

**WA2893 Booz Allen Hamilton Tech
Excellence Modern Software
Development Program - Phase 2**

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Using the GitFlow Workflow.....	4
Lab 2 - Monolith vs Microservices Design.....	18
Lab 3 - A Simple RESTful API in Spring Boot.....	22
Lab 4 - Use the Spring JDBCTemplate under Spring Boot.....	34
Lab 5 - Use the Spring Data JPA under Spring Boot.....	41
Lab 6 - Create a RESTful API with Spring Boot.....	48
Lab 7 - Create a RESTful Client with Spring Boot.....	62
Lab 8 - Enable Basic Security.....	67
Lab 9 - Configure Tools in Jenkins.....	71
Lab 10 - Create a Jenkins Job.....	75
Lab 11 - Create a Pipeline.....	89
Lab 12 - OPTIONAL: Advanced Pipeline with Groovy DSL.....	100

Environment. Read this section before starting Lab 1

This course requires a Windows environment to do the Labs in this document and an Ubuntu VM to do the project using instructions in another document.

The name of the VM's (Virtual Machines) are as follows:

- WA2893-REL_9_1 (Windows VM)
- Project-VM-4.0-2023-01-06(Ubuntu VM)

To do the Labs in this document you only need the Windows VM that we will call it as WA2893-REL_9_1 but your remote computer name may differ.

You should receive an email with instruction on how to connect to both VM's, please contact your instructor if you have any doubt on what VM should be used to do the Labs and to do the project.

Before starting please connect to the WA2893-REL_9_1 (Windows VM).

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'.

```
C:\Workspace\gitflow_test\local_repo>git status
On branch feature/HOME_PAGE
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.html

nothing added to commit but untracked files present (use "git add" to track)
```

__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.

```
C:\Workspace\gitflow_test\local_repo>git status
On branch feature/HOME_PAGE
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   index.html
```


__7. Commit changes:

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

```
C:\Workspace\gitflow_test\local_repo>git commit -m "Added index.html"
[feature/HOME_PAGE c1d089d] Added index.html
1 file changed, 4 insertions(+)
create mode 100644 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.

```
C:\Workspace\gitflow_test\local_repo>git ls-remote
From C:/Workspace/gitflow_test/remote_repo.git
11df4d384b7349504203b219bd1e53e1564aa277 HEAD
29ec16fead014e06c24dde5cae6e0cb659e0f60 refs/heads/develop
096c04306274087789ad55cbe17427e6c15ae6cf refs/heads/feature/HOME_PAGE
11df4d384b7349504203b219bd1e53e1564aa277 refs/heads/master
```

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>
```

```
C:\Workspace\gitflow_test\local_repo>git tag  
REL_1.0  
REL_2.0
```

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.

```
C:\Workspace\gitflow_test\local_repo>git tag
REL_1.0
REL_2.0
REL_2.0.1
```

__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.

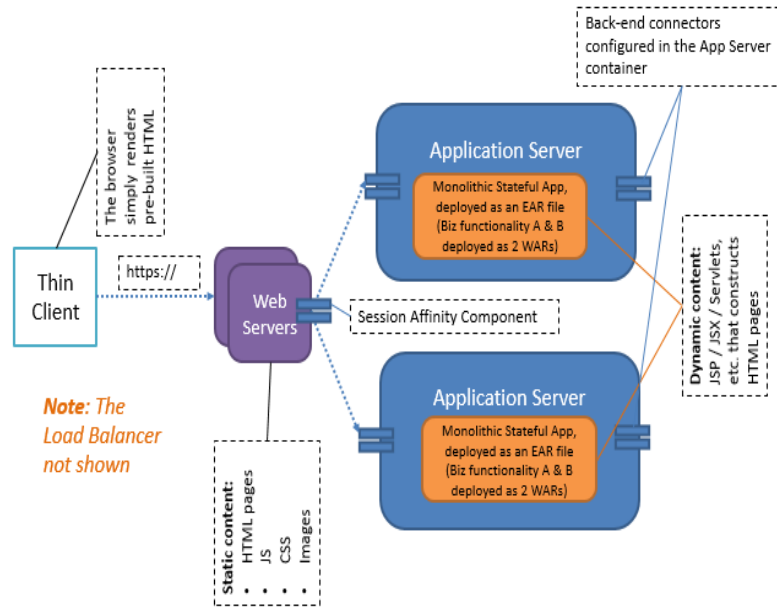


Fig 1. Application Architecture 1: Traditional (Java) Enterprise Application Architecture Example.

