServer**

You should see the following pane open on the right-hand side of the *VisualVM* UI:



__5. Click the **Monitor** tab; the process resource utilization monitoring pane should open.



For the most part in this lab, we will be monitoring the heap runtime metrics (the available heap size and heap utilization).

__6. In the *CURL* terminal, enter the following command twice:

```
curl http://localhost:8080/do
```

You should see the following output in the console (your timestamps will differ):

```
The current server time is 10:34:11.061233; previous time: N/A
The current server time is 10:35:10.179490400; previous time: 10:34:11.061233
```

__7. Enter the following command in the *CURL* terminal:

```
curl http://localhost:8080/do?task=1
```

This web request puts the app in its regular processing operating mode.

The web service will start simulating load for about 1 minute.

__8. Switch to the *LabServer* terminal.

You should see the following logging messages there:

```
10:34:11.045 [qtp473264680-24] INFO webage.LabServer - Received a request for: /do
10:35:10.179 [qtp473264680-19] INFO webage.LabServer - Received a request for: /do
10:36:45.404 [qtp473264680-21] INFO webage.LabServer - Received a request for: /do
10:36:45.406 [qtp473264680-21] INFO webage.LabServer - Received the query parameter
'task': 1
10:36:45.406 [qtp473264680-21] INFO webage.LabServer - Processing task #1 ...
10:36:45.407 [qtp473264680-21] INFO webage.LabServer - Simulating regular processing
10:36:45.407 [qtp473264680-21] INFO webage.LabServer - Iterations left: 30
10:36:45.407 [qtp473264680-21] INFO webage.LabServer - About to request 42120076 bytes
off of free memory.
10:36:47.487 [qtp473264680-21] INFO webage.LabServer - Iterations left: 29
10:36:47.488 [qtp473264680-21] INFO webage.LabServer - About to request 68937612 bytes
off of free memory.
10:36:49.632 [qtp473264680-21] INFO webage.LabServer - Iterations left: 28
10:36:49.632 [qtp473264680-21] INFO webage.LabServer - About to request 41189280 bytes
off of free memory.
10:36:51.730 [qtp473264680-21] INFO webage.LabServer - Iterations left: 27
...
10:37:46.957 [qtp473264680-21] INFO webage.LabServer - Iterations left: 0
10:37:46.957 [qtp473264680-21] INFO webage.LabServer - About to request 38669164 bytes
off of free memory.
```

The related code for load simulation in the *regularProcessing()* function is shown below for your reference:

```
while (iterations-- > 0) {
    int freeMemory = (int) (Runtime.getRuntime().freeMemory() * 0.8);
    logger.info("Iterations left: " + iterations);
    logger.info("About to request " + freeMemory + " bytes off of free
memory.");

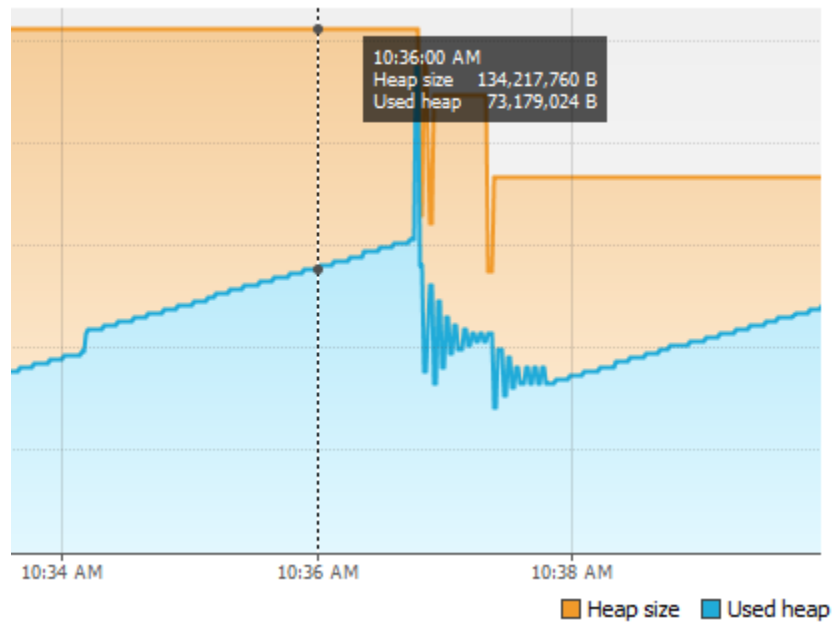
    byte[] memoryGrabber = new byte[freeMemory];
    holder.add(memoryGrabber);
    Thread.sleep(sleep_ms);
    holder = new ArrayList<byte[]>();
    Runtime.getRuntime().gc();
}
```

The process requests **80%** of free memory on the heap, and then, after a two-second delay, releases the memory and calls the garbage collector to clean the heap (GC may be delayed in executing its task as the *Runtime.getRuntime().gc()* command is simply a suggestion to the JVM that can be ignored by it at that moment). The cycle is repeated 30 times.

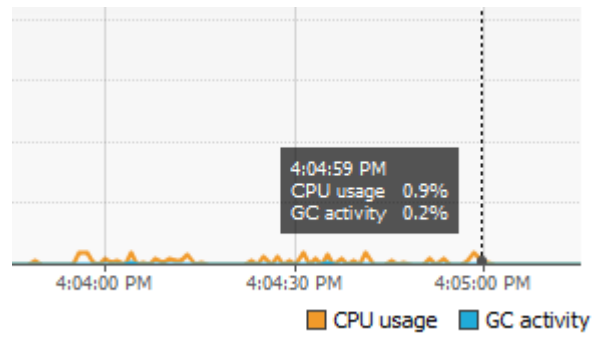
The app survives these aggressive memory allocation requests as the JVM can find enough memory by making more room on the heap as needed.

__9. Switch to the *VisualVM* window, and observe the heap utilization dynamic.

Mind you that the dynamic captured in your heap utilization graph may look differently.



In the *CPU Usage* pane, you can also observe small spikes in the GC activity which contribute to the overall CPU usage:



High CPU usage levels in your app may, in some situations, indicate intensive GC-ing occurring in the JVM, which, in turn, may be caused by inefficient code. Such situations can be dealt with by refactoring the application code to use static factory methods instead of constructors for object creation, using the Singleton design pattern in a thread-safe manner, single-element enum types, etc. You may also need to review and change the current GC policy (using the JVM CLI) to better match the runtime profile and the needs of your Java application.

Part 4 - Generate a StackOverflow Error

Now let's simulate a situation resulting in a Stack Overflow Java runtime error.

__1. Switch to the *CURL* terminal and enter the following command:

```
curl http://localhost:8080/do?task=2
```

It will take about 20 seconds or so to generate a *java.lang.StackOverflowError*. that you can see in the *LabServer* terminal:

```
11:05:28.927 [qtp473264680-20] INFO webage.LabServer - Received a request for: /do
11:05:28.927 [qtp473264680-20] INFO webage.LabServer - Received the query parameter
'task': 2
11:05:28.927 [qtp473264680-20] INFO webage.LabServer - Processing task #2 ...
11:05:45.342 [qtp473264680-20] ERROR webage.LabServer - java.lang.StackOverflowError
```

As you can see, the message is logged using the ERROR logging category (level), and it is done in this statement:

```
logger.error(e);
```

Note that this runtime error did not kill the process, which proceeded unscathed to generating the server system time (that you can see in the *CURL* terminal); the heap consumption, as reported in *VisualVM*, has not been affected.

The negative side-effect of a *StackOverflowError* is that the whole stack-frame lineage will be busted, killing all the in-flight computations.

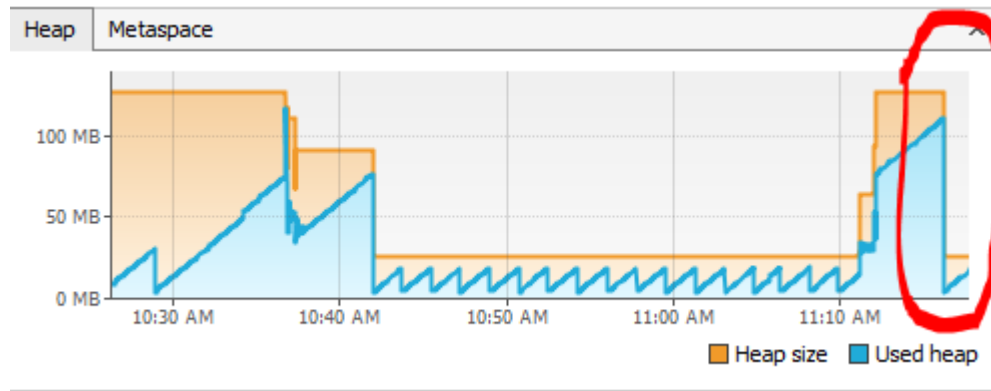
StackOverflowError:

A *StackOverflowError* is a runtime error in Java. It is thrown when the amount of call stack memory allocated by the JVM for a thread of execution has been exceeded. A common cause for a *StackOverflowError* is excessively deep or infinite recursion.

One way to tackle this type of error is to increase the thread stack size using this JVM flag:: *-Xss<larger stack size>*. Alternatively, code rewrite might be required that may involve substituting "elegant" recursion idioms with simple *for* or *while* cycles.

Interestingly, if you submit a curl request for regular processing operating mode with the `task=1` query parameter, you may see that processing may still be done on the same thread (the thread's name follows the timestamp in log messages) that hit the `StackOverflowError` exception.

2. By watching the *VisualVM* UI, wait until the system "cools down" and the heap's "speculative growth" has stopped and the heap is reclaimed - it may take a couple of minutes or so. When it happens, there should be a sharp drop in the used heap size (the blue graph) followed by the compaction of the heap itself (the orange-colored graph).



It is quite possible that this re-cycling may not happen while you are waiting, in which case, do not waste your time, and proceed to the next lab part.

Part 5 - Generate an Out-of-Memory Error

Now, let's create a condition in our application that will cause a memory leak resulting in `java.lang.OutOfMemoryError` (the OOM error).

According to Java Documentation, "One common indication of a memory leak is the `java.lang.OutOfMemoryError` exception. Usually, this error is thrown when there is insufficient space to allocate an object in the Java heap. In this case, The garbage collector cannot make space available to accommodate a new object, and the heap cannot be expanded further. Also, this error may be thrown when there is insufficient native memory to support the loading of a Java class. In a rare instance, a `java.lang.OutOfMemoryError` may be thrown when an excessive amount of time is being spent doing garbage collection and little memory is being freed. The `java.lang.OutOfMemoryError` exception can also be thrown by native library code when a native allocation cannot be satisfied (for example, if swap space is low)."

<https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html>

__1. Switch to the *CURL* terminal, and enter the following command:

```
curl http://localhost:8080/do?task=3
```

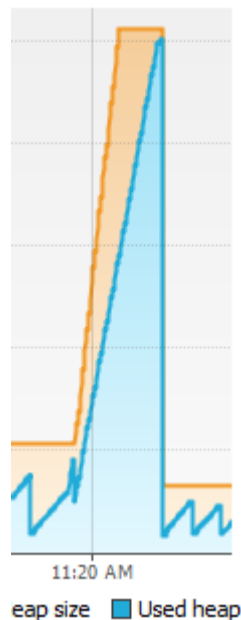
__2. In the *LabServer* terminal, you should see the following log messages:

```
11:19:12.701 [qtp473264680-25] INFO webage.LabServer - Generating an OOM error in a few iterations ....
11:19:12.701 [qtp473264680-25] INFO webage.LabServer - Take #1: About to request 1076030 bytes off of free memory.
11:19:15.711 [qtp473264680-25] INFO webage.LabServer - Take #2: About to request 1076030 bytes off of free memory
...
```

__3. In the *VisualVM* UI, you should see the gradual growth in the used heap sizes that pushes the heap size up. Then, at a certain point, where the JVM can no longer accommodate requests for more memory on the heap, an OOM occurs – the arrival of the error can be seen in the *LabServer* terminal:

```
....
11:22:10.208 [qtp473264680-25] INFO webage.LabServer - Take #60: About to request 1076030 bytes off of free memory.
11:22:13.217 [qtp473264680-25] INFO webage.LabServer - Take #61: About to request 1076030 bytes off of free memory.
11:22:13.264 [qtp473264680-25] ERROR webage.LabServer - java.lang.OutOfMemoryError: Java heap space
```

The *VisualVM* UI reports this fact with a sharp drop in the used and allocated heaps.