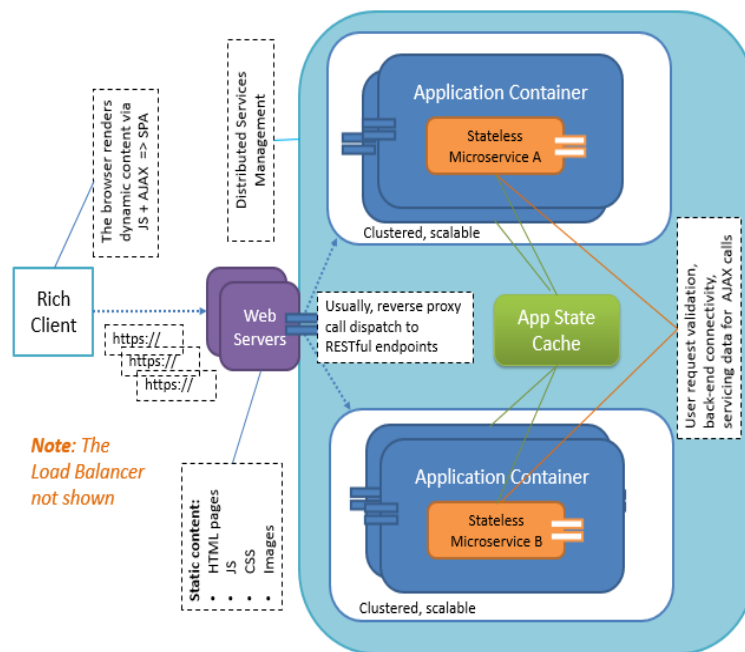


Fig. 2 Application Architecture 2: Microservices-based Architecture Example.

- ___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 - A Simple RESTful API in Spring Boot

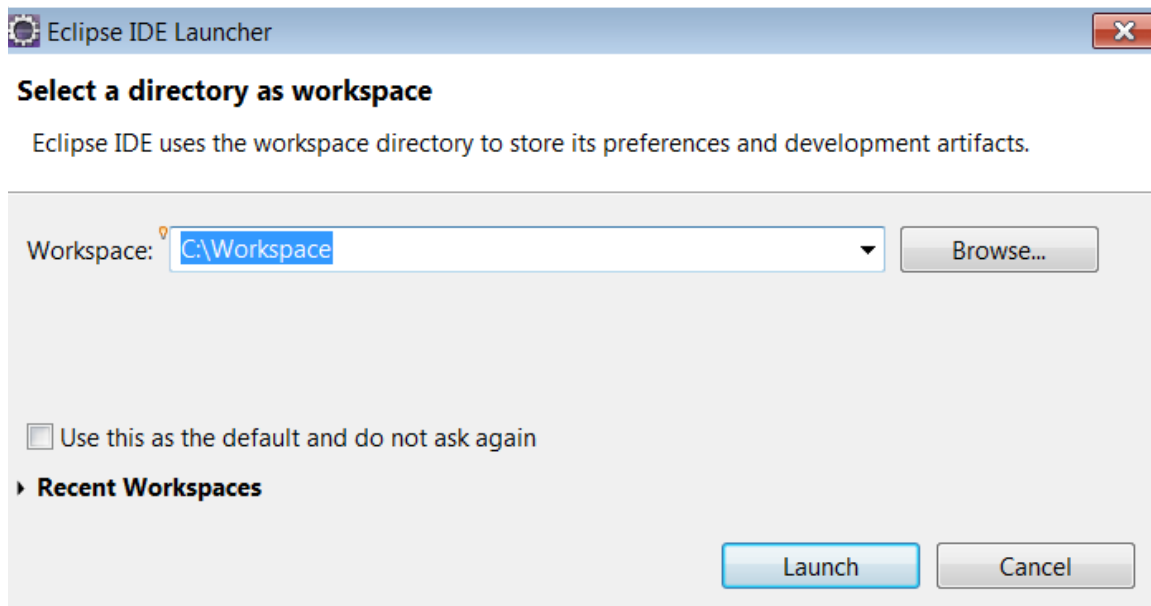
In this lab we're going to build a simple "Hello World" API using Spring Framework and Spring Boot. The API will implement a single resource, "/hello-message" that returns a JSON object that contains a greeting.

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**.



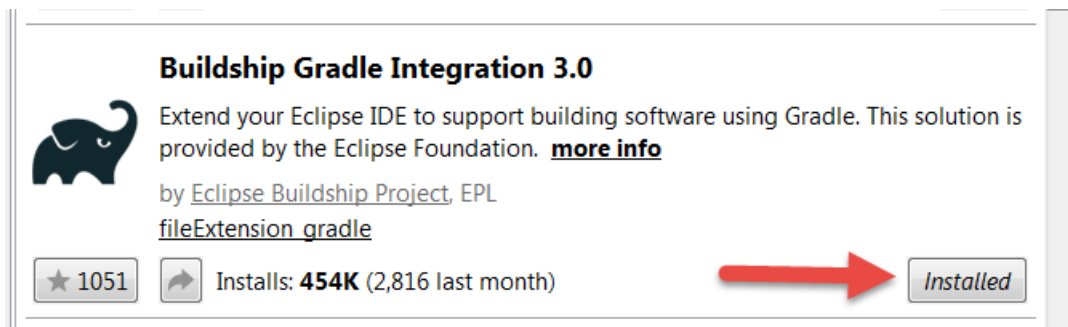
__3. Close the **Welcome** panel if open.