Our app is still being able to generate the server response without losing the information about the previous server time stashed in the static segment of the JVM before the OOM error:

```
#curl http://localhost:8080/do?task=1
The current server time is 11:12:08.867714600; previous time: 11:05:45.342379600
#curl http://localhost:8080/do?task=3
The current server time is 11:22:13.264326600; previous time: 11:12:08.867714600
```

At the moment the OOM error occurred, the heap is compacted and your work is lost (in older JVMs that could also mean the death of the application server requiring its restart), but modern web server engines and JVMs can survive OOMs by retaining the loaded class definitions and native code components outside of the main heap so that your microservice can continue from a clean slate without restarting; you can confirm this fact with our regular processing command:

```
curl http://localhost:8080/do?task=1
```

What we have just observed was a gradual (but rather rapid) memory leak that exhausted all the available heap that we capped by 128 MB with this command flag: **-Xmx128m**

Note: In some situations, you can delay the negative effect of a memory leak by increasing the available heap using the `-Xmx<larger size>` JVM CLI command flag. Another, rather obtrusive mechanism for heap correction is to force service re-cycling when the heap utilization factor is getting dangerously high. To minimize the loss of the in-flight transactions, design your app to be able to enter into a quiescing mode where it would stop accepting new requests for service waiting for all the in-flight transactions to complete and the associated resources to drain before you can safely restart the service. This operation could greatly benefit from making your service stateless, externalizing the app's state into another system (a database or some sort of caching solution).

Part 6 - Terminate the Service

We are done with reviewing common runtime problems your microservice may face.

__1. In the *CURL* terminal, enter the following command to terminate the LabServer process:

```
curl -X POST http://localhost:8080/stop
```

You should see the following output:

```
curl: (52) Empty reply from server
```

In the *LabServer* terminal, you should see the following message:

```
12:45:52.648 [qtp473264680-24] INFO  webage.LabServer - Received a
request to stop...
```

After about a minute, the process should return the prompt.

__2. Close all the open windows and terminals.

Part 7 - Comparing Java with Node.js (Optional)

Java is an enterprise platform used in many mission-critical applications. There is some criticism of Java for its rather ceremonial development and deployment cycles, which involves the following activities:

- compilation of source code to bytecode (the *.class* files)
- packaging the *.class* files into JAR files, or WAR / EAR application server web container deployment artifacts
- deployment of the deployment artifacts on the web/application server
 - requires a restart; restarting a JVM has associated latencies

Node.js development and deployment cycles are much simpler compared to the above Java cycle. Node applications are essentially JavaScript text files (Java deployment artifacts are in binary form) that can be edited for changes in-place followed by the Node executable restart (this practice should only be used in the development and in cases of severe errors in production that must be fixed ASAP – in regular production environments the established CI/CD cycles must be followed without exceptions). Using TypeScript to add type safety to JavaScript code further improves the position of Node-based apps in the field. So, overall, Node-based application development results in much faster development/deployment cycles; this agile development is also more in line with the philosophy of microservices.

Some Java web frameworks, including *Java Spark*, are trying to simplify the awkward plumbing code of a standard web application by leveraging, for example, lambda expressions added in Java 8. This approach makes the code look like that of a Node app. Below is a sample code of a basic Node-based app that uses the Express web library. Compare it with the Java code listed in the next lab part.

```
const express = require('express')
const app = express()

const hostname = '127.0.0.1';
const default_port = 4567;

// Reading the first app command-line argument
var port = process.argv[2];
```

```

if (typeof port == 'undefined'){
    port = default_port;
}

app.get('/do', (req, res) => {
    var task = req.query.task;
    if (typeof task !== 'undefined') {
        console.log("Processing task query parameter: " + task);
    }

    var now = new Date();
    console.log('The current server date/time to be returned to the client: ' + now);
    res.send(now);
})

app.post('/stop', (req, res) => {
    console.log("Received a request to stop... ");
    process.exit(1);
    res.end();
})

const server = app.listen(port, hostname, () => {
    console.log(`Server app listening at http://${hostname}:${port}/`);
})

```

Part 8 - The LabServer App Code (For Your Reference)

```

package webage;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

//import java.time.LocalDateTime;
import java.time.LocalDateTime;
import java.util.ArrayList;

import static spark.Spark.*;

public class LabServer {
    private static Logger logger = LogManager.getLogger(LabServer.class.getName());

    private static String prev_time = "N/A";

    public static void main(String[] args) throws InterruptedException {
        try {
            logger.info("About to begin processing the command line parameters.");

            int port = 4567; // The default port is 4567
            if (args.length != 0) {
                port = Integer.parseInt(args[0]);
                // To find out which process uses a TCP port on Windows(R) :
                // netstat -ano -p tcp | grep 4567
                // So you can kill it in Task Manager, if needed
            }

            port(port);

            logger.info("The server is listening on port: " + port);
        }
    }
}

```

```

get("/do", (req, res) -> {
    logger.info("Received a request for: " + req.pathInfo());
    String task = req.queryParams("task");

    if (task != null) {
        logger.info("Received the query parameter 'task': " + task);
        handleTasks(task);
    }

//
    LocalDateTime now = LocalDateTime.now();
    LocalTime now = LocalTime.now();
    String client_msg = "The current server time is " + now + "; previous
time: " + prev_time;
    prev_time = now.toString();
    return client_msg;
});

post("/stop", (req, res) -> {
    logger.info("Received a request to stop... ");
    Thread.sleep(1000L); // let the message get printed
    stop();
    System.exit(0);
    return "Exiting the application ..."; // The signature requires a return
                                         // value even though now
                                         // the process is dead
                                         // ...
});

    logger.info("All Spark routers loaded ...");

} catch (Exception e) {
    logger.error(e);
    Thread.sleep(1000);
    usage();
} finally {
}

}

private static void handleTasks(String task) throws InterruptedException {

    /*
    * Task Types:
    * -----
    * 1 - Demonstrate regular processing memory utilization
    * 2 - Generate java.lang.StackOverflowError
    * 3 - Simulate a memory leak that leads to java.lang.OutOfMemoryError
    */
    try {
        int task_id = Integer.parseInt(task);
        switch (task_id) {
            case 1:
                logger.info("Processing task #1 ...");
                regularProcessing();
                break;
            case 2:
                logger.info("Processing task #2 ...");
                causeStackOverflow();
                break;
            case 3:
                logger.info("Processing task #3 ...");
                generateOOM();
                break;
        }
    }
}

```



```

        default:
            logger.error("Expected parameters are 1,2, or 3");
        }
    } catch (Throwable e) {
        logger.error(e);
    }
}

private static void causeStackOverflow() {
    try {
        Thread.sleep(2L);
        causeStackOverflow();
    } catch (Throwable e) {
        logger.error(e.toString());
    }
}

private static void regularProcessing() throws InterruptedException {
    logger.info("Simulating regular processing");

    ArrayList<byte[]> holder = new ArrayList<byte[]>();
    long sleep_ms = 2000L;
    int one_minute = 60 * 1000;
    int iterations = one_minute / (int) sleep_ms + 1;

    try {
        while (iterations-- > 0) {
            int freeMemory = (int) (Runtime.getRuntime().freeMemory() * 0.8);
            logger.info("Iterations left: " + iterations);
            logger.info("About to request " + freeMemory + " bytes off of free
memory.");

            byte[] memoryGrabber = new byte[freeMemory];
            holder.add(memoryGrabber);
            Thread.sleep(sleep_ms);
            holder = new ArrayList<byte[]>();
            Runtime.getRuntime().gc();
        }
    } catch (Throwable e) {
        logger.error(e);
    }
}

private static void generateOOM() throws InterruptedException {
    logger.info("Max memory available to the process: " +
Runtime.getRuntime().maxMemory() + " bytes.");
    logger.info("Generating an OOM error in a few iterations ....");
    ArrayList<byte[]> holder = new ArrayList<byte[]>();
    long sleep_ms = 3000L;

    int freeMemory = (int) Runtime.getRuntime().freeMemory();
    int memoryToGrab = freeMemory / 10;

    int iter_count = 1;
    try {
        while (true) {

            logger.info("Take #" + iter_count + ": About to request " + memoryToGrab
+ " bytes off of free memory.");
            iter_count++;

```

```

byte[] memoryGrabber = new byte[memoryToGrab];
        holder.add(memoryGrabber);
        Thread.sleep(sleep_ms);
    }
    } catch (Throwable e) {
        logger.error(e);
    }
}

static void usage() {
    logger.warn("Usage: java -cp lab_server-1.0.jar webage.LabServer port_num");
}
}

```

Lab 7 - Configure Tools in Jenkins

In this lab you will verify that Jenkins Continuous Integration is already installed and you will configure it.

At the end of this lab you will be able to:

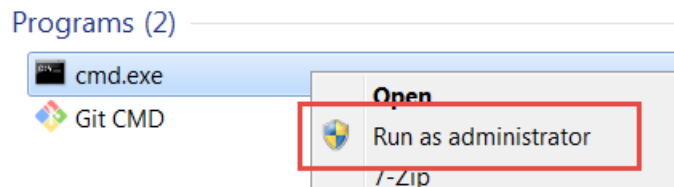
1. Verify Jenkins is running
2. Configure tools in Jenkins

Part 1 - Configure Jenkins

After the Jenkins installation, you can configure few other settings to complete the installation before creating jobs.

In this part, you will set JDK HOME and Gradle Installation directory.

- __1. Open a command prompt window as an administrator.



- __2. Change to Jenkins installation folder:

```
cd C:\Program Files\Jenkins
```

- __3. Verify Jenkins is started:

```
jenkins status
```

- __4. If it is not started then enter this command:

```
jenkins start
```

- __5. Close the window.

- __6. To connect to Jenkins, open Chrome and enter the following URL:

```
http://localhost:8080/
```

__7. Enter **wasadmin** as user and password and click **Sign in**.



Welcome to Jenkins!

Sign in

__8. Don't save the password if prompt or select Never Remember password for this site.

__9. Click on the **Manage Jenkins** link.



Don't worry about any warning.

__10. Click **Global Tool Configuration**.

__11. Scroll down and find the JDK section, Click **Add JDK**.

__12. Enter **OracleJDK** for JDK name.

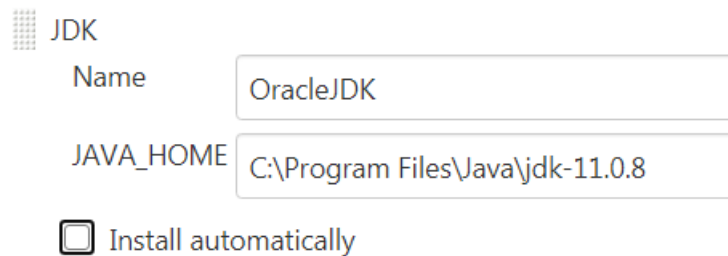
__13. Uncheck the 'Install automatically' option.

__14. Enter **JAVA_HOME** value as:

C:\Program Files\Java\jdk-11.0.8

Note. You may need to use another path if Java was installed in a different folder like above, contact your instructor or search for the right path and use it as JAVA_HOME.

__15. Verify your settings look as below:



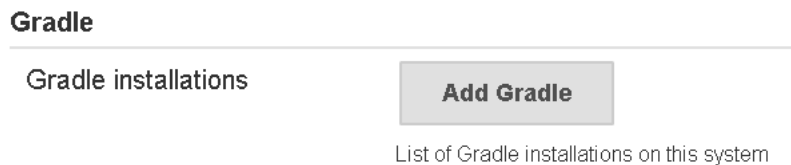
JDK

Name OracleJDK

JAVA_HOME C:\Program Files\Java\jdk-11.0.8

☐ Install automatically

__16. In the Gradle section, click **Add Gradle**.



Gradle

Gradle installations

Add Gradle

List of Gradle installations on this system


__17. Enter **Gradle** for Gradle name.

__18. Uncheck the 'Install automatically' option.

__19. Enter the following for GRADLE_HOME. Make sure this folder is correct:


C:\Software\gradle-6.6.1

__20. Verify your settings look as below:

	Gradle
name	<input type="text" value="Gradle"/>
GRADLE_HOME	<input type="text" value="C:\Software\gradle-6.6.1"/>
<input type="checkbox"/>	Install automatically

__21. Scroll and find **Git**, you may see an error regarding the git path. Enter the following path and then hit the tab key (Make sure the path is valid or find the right path):

C:\Program Files\Git\bin\git.exe

Git	
Git installations	
	Git
Name	<input type="text" value="Default"/>
Path to Git executable	<input type="text" value="C:\Program Files\Git\bin\git.exe"/>
<input type="checkbox"/>	Install automatically

__22. Click **Save**.

Part 2 - Review

In this lab you configured the Jenkins Continuous Integration Server.

Lab 8 - Create a Jenkins Job

In this lab you will create and build a job in Jenkins.

Jenkins freestyle projects allow you to configure just about any sort of build job, they are highly flexible and very configurable.

At the end of this lab you will be able to:

1. Create a Jenkins Job that accesses a Git repository.

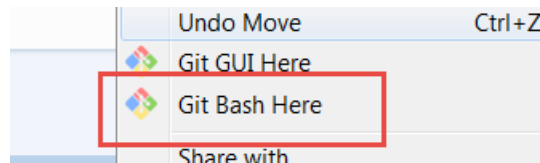
Part 1 - Create a Git Repository

As a distributed version control system, Git works by moving changes between different repositories. Any repository apart from the one you're currently working in is called a "remote" repository. Git doesn't differentiate between remote repositories that reside on different machines and remote repositories on the same machine. They're all remote repositories as far as Git is concerned. In this lab, we're going to start from a source tree, create a local repository in the source tree, and then clone it to a local repository. Then we'll create a Jenkins job that pulls the source files from that remote repository. Finally, we'll make some changes to the original files, commit them and push them to the repository, showing that Jenkins automatically picks up the changes.

- __1. Using File Explorer, navigate to the folder:

`C:\LabFiles\Create A Jenkins Job_GRADLE\SimpleGreeting`

- __2. Right click in the empty area and select **Git Bash Here**. The Git command prompt will open.



- __3. Enter the following command:

`ls`

```
$ ls
build.gradle  gradle/  gradlew*  gradlew.bat  pom.xml  settings.gradle  src/
```

__4. Enter the following lines. Press enter after each line:

```
git config --global user.email "wasadmin@webagesolutions.com"
git config --global user.name "Bob Smith"
```

The lines above are actually part of the initial configuration of Git. Because of Git's distributed nature, the user's identity is included with every commit as part of the commit data. So we have to tell Git who we are before we'll be able to commit any code.

__5. Enter the following lines to actually create the Git repository:

```
git init
git add .
git commit -m "Initial Commit"
```

The above lines create a git repository in the current directory (which will be **C:\LabFiles\Create A Jenkins Job_GRADLE\SimpleGreeting**), add all the files to the current commit set (or 'index' in git parlance), then actually performs the commit.

__6. Enter the following, to create a folder called **repos** under the C:\Software folder.

```
mkdir /c/Software/repos
```

__7. Enter the following to clone the current Git repository into a new remote repository.

```
git clone --bar . /c/Software/repos/SimpleGreeting.git
```

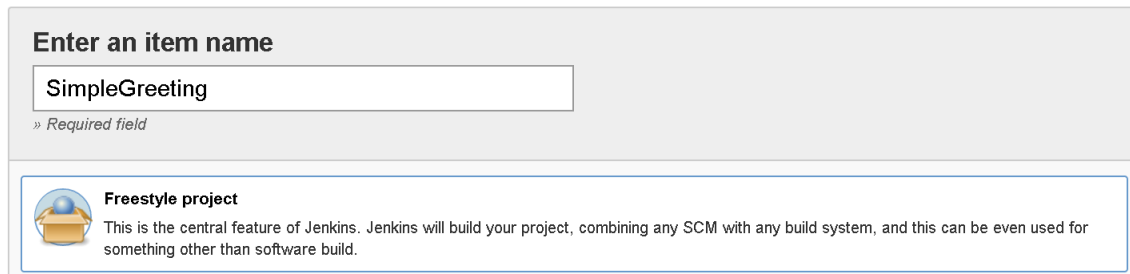
At this point, we have a "remote" Git repository in the folder **C:\Software\repos\SimpleGreeting.git**. Jenkins will be quite happy to pull the source files for a job from this repo.

Part 2 - Create the Jenkins Job

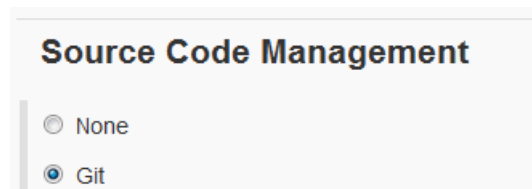
__1. Go to the Jenkins home and click on the **New Item** link.



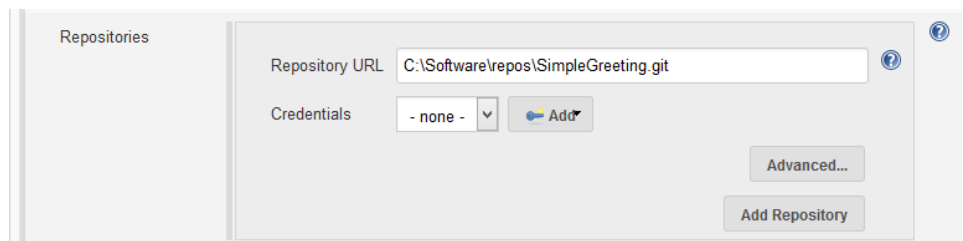
- __2. Enter **SimpleGreeting** for the project name.
- __3. Select **Freestyle project** as the project type.



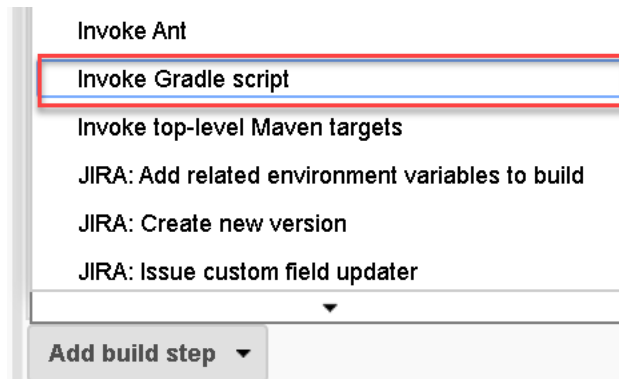
- __4. Click **OK**, to add a new job.
- After the job is created, you will be on the job configuration page.
- __5. Scroll down to the **Source Code Management** section and then select **Git**.



- __6. Under Repositories, enter **C:\Software\repos\SimpleGreeting.git** and press tab key.



__7. In **Build** section, click **Add build step** > **Invoke Gradle script**

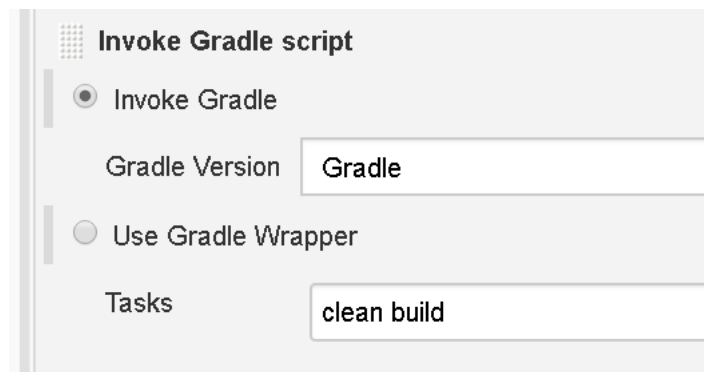


__8. Ensure **Invoke Gradle** radio button is selected.

__9. In **Gradle Version**, select **Gradle**

__10. In **Tasks**, enter **clean build**

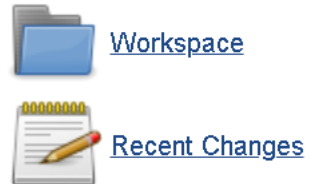
The **Invoke Gradle script** configuration should look like this:



__11. Click **Save**.

You will see the Job screen.

Project SimpleGreeting

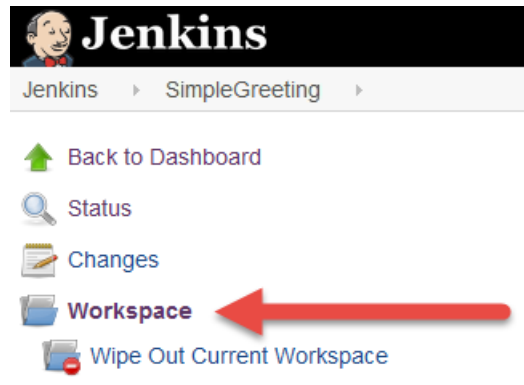


__12. Click **Build Now**.

You should see the build in progress in the **Build History** area.

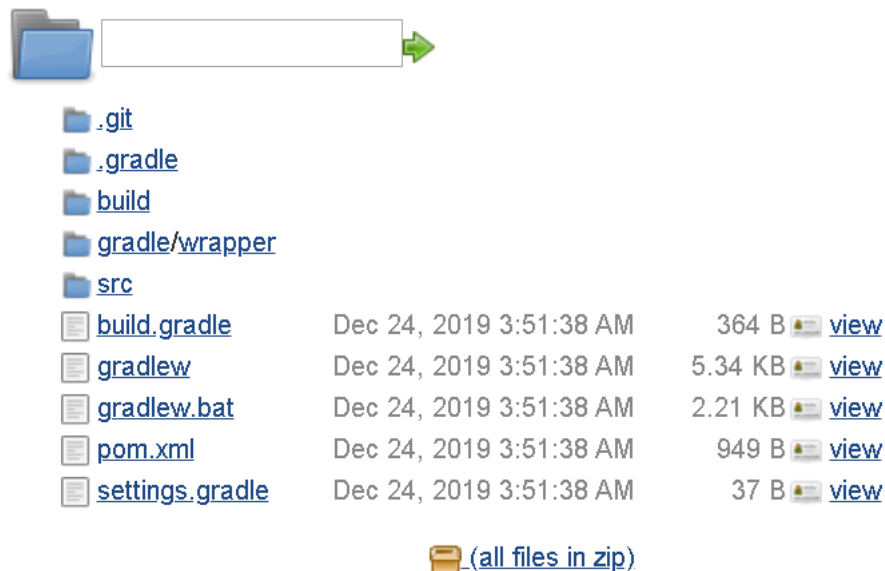


13. After a few seconds the build will complete, the progress bar will stop. Click on **Workspace**.



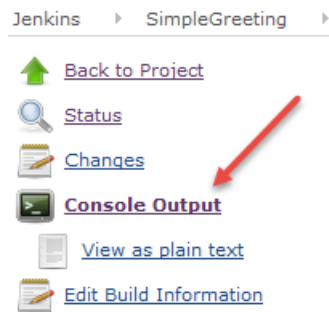
You will see that the directory is populated with the source code for our project.

Workspace of SimpleGreeting on master



14. Find the **Build History** box, and click on the 'time' value for the most recent build. You should see that the build was successful.

__15. Click the **Console Output** from the left menu.



__16. At the end of the console you will also see the build success and successful build finish.

```
BUILD SUCCESSFUL in 11s
5 actionable tasks: 5 executed
Build step 'Invoke Gradle script' changed build result to SUCCESS
Finished: SUCCESS
```

You have created a project and built it successfully.

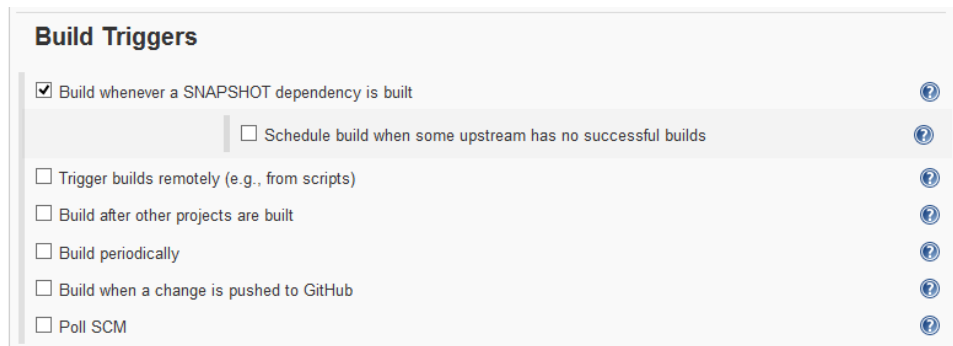
Part 3 - Enable Polling on the Repository

So far, we have created a Jenkins job that pulls a fresh copy of the source tree prior to building. But we triggered the build manually. In most cases, we would like to have the build triggered automatically whenever a developer makes changes to the source code in the version control system.

__1. In the Jenkins web application, navigate to the **SimpleGreeting** project. You can probably find the project in the breadcrumb trail near the top of the window. Alternately, go to the Jenkins home page and then click on the project.

__2. Click the **Configure** link.

___ 3. Scroll down to find the **Build Triggers** section.



___ 4. Click on the check box next to **Poll SCM**

___ 5. Enter '* * * * *' into the **Schedule** text box. [Make sure there is a space between each *]



Note: The above schedule sets up a poll every minute. In a production scenario, that's a higher frequency than we need, and it can cause unnecessary load on the repository server and on the Jenkins server. You'll probably want to use a more reasonable schedule - perhaps every 15 minutes. That would be 'H/15 * * * *' in the schedule box.

___ 6. In **Post-build Actions** section, click **Add post-build action** and select **Publish JUnit test result report**.

This will allow you to graphically view unit test results.

___ 7. In **Test reports XMLs**, enter the following:

```
build\test-results\test\TEST-*.xml
```

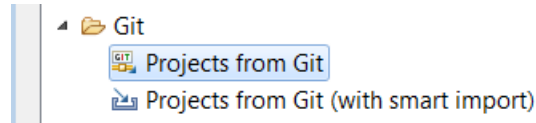
Note: By default, Gradle stores test results in the folder mentioned above. In this case, the actual file name is TEST-com.simple.TestGreeting.xml but you can use * wildcard to specify the file name.

___ 8. Click **Save**.

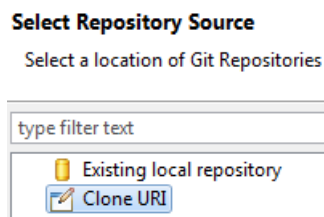
Part 4 - Import the Project into Eclipse

In order to make changes to the source code, we'll clone a copy of the Git repository into an Eclipse project.

- ___1. Start Eclipse by running `C:\Software\eclipse\eclipse.exe` and use `C:\Workspace` as Workspace.
- ___2. From the main menu, select **File** → **Import...**
- ___3. Select **Git** → **Projects from Git**.

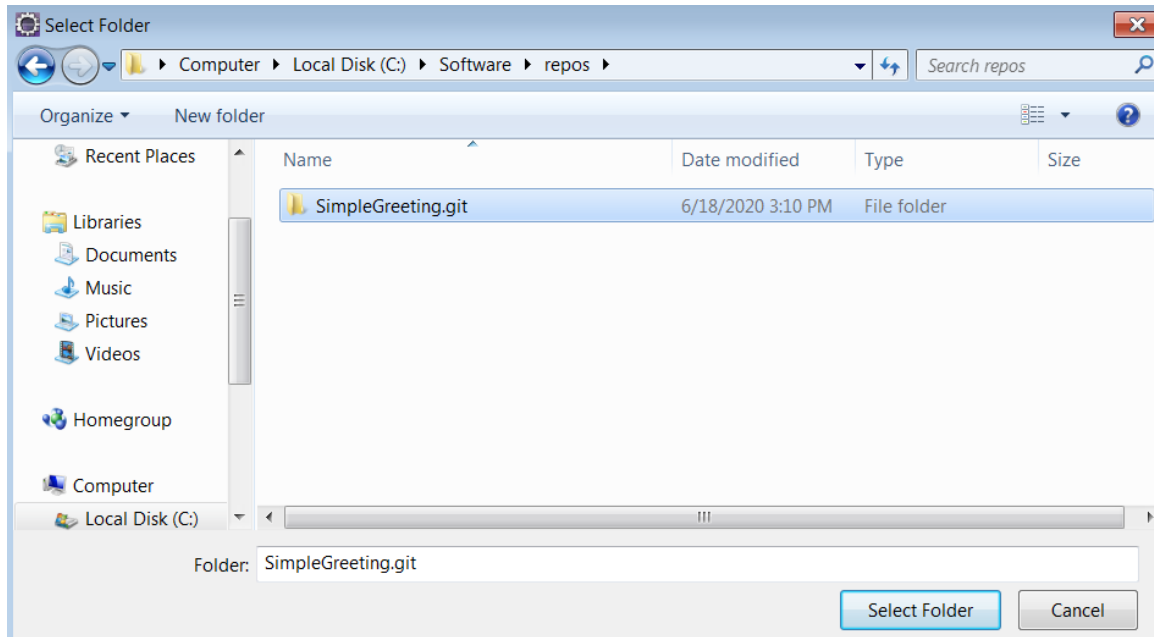


- ___4. Click **Next**.
- ___5. Select **Clone URI** and then click **Next**.



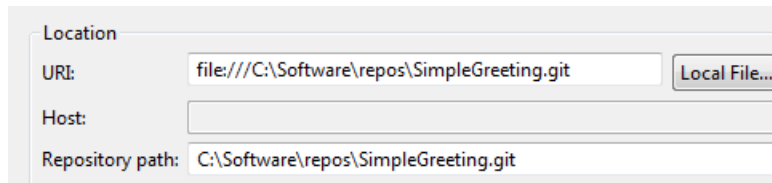
You might think that 'Existing local repository' would be the right choice, since we're cloning from a folder on the same machine. Eclipse, however, expects a "local repository" to be a working directory, not a bare repository. On the other hand, Jenkins will complain if we try to get source code from a repository with a working copy. So the correct thing is to have Jenkins pull from a bare repository, and use **Clone URI** to have Eclipse import the project from the bare repository.

__6. Click on **Local File...** and then navigate to **C:\Software\repos\SimpleGreeting.git**

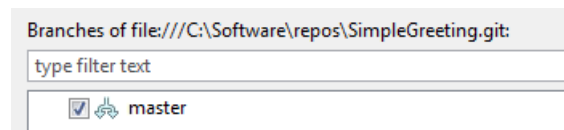


__7. Click **Select Folder**.

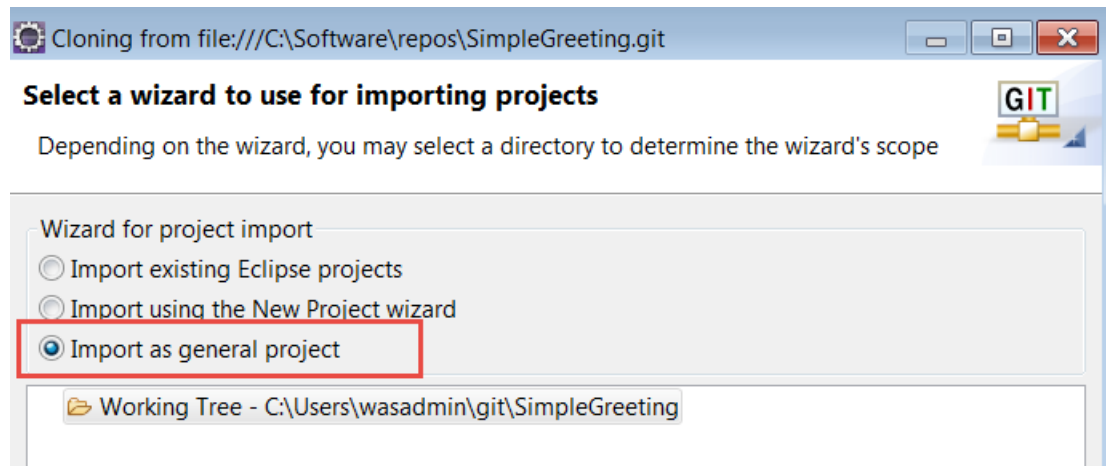
__8. Back in the **Import Projects** dialog, click **Next**.



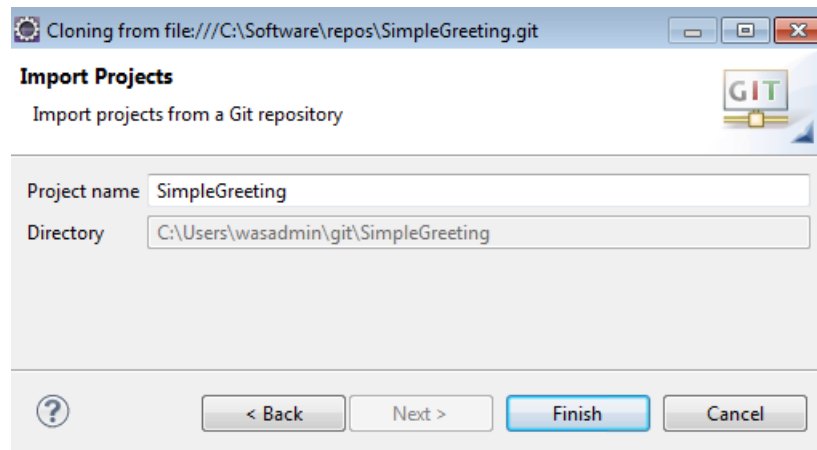
__9. Click **Next** to accept the default 'master' branch.



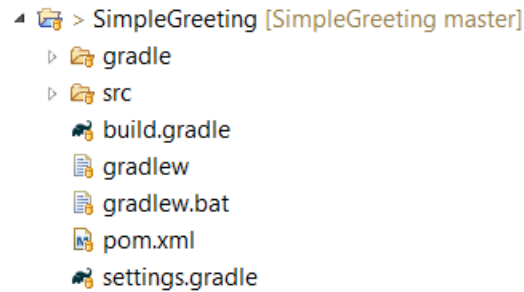
- __10. In the **Local Destination** pane, leave the defaults and click **Next**.
- __11. Select **Import as a general project** and click **Next**.



- __12. Click **Finish**.



__13. You should see the new project in the **Project Explorer**, expand it.



In real-world, after this step, we should convert our project in Eclipse so it understands Gradle project layout. However, it's not required to understand Jenkins continuous integration so we will leave the project layout as it is.

Part 5 - Make Changes and Trigger a Build

The project that we used as a sample consists of a basic "Hello World" style application, and a unit test for that application. In this section, we'll alter the core application so it fails the test, and then we'll see how that failure appears in Jenkins.

- __1. In the **Project Explorer**, expand the **src/main/java/com/simple** tree node to reveal the **Greeting.java** file.
- __2. Double-click on **Greeting.java** to open the file.
- __3. Find the line that says 'return "GOOD";'. Edit the line to read 'return "BAD";'

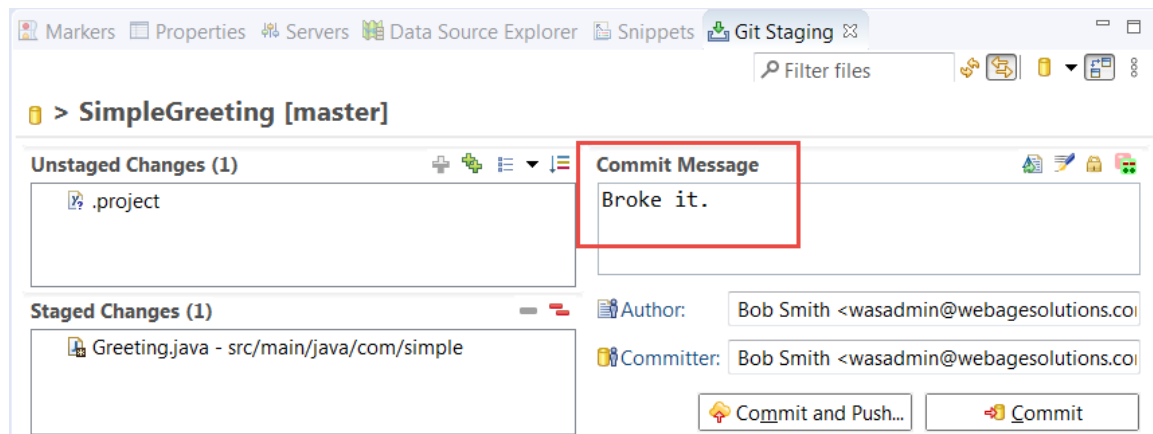
```
public String getStatus(){  
    return "BAD";  
}
```

- __4. Save the file by pressing Ctrl-S or selecting **File** → **Save**.

Now we've edited the local file. The way Git works is that we'll first 'commit' the file to the local workspace repository, and then we'll 'push' the changes to the upstream repository. That's the same repository that Jenkins is reading from. Eclipse has a short-cut button that will commit and push at the same time.

__5. Right-click on **SimpleGreeting** in the **Project Explorer** and then select **Team** → **Commit**.

__6. Eclipse will open the **Git Staging** tab. Enter a few words as a commit message, and then click **Commit and Push**.



__7. Click **Close** in the status dialog that pops up.

__8. Now, flip back to the web browser window that we had Jenkins running in. If you happen to have closed it, open a new browser window and navigate to **http://localhost:8080/SimpleGreeting**. After a few seconds, you should see a new build start up or you can click **Build now** to launch a new build if it's taking too long.

__9. This build should fail. (red circle)

Make sure you are using Chrome, when writing this Lab, Mozilla was not working properly.

What happened is that we pushed the source code change to the Git repository that Jenkins is reading from. Jenkins is continually polling the repository to look for changes. When it saw that a new commit had been performed, Jenkins checked out a fresh copy of the source code and performed a build. Since Gradle automatically runs the unit tests as part of a build, the unit test was run. It failed, and the failure results were logged.

Part 6 - Fix the Unit Test Failure

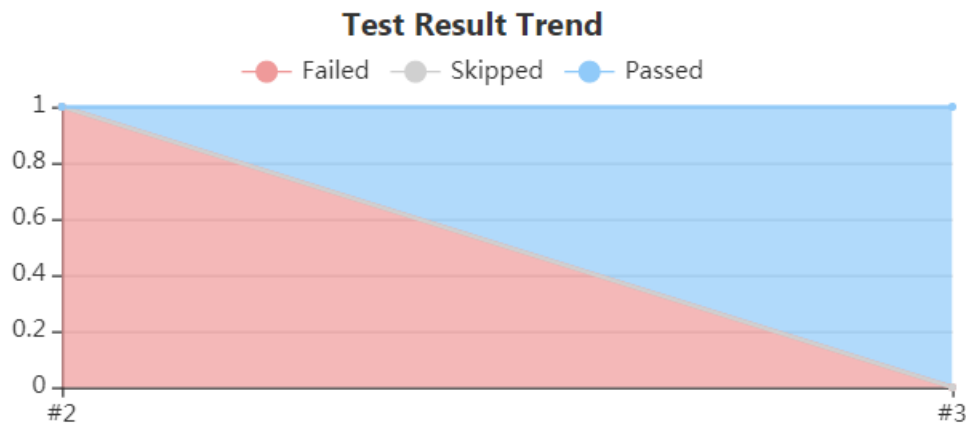
__1. Back in eclipse, edit the file **Greeting.java** so that the class once again returns 'GOOD'.

```
public String getStatus(){  
  
    return "GOOD";  
  
}
```

__2. As above, save the file, commit and then 'Commit and Push' the change.

__3. Watch the Jenkins web browser window. After a minute or two you should see the build start automatically or you can click **Build now**, this time the build will pass, when the build is done then refresh the page.

You will see a graph in the right corner:



Part 7 - Review

In this lab you learned

- How to Set-up a set of distributed Git repositories
- How to create a Jenkins Job that reads from a Git repository
- How to configure Jenkins to build automatically on source code changes.

Lab 9 - Create a Pipeline

In this lab you will explore the Pipeline functionality.

At the end of this lab you will be able to:

1. Create a simple pipeline
2. Use a 'Jenkinsfile' in your project
3. Use manual input steps in a pipeline

Part 1 - Create a Simple Pipeline

We can create a pipeline job that includes the pipeline script in the job configuration, or the pipeline script can be put into a 'Jenkinsfile' that's checked-in to version control.


To get a taste of the pipeline, we'll start off with a very simple pipeline defined in the job configuration.

Prerequisite: We need to have a project in source control to check-out and build. For this example, we're using the 'SimpleGreeting' project that we previously cloned to a 'Git' repository at 'C:\Software\repos\SimpleGreeting.git'


- __ 1. In Jenkins, go to home and then click on the **New Item** link.
- __ 2. Enter '**SimpleGreetingPipeline**' as the new item name, and select '**Pipeline**' as the item type.

Enter an item name

» Required field

**Freestyle project**

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Pipeline**

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

- __ 3. When the input looks as above, click on **OK** to create the new item.

__4. Scroll down to the **Pipeline** section and enter the following in the **Script** text window.

```
node {  
  stage('Checkout') {  
    git url: 'C:\\Software\\repos\\SimpleGreeting.git'  
  }  
  
  stage('Gradle build') {  
    bat 'gradle build'  
  }  
}
```

This pipeline is divided into two stages. First, we checkout the project from our 'git' repository. Then we use the 'bat' command to run 'gradle build' as a Windows batch file.

All of the above is wrapped inside the 'node' command, to indicate that we want to run these commands in the context of a workspace running on one of Jenkins execution agents (or the master node if no agents are available).

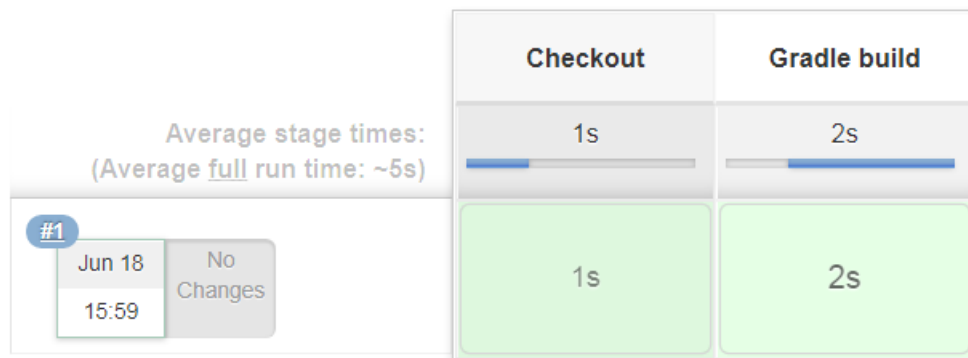
__5. Click on **Save** to save the changes and return to the project page.

__6. Click on **Build Now** to start up a pipeline instance.



__7. After a few moments, you should see the **Stage View** appear, and successive stages will appear as the build proceeds, until all stages are completed.

Stage View



Part 2 - Pipeline Definition in a 'Jenkinsfile'

For simple pipelines or experimentation, it's convenient to define the pipeline script in the web interface. But one of the common themes of modern software development is "If it isn't in version control, it didn't happen". The pipeline definition is no different, especially as you build more and more complex pipelines.

You can define the pipeline in a special file that is checked out from version control. There are several advantages to doing this. First, of course, is that the script is version-controlled. Second, we can edit the script with the editor or IDE of our choice before checking it in to version control. In addition, we can employ the same kind of "SCM Polling" that we would use in a more traditional Jenkins job.

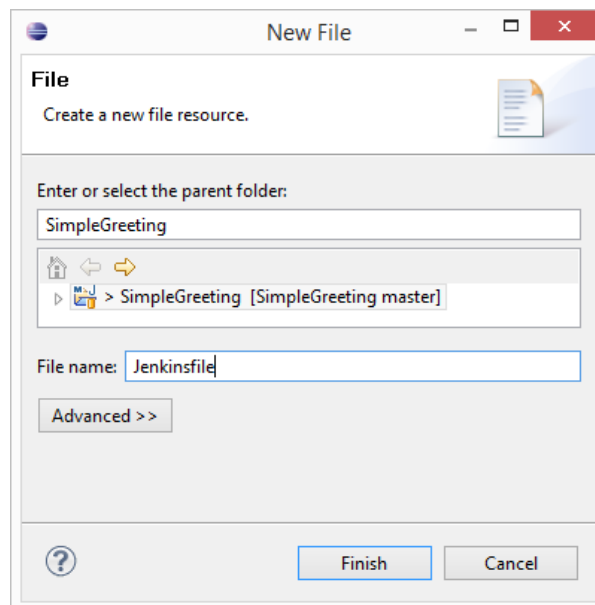
In the following steps, we'll create a Jenkinsfile and create a pipeline job that uses it.

___ 1. Open the Eclipse editor.

If this lab is completed in the normal sequence, you should have the 'SimpleGreeting' project already in Eclipse's workspace. If not, check out the project from version control (consult your instructor for directions if necessary).

___ 2. In the **Project Explorer**, right-click on the root node of the **SimpleGreeting** project, and then select **New** → **File**.

___ 3. Enter '**Jenkinsfile**' as the file name.

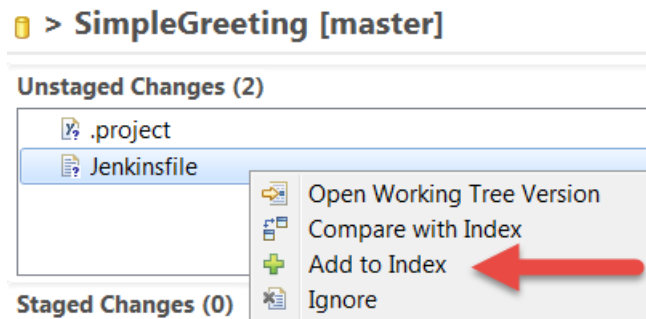


___ 4. Click **Finish** to create the new file.

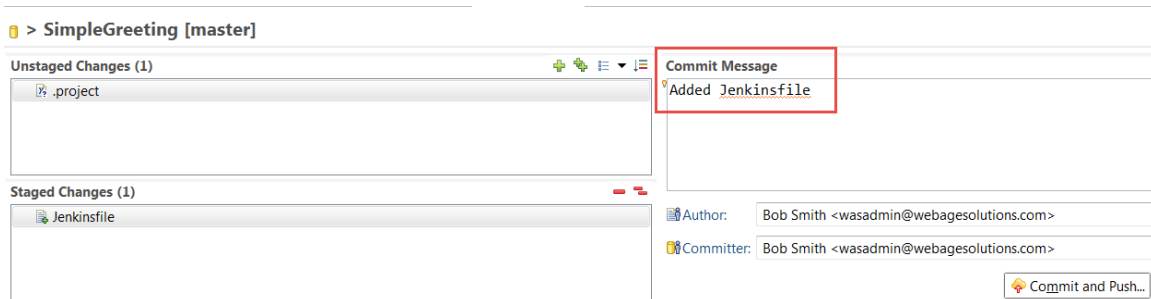
- __5. You may see the 'Editors available on the Marketplace' window, just click Cancel.
- __6. You may see a compatibility error, just click OK.
- __7. Enter the following text into the new file (Note: this is the same script that we used above, so you could copy/paste it from the Jenkins Web UI if you want to avoid some typing):

```
node {  
    stage('Checkout') {  
        git url: 'C:\\Software\\repos\\SimpleGreeting.git'  
    }  
  
    stage('Gradle build') {  
        bat 'gradle build'  
    }  
}
```

- __8. Save the Jenkinsfile by selecting **File** → **Save** from the main menu, or by hitting Ctrl-S.
- __9. In the **Project Explorer**, right-click on the **SimpleGreeting** node, and then select **Team** → **Commit...**
- __10. Eclipse will open the **Git Staging** tab. Right click **Jenkinsfile** and click **Add to Index**.



__11. The file will be now available in the 'Staged Changed' section. Add a comment and click **Commit and Push** then click **Close** to dismiss the next window.



Now we have a Jenkinsfile in our project, to define the pipeline. Next, we need to create a Jenkins job to use that pipeline.

__12. In the Jenkins user interface, navigate to the root page, and then click on **New Item**.

__13. Enter '**SimpleGreetingPipelineFromGit**' as the name of the new item.

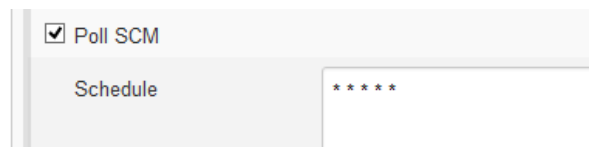
__14. Select **Pipeline** as the item type.

__15. Click **OK** to create the new item.

__16. Scroll down to the **Build Triggers** section.

__17. Click on **Poll SCM**

__18. Enter '*** * * * ***' as the polling schedule. This entry will cause Jenkins to poll once per minute.

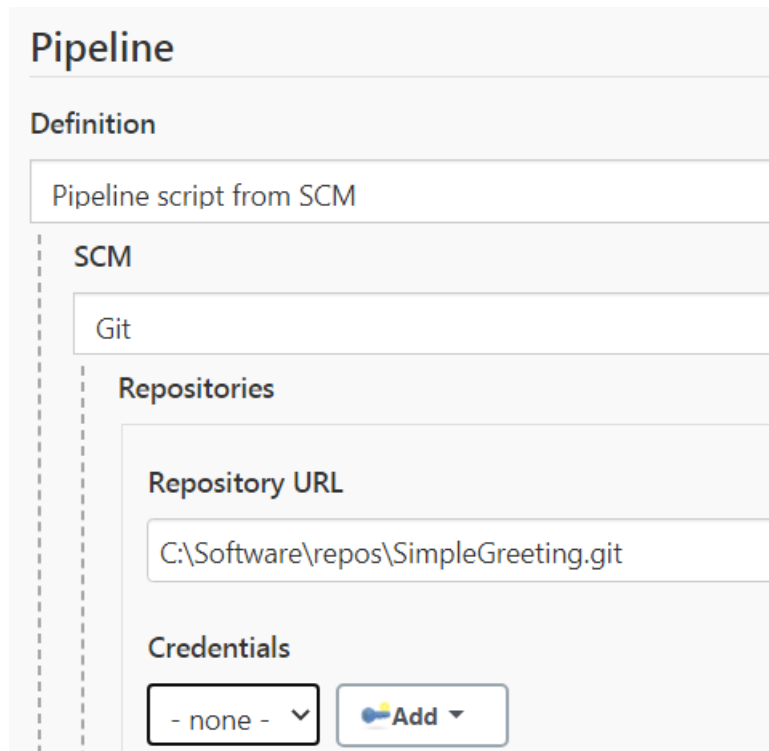


__19. Scroll down to the **Pipeline** section, and change the **Definition** entry to '**Pipeline Script from SCM**'.

__20. Enter the following:

SCM:	Git
Repository URL:	C:\Software\repos\SimpleGreeting.git (Press the tab key)

__21. The **Pipeline** section should look similar to:



The screenshot shows a web-based configuration interface for a pipeline. It has a main title 'Pipeline' and a section 'Definition' with a dropdown menu set to 'Pipeline script from SCM'. Below this is a section 'SCM' with a dropdown menu set to 'Git'. Under 'SCM', there is a 'Repositories' section with a 'Repository URL' field containing 'C:\Software\repos\SimpleGreeting.git'. At the bottom, there is a 'Credentials' section with a dropdown menu set to '- none -' and an 'Add' button with a key icon.

__22. Click **Save** to save the new configuration.

__23. Click **Build Now** to launch the pipeline.

__24. You should see the pipeline execute, similar to the previous section.

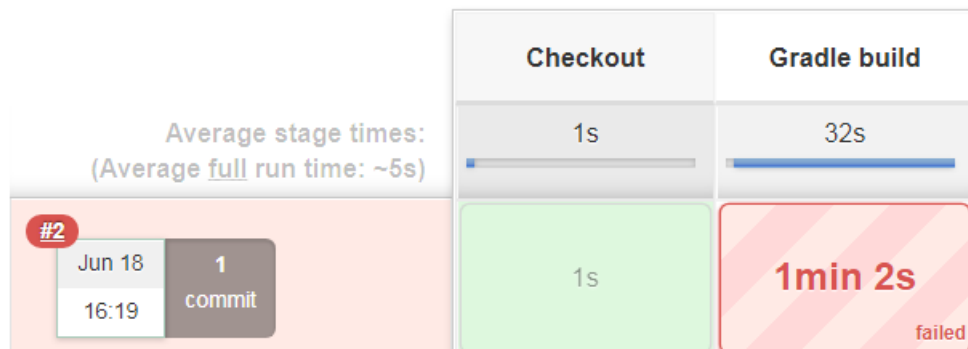
Part 3 - Try out a Failing Build

The pipeline that we've defined so far appears to work perfectly. But we haven't tested it with a build that fails. In the following steps, we'll insert a test failure and see what happens to our pipeline.

- __1. In Eclipse, go to the **Project Explorer** and locate the file 'Greeting.java'. It will be under **src/main/java** in the package 'com.simple'.
- __2. Open 'Greeting.java'.
- __3. Locate the line that reads 'return "GOOD";'. Change it to read 'return "BAD";'

```
public String getStatus() {  
  
    return "BAD";  
  
}
```

- __4. Save the file.
- __5. In the **Project Explorer**, right-click on **Greeting.java** and then select **Team** → **Commit...** (This is a shortcut for committing a single file).
- __6. Enter an appropriate commit message and then click **Commit and Push**, then click **Close** in the results box.
- __7. Switch back to Jenkins.
- __8. In a minute or so, you should see a build launched automatically. Jenkins has picked up the change in the 'Git' repository and initiated a build. If nothing happens then click **Build Now**.



This time, the results are a little different. The 'Gradle Build' stage is showing a failure.

What's happened is that the unit tests have failed and Gradle exited with a non-zero result code because of the failure. As a result, the rest of the pipeline was canceled. In case if you want to perform a build without running unit tests, you can add `-x test` to *gradle build*. This way Gradle will continue with the build even if the tests fail.

__ 9. Go back to Eclipse and open the '**Jenkinsfile**' if necessary.

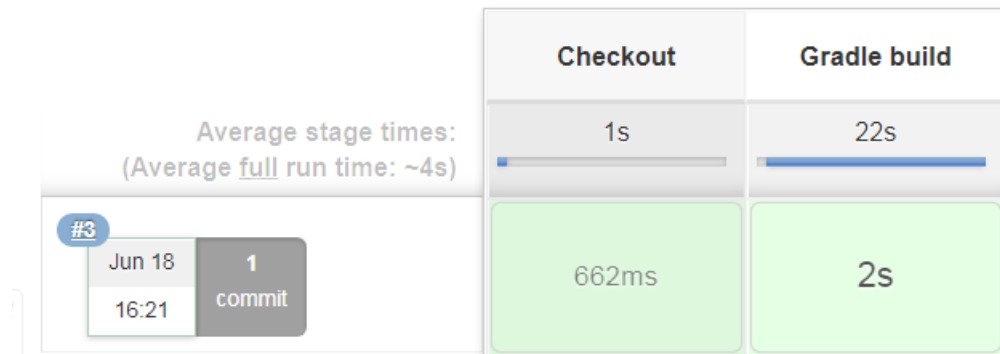
__ 10. Alter the 'bat "gradle"' line to read as follows:

```
bat 'gradle build -x test'
```

__ 11. Save the 'Jenkinsfile'.

__ 12. Commit and push the changes using the same technique as above.

__ 13. After a minute or so, you should see a new Pipeline instance launched. If nothing happens then click **Build Now**.



This time, the pipeline runs to completion.

Part 4 - Add a Manual Approval Step

One of the interesting features of the Pipeline functionality is that we can include manual steps. This is very useful when we're implementing a continuous deployment pipeline. For example, we can include a manual approval step (or any other data collection) for cases like 'User Acceptance Testing' that might not be fully automated.

In the steps below, we'll add a manual step before a simulated deployment.

__ 1. Go to Eclipse and open the '**Jenkinsfile**' if necessary.

__2. Add the following to the end of the file before the **node** block closing }:

```
stage('User Acceptance Test') {

    def response= input message: 'Is this build good to go?',
        parameters: [choice(choices: 'Yes\nNo',
            description: '', name: 'Pass')]

    if(response=="Yes") {
        stage('Deploy') {
            bat 'gradle build -x test'
        }
    }
}
```

This portion of the script creates a new stage called 'User Acceptance Test', then executes an 'input' operation to gather input from the user. If the result is 'Yes', the script executes a deploy operation. (In this case, we're repeating the 'gradle build' that we did previously. Only because we don't actually have a deployment repository setup)

__3. Save and commit 'Jenkinsfile' as previously.

__4. After a minute or so, you should see a new Pipeline instance launched. If nothing happens then click **Build Now**. When the pipeline executes, watch for a "paused" stage called 'User Acceptance Test'. Move your mouse over this step, you'll be able to select "Yes" or "No".

Note. If the Build is taking too long is probably because SimpleGreeting is launching a build every time you commit changes in eclipse and your server is hanging the request in the SimpleGreetingPipelineFromGit project. You can stop all builds on SimpleGreeting but do not delete it because will be used in a later lab.

The screenshot displays the Jenkins Pipeline console interface. At the top, a table shows the average stage times: Checkout (990ms), Gradle build (17s), and User Acceptance Test (373ms). Below this, a list of pipeline runs is shown, with the most recent run (#4) highlighted. The console output for run #4 shows the 'User Acceptance Test' stage is paused. A dialog box is overlaid on the console, asking 'Is this build good to go?' with a 'Pass' status and a dropdown menu set to 'Yes'. The dialog has 'Proceed' and 'Abort' buttons. The background of the console shows the 'User Acceptance Test' stage is paused for 10s.

Stage	Average time
Checkout	990ms
Gradle build	17s
User Acceptance Test	373ms

Average stage times:
(Average full run time: ~4s)

#4 Jun 18 16:23 1 commit

#3 Jun 18 16:21 1 commit

Is this build good to go? ×

Pass

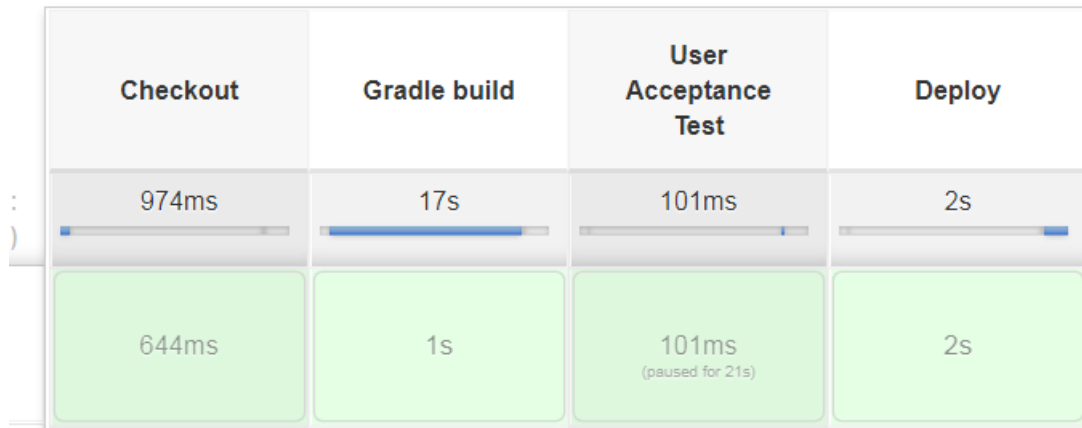
Yes ▼

Proceed Abort

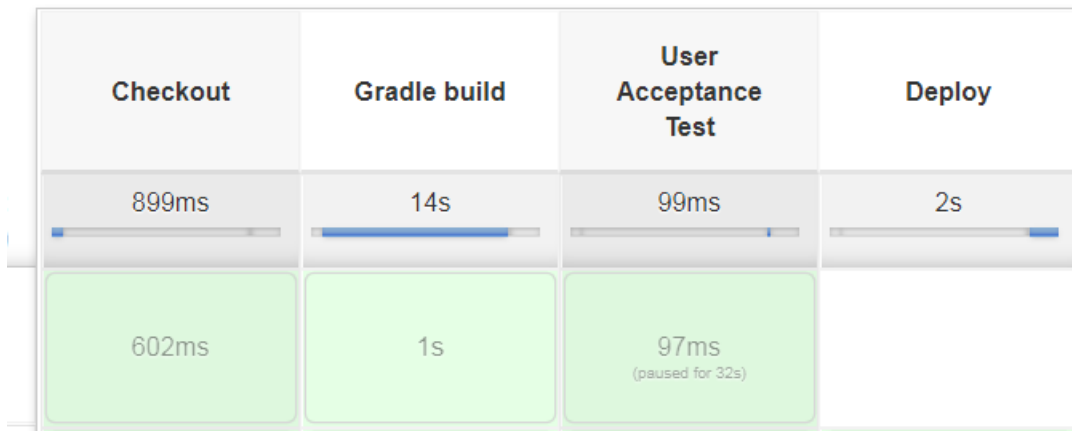
(paused for 10s)

___5. Select **Yes** and click **Proceed**.

You should see the job run to completion.



___6. Click **Build Now** to run the pipeline again but this time, select **No** on the 'User Acceptance Test' and click **Proceed**, you'll see that the pipeline exits early and doesn't run the 'deploy' stage.



What's happened is that the final 'Deploy' stage was only executed when we indicated that the 'User Acceptance Test' had passed.

___7. Close all.

Part 5 - Review

In this lab, we explored the Pipeline functionality in Jenkins. We built a simple pipeline in the Jenkins web UI, and then used a 'Jenkinsfile' in the project. Lastly, we explored how to gather user input, and then take different build actions based on that user input.

Lab 10 - Advanced Pipeline with Groovy DSL

In this lab you will utilize Groovy DSL and Shared Library to create an advanced pipeline in Jenkins.

Part 1 - Explore Groovy DSL basics

In this part you will explore Groovy DSL basics. You will use Jenkins built-in Script Console for testing the Groovy script.

__ 1. In the web browser enter following URL to access Jenkins:

`http://localhost:8080/`

__ 2. Login as **wasadmin** for user and password.

__ 3. Click **Manage Jenkins**.



__ 4. Click **Manage Nodes and Clouds**.



Manage Nodes and Clouds

Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

__ 5. If you see a **Linux Build** like below, click on it.

S	Name ↓	Architecture	Clock Difference	Free Disk
	Linux Build	Linux (amd64)	In sync	6
	master	Windows 8.1 (x86)	In sync	24
Data obtained		33 sec	33 sec	

__ 6. Then click **Delete Agent** and click **Yes** to confirm deletion.



[Delete Agent](#)

__7. Click **Manage Jenkins**.

__8. Click **Script Console**.



Script Console

Executes arbitrary script for administration/trouble-shooting/diagnostics.

Notice it displays a text box where you can write script and run it to see the result. Alternatively, you can access the Script Console by accessing the URL <http://localhost:8080/script>

__9. Enter following text:

```
int square(int a) {  
    return a * a;  
}  
  
println "Square of 5 is: ${square(5)}"
```

__10. Click **Run** button.

Notice it displays "Square of 5 is: 25".

__11. In the **Script Console** text box, remove previous code and enter the following code:

```
def printFile(location) {  
    pub = new File(location)  
    if (pub.exists()){  
        println "Location ${location}"  
        pub.eachLine{line-> println line}  
    }  
    else{  
        println "${location} does not exist"  
    }  
}  
  
printFile("C:/Windows/System32/drivers/etc/hosts")
```

__12. Click **Run**.

Notice it displays file contents.

Part 2 - Create a directory structure for storing Shared Library content and initialize the Git repository.

In this part you will you create a simple library of reusable functions and add it to Jenkins as a shared library.

- __ 1. Open **Command Prompt**.
- __ 2. Switch to the **Workspace** directory:

```
cd c:\workspace
```

- __ 3. Create a directory for storing Groovy libraries and switch to it:

```
mkdir groovy && cd groovy
```

- __ 4. Create a directory for storing your first library's scripts and switch to the directory:

```
mkdir MyFirstLibrary.git && cd MyFirstLibrary.git
```

- __ 5. Initialize Git repository:

```
git init .
```

- __ 6. Configure user name and email address required for committing changes to the repository:

```
git config user.name "Bob"
```

```
git config email.address "bob@abcinc.com"
```

- __ 7. Create directory structure for storing the Groovy source code:

```
mkdir src\com\abcinc
```

Note: **src** is where source code must reside. In the **src** directory you can organize source code in the form of packages. com.abcinc package corresponds to com\abcinc directory structure.

Part 3 - Create a simple Groovy script, commit it to the repository, and add it to Jenkins as a shared library.

In this part you will create a simple Groovy script, commit it to the repository, add it to Jenkins as a shared library.

__1. Create a simpleClass.groovy file using the Notepad. Click Yes to create the file:

```
notepad src\com\abcinc\simpleClass.groovy
```

__2. Enter following text:

```
package com.abcinc

import java.text.SimpleDateFormat

class simpleClass {
    int square(int a) {
        return a * a;
    }

    def sayHello(def name) {
        return "Hi, ${name}";
    }

    def getDateTime() {
        def date = new Date()
        def sdf = new SimpleDateFormat("MM/dd/yyyy HH:mm:ss")
        return sdf.format(date);
    }
}
```

__3. Save and close the file.

__4. Add the content to the Git repository:

```
git add .
```

__5. Commit changes:

```
git commit -m "Added simpleClass"
```

__ 6. In the web browser, open Jenkins:

http://localhost:8080

__ 7. Click **Manage Jenkins**.

__ 8. Click **Configure System**.

__ 9. Scroll down the page, locate **Global Pipeline Libraries**, and click **Add** button.

Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.



__ 10. In **Name** text box, enter "**my-shared-lib**" (without quotes).

__ 11. In **Default Version** text box, enter "**master**" (without quotes).

__ 12. Under **Retrieval Method**, click "**Modern SCM**".

__ 13. Under **Source Code Management** click **Git**.

__ 14. In **Project Repository** text box, enter "**c:\workspace\groovy\MyFirstLibrary.git**" (without quotes) and press tab in your keyboard.

Library	
Name	<input type="text" value="my-shared-lib"/>
Default version	<input type="text" value="master"/> <small>Cannot validate default version until after saving and reconfiguring.</small>
Load implicitly	<input type="checkbox"/>
Allow default version to be overridden	<input checked="" type="checkbox"/>
Retrieval method	
<input checked="" type="radio"/> Modern SCM	
Source Code Management	
<input checked="" type="radio"/> Git	
Project Repository	<input type="text" value="c:\workspace\groovy\MyFirstLibrary.git"/>

Note, screen may vary between Jenkins versions.

__ 15. Click **Apply** button.

__ 16. Click **Save** button.

Note: Next you will verify you are connected to the repository properly.

__ 17. Click **Manage Jenkins**.

__ 18. Click **Configure System**.

__ 19. Scroll down the page and notice it shows the repository revision number.

Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library	
Name	my-shared-lib
Default version	master
	Currently maps to revision: 2ea0ae0b71303c9300a92c795297473bf2113e49
Load implicitly	<input type="checkbox"/>
Allow default version to be overridden	<input checked="" type="checkbox"/>

Note: The number will most likely be different on your side.

Part 4 - Create a Jenkins Job and utilize the shared library you created previously

In this part you will create a Jenkins Job and utilize the shared library you created previously.

__ 1. In the web browser ensure you have Jenkins web site open:

http://localhost:8080

__ 2. Click **New Item**.

__ 3. In **Enter an item name** enter "**Shared Library Pipeline**" (without quotes).

__ 4. Select **Pipeline**.

__ 5. Click **OK** button.

__ 6. Scroll through the page, locate **Pipeline** section, and in **Script** enter following text:

```
@Library('my-shared-lib')
import com.abcinc.simpleClass;

def m = new simpleClass();

println m.sayHello("Bob");
println "Square is: ${m.square(5)}";
println "Date is: ${m.getDateTime()}";
```

__ 7. Click **Apply** button.

Note: Ignore red icon(s), if there are any.

__ 8. Click **Save** button.

__ 9. Click **Build Now**.

Notice a build shows up under **Build History** with blue colored icon.

__ 10. Click the build under **Build History**.

__ 11. Click **Console Output**.

Notice the pipeline execution checkouts the Groovy script(s) from the repository then displays the results. The result should be similar to this:

```
Started by user Administrator
Loading library my-lib@master
> git.exe rev-parse --is-inside-work-tree # timeout=10
Setting origin to C:\Workspace\groovy\MyGroovyLibrary
> git.exe config remote.origin.url C:\Workspace\groovy\MyGroovyLibrary # timeout=10
Fetching origin...
Fetching upstream changes from origin
> git.exe --version # timeout=10
> git.exe fetch --tags --progress origin +refs/heads/*:refs/remotes/origin/*
> git.exe rev-parse "master^{commit}" # timeout=10
> git.exe rev-parse "origin/master^{commit}" # timeout=10
> git.exe rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
> git.exe config remote.origin.url C:\Workspace\groovy\MyGroovyLibrary # timeout=10
Fetching upstream changes from C:\Workspace\groovy\MyGroovyLibrary
> git.exe --version # timeout=10
> git.exe fetch --tags --progress C:\Workspace\groovy\MyGroovyLibrary +refs/heads/*:refs/remotes/origin/*
Checking out Revision 56c3b67d784ee6d41060d65e093435e8d2009e26 (master)
> git.exe config core.sparsecheckout # timeout=10
> git.exe checkout -f 56c3b67d784ee6d41060d65e093435e8d2009e26
> git.exe rev-list 56c3b67d784ee6d41060d65e093435e8d2009e26 # timeout=10
[Pipeline] echo
Hi, Bob
[Pipeline] echo
Square is: 25
[Pipeline] echo
Date is: 02/17/2017 07:32:49
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Part 5 - Create a standardized software engineering process shared library

In this part you will define a standardized software engineering process as a shared library. It will clean, build, verify, await approval, and deploy the project.

__1. In the **Command Prompt** switch to the directory where you will store scripts for the shared library:

```
cd c:\workspace\groovy\MyFirstLibrary.git
```

__2. Open **Notepad** and create file for the library. Click Yes to create the file:

```
notepad src\com\abcinc\utils.groovy
```

__3. Enter following text:

```
package com.abcinc;

def checkout() {
    node {
        stage 'Checkout'
        git url: 'C:\\Software\\repos\\SimpleGreeting.git'
    }
}

def mvn_install() {
    node {
        stage 'Install'
        bat 'mvn install'
    }
}

def mvn_clean() {
    node {
        stage 'Clean'
        bat 'mvn clean'
    }
}
```

```

def mvn_verify() {
  node {
    stage 'Verify'
    bat 'mvn verify'
  }
}

def archive_reports() {
  node {
    stage 'Archive Reports'
    step([$class: 'JUnitResultArchiver', testResults: '**/target/
surefire-reports/TEST-*.xml'])
  }
}

def user_acceptance(wkdir) {
  node {
    stage 'User Acceptance Test'

    def response= input message: 'Is this build good to go?',
      parameters: [choice(choices: 'Yes\nNo',
        description: '', name: 'Pass')]

    if(response=="Yes") {
      stage 'Deploy'

      bat "xcopy \"$WORKSPACE\\target\\SimpleGreeting*.jar\"
C:\\workspace\\dev\\ /y"
    }
  }
}

```

IMPORTANT NOTE:

Checkout function utilizes git to checkout from an existing repository. Ensure **c:\software\repos\SimpleGreeting.git** directory is present.

In case if SimpleGreeting.git is not present, duplicate the SimpleGreeting directory as SimpleGreeting.git, then run **git add .** followed by **git commit -m "added SimpleGreeting.git"**

__ 4. Save and close the file.

__ 5. In **Command Prompt** stage the new file:

```
git add .
```


__ 6. Commit the changes to the repository:

```
git commit -a -m "added utils.groovy"
```

Note: You have already configured the c:\workspace\groovy\MyFirstLibrary as a shared library in previous parts of this lab. It is configured as shared library with the name "my-shared-lib".

Part 6 - Clean projects in Jenkins

__ 1. In the web browser ensure you have Jenkins web site open:

```
http://localhost:8080
```

__ 2. Click on **SimpleGreetingPipelineFromGit** project.



__ 3. Click **Delete Pipeline**

__ 4. Click **OK** to confirm.

__ 5. You will send back to Jenkins home, make sure the project has gone.

Part 7 - Create a Jenkins Job and utilize the shared library

In this part you will create a Jenkins Job and utilize the shared library created in the previous part of this lab.

__ 1. In the web browser ensure you have Jenkins web site open:

```
http://localhost:8080
```

__ 2. Click **New Item**.

__ 3. In **Enter an item name** enter "ABCInc" (without quotes).

__ 4. Select **Pipeline**.

__ 5. Click **OK** button. Scroll through the page, locate **Pipeline** section, and in **Script** enter following text:

```
@Library('my-shared-lib')
import com.abcinc.utils;

def u = new utils();

u.checkout();
u.mvn_clean();
u.mvn_install();
u.mvn_verify();
u.user_acceptance("ABCInc");
```

Note: utils is an implicit class. A class having same name as groovy script file is automatically created.

__ 6. Click **Apply** button.

Note: Ignore red icon(s), if there are any.

__ 7. Click **Save** button.

Before Build we need to fix the code that we broke in a previous lab.

__ 8. Open eclipse in the same workspace.

__ 9. Go to the **Project Explorer** and locate the file 'Greeting.java'. It will be under **src/main/java** in the package 'com.simple'.

__ 10. Open 'Greeting.java'.

__ 11. Locate the line that reads 'return "BAD";'. Change it to read 'return "GOOD";'

__ 12. Save the file.

__ 13. In the **Project Explorer**, right-click on **Greeting.java** and then select **Team → Commit...** (This is a shortcut for committing a single file).

__ 14. Enter an appropriate commit message and then click **Commit and Push**.

__ 15. Click Close.

__ 16. Go back to Jenkins.

__ 17. Click **Build Now**.

__ 18. When the stage progresses to "User Acceptance", hover the mouse over "**User Acceptance Test**", select "**Yes**", and click "**Proceed**" button.

The build will complete all stages.

Checkout	Clean	Install	Verify	User Acceptance Test	Deploy
582ms	1s	3s	2s	150ms	445ms
718ms	1s	3s	2s	120ms (paused for 9s)	642ms

__19. After all stages are completed, notice a new build shows up under **Build History**.

__20. Click the latest build.

__21. Click **Console Output**.

__22. Notice it shows all stages and their details.

```
Approved by Administrator
[Pipeline] stage (Deploy)
Using the 'stage' step without a block argument is deprecated
Entering stage Deploy
Proceeding
[Pipeline] bat

C:\Users\wasadmin\AppData\Local\Jenkins\.jenkins\workspace\ABCInc>xcopy
"C:\Users\wasadmin\AppData\Local\Jenkins\.jenkins\workspace\ABCInc\target\SimpleGreeting*.jar"
C:\workspace\dev\ /y
C:\Users\wasadmin\AppData\Local\Jenkins\.jenkins\workspace\ABCInc\target\SimpleGreeting-1.0-
SNAPSHOT.jar
1 File(s) copied
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

__23. In **File Explorer** go to **c:\workspace\dev** and notice SimpleGreeting-*.jar is copied here by the shared library script upon user acceptance.

__24. Close all.

Part 8 - Review

In this lab you utilized Groovy DSL and Shared Library to create an advanced pipeline in Jenkins.

Lab 11 - Configure Jenkins Security

In this lab we will configure Jenkins to authenticate users against its internal user database, and enforce capability limitations on users.

At the end of this lab you will be able to:

1. Enable security and select the internal user database
2. Create users
3. Assign global privileges to users
4. Assign project-specific privileges to users.

Part 1 - Enable Jenkins Security

__1. Make sure Jenkins is started. Since we configured as windows service it will be started every time you start the machine.

__2. Go to the Jenkins console:

`http://localhost:8080`


__3. Enter **wasadmin** as user and password and click **Log in**.


__4. On the Menu click **Manage Jenkins**.



__5. Click **Configure Global Security**.

Security


**Configure Global Security**
Secure Jenkins; define who is allowed to access/use the system.





__6. Jenkins will display the **Configure Global Security** page.


__7. Under the **Security Realm** heading, select **Jenkins own user database** and check the box to **Allow users to sign up**.

Security Realm

☐ Delegate to servlet container 

☒ Jenkins' own user database 

☒ Allow users to sign up 

 **With signup enabled, anyone on your network can become an authenticated user. It is recommended in this case to minimize the permissions granted to any authenticated user.**

__8. Click the **Save** button at the bottom of the page.

At this point, we have enabled security, but we haven't created any users, nor have we actually applied an authorization requirement. Let's go ahead and add a couple of users.

Part 2 - Create an Administrative User

There are two ways to add users: We can do it through Jenkins' management console, or we can allow users to sign up themselves. Let's first use the management console to add an administrative user so we can lock down the security.

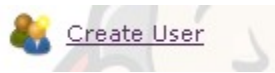
__1. In the “Manage Jenkins” page (the last part of the lab should have left you here), click on **Manage Users**.



Manage Users

Create/delete/modify users that can log in to this Jenkins

__2. Click on **Create User**.



__3. The system displays the **Sign Up** page. Enter the following information in the appropriate fields:

Username:	admin
Password:	password
Confirm Password:	password
Full name:	Administrative User
E-mail address:	admin@localhost.com

Note that the e-mail address is not actually verified to be a valid email address, but Jenkins will reject it unless it is in the usual email format.

__4. When the page looks like below, click on the **Create User** button.

Create User


Username:	<input type="text" value="admin"/>
Password:	<input type="password" value="password"/>
Confirm password:	<input type="password" value="password"/>
Full name:	<input type="text" value="Administrative User"/>
E-mail address:	<input type="text" value="admin@localhost.com"/>

Create User

__5. The system will display the list of current users, including the 'admin' user that you just created.

Users

These users can log into Jenkins. This is a sub set of [this list](#), which some commits on some projects and have no direct Jenkins access.

User Id	
 admin	Administrative User
 wasadmin	Administrator

__6. Click on **Manage Jenkins** to return to the management console.

Part 3 - Enable Authentication

__1. Click on **Configure Global Security**.

__2. Under the **Authorization** heading, select **Project-based Matrix Authorization Strategy**.

Authorization

- ☐ Anyone can do anything
- ☐ Legacy mode
- ☐ Logged-in users can do anything
- ☐ Matrix-based security
- ☒ Project-based Matrix Authorization Strategy



When you click on the radio button above, Jenkins will display a list of global authorizations. We need to enter the 'admin' user here with full permissions, and then we'll add other users to individual projects.

- ___ 3. Click **Add user or group**.
- ___ 4. Enter **admin**, and then click **OK**.

User or group name:

___ 5. Jenkins displays the newly-added user in the list of users. Now we need to select all the permissions. You could click each permission box listed for 'admin' individually, but to save a little time, if you scroll the window horizontally all the way over to the right-hand side, you'll find a button that will select all the permissions in one operation. Find that button and click it.

All check boxes will be selected.

User/group	Overall	Credentials	Agent			Job				Run	View	SCM	Lockable Resources												
	Administer	Read	Create	Delete	Update	View	Build	Configure	Connect	Create	Delete	Discover	Move	Read	Workspace	Delete	Replay	Update	Configure	Create	Delete	Read	Tag	Reserve	Unlock
 Anonymous Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 Authenticated Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 Administrative User	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

- ___ 6. Click **Save**.
- ___ 7. Since we have altered the authorization strategy, Jenkins resets its security system, which requires us to log in again. You will see that wasadmin access has been denied. Click the **log out** link at the upper-right corner of Jenkins' home page.

Administrator | **log out**

- ___ 8. At the login screen, enter **admin** as the userid and **password** as the password, then click **Sign in**.

If for whatever reason, Jenkins doesn't let you log in, check with your instructor – you may need to reset the security system by editing the configuration file manually and then start over with the security setup.

- __9. Do not save the password.
- __10. Click the **log out** link at the upper-right corner of Jenkins' home page.
- Jenkins should display the login page again.

Part 4 - Create a Self-Signed-Up User

When we enabled Jenkins security, we left the checkbox selected for “Allow users to sign up”. As a result, there is a **Create an Account** link on the login page. Let's create a user that way, and then we'll go back and grant them privileges on a project.

- __1. Click on **Create an account** in the login page.



Welcome to Jenkins!

Please sign in below or [create an account](#).

- __2. Jenkins will display the **Sign up** page. Enter the following information in the appropriate fields:

Username:	jane
Full name:	Non-Administrative User
E-mail address:	jane@localhost.com
Password:	password

Note that the e-mail address is not actually verified to be a valid email address, but Jenkins will reject it unless it is in the usual email format.

- __3. Click **Create account**.

__4. Jenkins will display the **Success** window. Do not save the password.

Success

You are now logged in. Go back to [the top page](#).

__5. Click on the link to go to **the top page**.

Since the new user has no permissions, access is denied.

Access Denied

 jane is missing the Overall/Read permission

__6. Click on the **log out** link.

__7. Sign in using 'admin/password'.

__8. We should see the main dashboard page. Click on the **SimpleGreeting** job (we created this job in an earlier lab).

__9. Click **Configure**.

__10. In the configuration page, click on the check box marked **Enable project-based security**, to select it.




☒ Enable project-based security

When you select the check box, Jenkins will display a matrix of users and permissions.

__11. Click **Add user or group**.

__12. Enter **jane**, and click **OK**.

__13. In the row labeled **Non-Administrative User**, select the check boxes for **Discover**, **Read** and **Workspace**.

User/group	Credentials			Job									Run			SCM
	Create	Delete	ManageDomains Update	View	Build	Cancel	Configure	Delete	Discover	Move	Read	Workspace	Delete	Replay	Update	Tag
 Anonymous Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 Authenticated Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
 Non-Administrative User	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

__14. Click **Save**.

__15. We also need to grant jane the 'overall read' permission. Click **Back to Dashboard**.

__16. Click the **Manage Jenkins** link, and then select **Configure Global Security**.

__17. Under the **Authorization** heading, Click **Add user or group**.

__18. Enter **jane**, and click **OK**.

__19. In the row labeled **Non-Administrative User**, select the check box for **Read** under **Overall**.

User/group	Overall	Cre		
	Administer	Read	Create	Delete
Anonymous Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Authenticated Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Administrative User	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Non-Administrative User	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

__20. Click **Save**.

__21. Log out, and then log back in as 'jane/password'.

Note that Jenkins only displays the 'SimpleGreeting' job. This is because Jane only has read and discover access to this project.

__22. Click on the **SimpleGreeting** job.

__23. Notice that Jane doesn't have full privileges – there is no **Build Now** or **Configure** option on the project.



__24. Log out of Jenkins.

Next we will give full permission to the user wasadmin.

__25. Login back as admin/password.

__26. Click **Manage Jenkins**.

__27. Click on the link for **Configure Global Security**.

__28. Under the **Authorization** heading, Click **Add user or group**.

__29. Enter **wasadmin**, and click **OK**.

__30. Jenkins displays the newly-added 'Administrator' user in the list of users. Now we need to select all the permissions. You could click each permission box listed for 'Administrator' individually, but to save a little time, if you scroll the window horizontally all the way over to the right-hand side, you'll find a button that will select all the permissions in one operation. Find that button and click it.

All check boxes will be selected.

User/group	Overall	Credentials	Agent				Job				Run	View	SCM	Lockable Resources					
	Administer	Create Read	Delete Manage Domains	Update View	Configure Build	Connect	Create Delete	Disconnect Delete	Build Cancel	Configure Cancel	Create Delete	Discover Delete	Move Read	Workspace Read	Delete Replay	Update Configure	Create Delete	Read Tag	Reserve Unlock
Anonymous Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Authenticated Users	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Administrative User	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Non-Administrative User	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Administrator	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Add user or group...																			

- __31. Click **Save**.
- __32. Log out.
- __33. Sign in back as wasadmin and make sure you have full access.
- __34. Close all.

Part 5 - Review

In this lab, we went through a series of steps to enable and configure Jenkins security. We saw how to create users administratively, and how to configure users who sign up using the self-sign-up screen.