We need to update gradle.

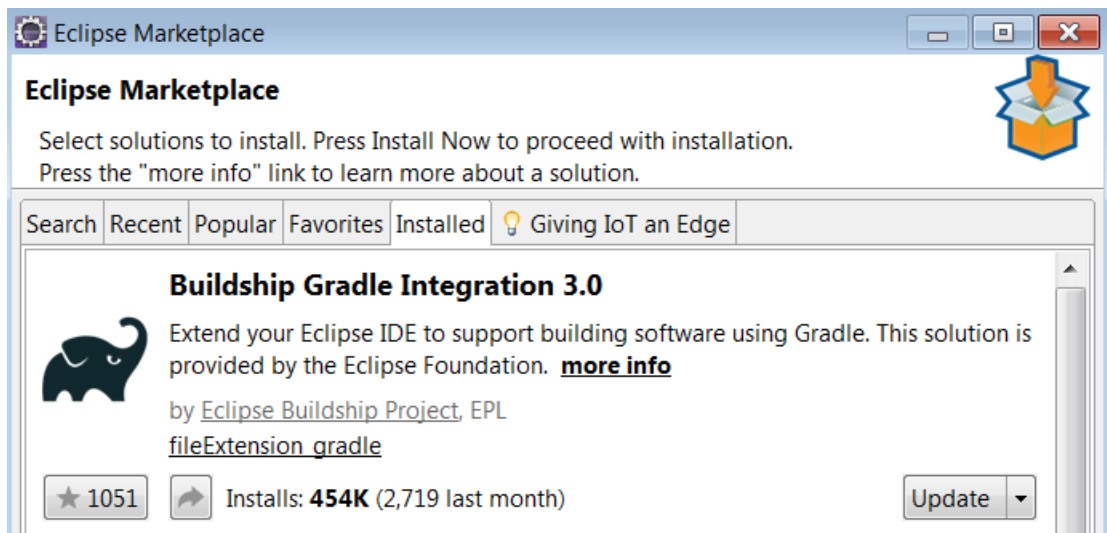
__4. From the menu, select **Help > Eclipse Marketplace**.

__5. In the Find box, type **gradle** and hit enter.

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



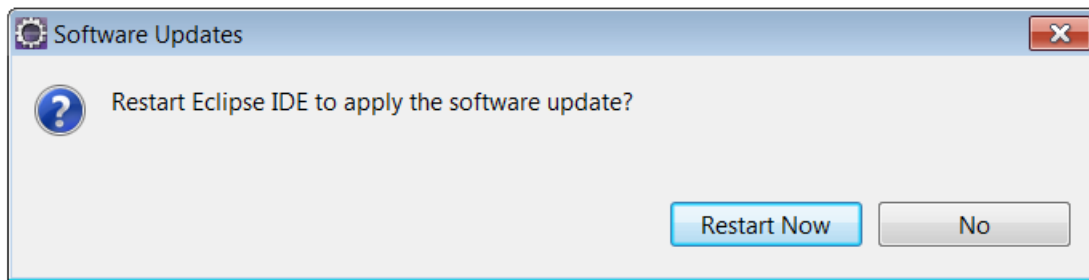
__7. This will take you to the **Installed** tab, locate **Buildship Gradle Integration 3.0** and click the **Update** button.



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

Wait for the installation to be completed.

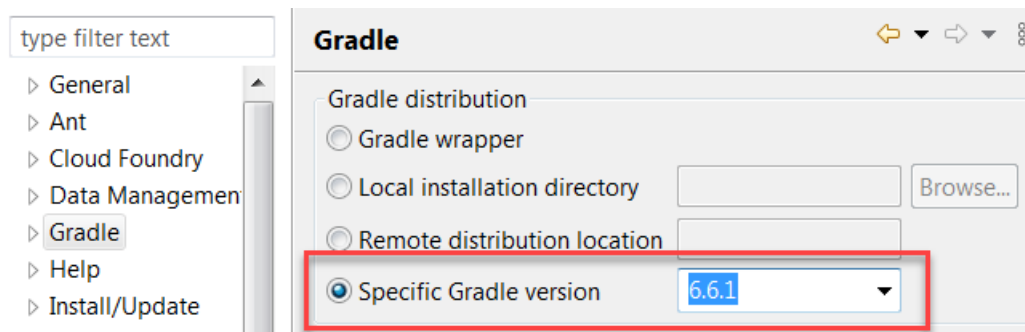
__9. A dialog will open when the updates are installed, click Restart Now.



We are ready to work with Gradle.

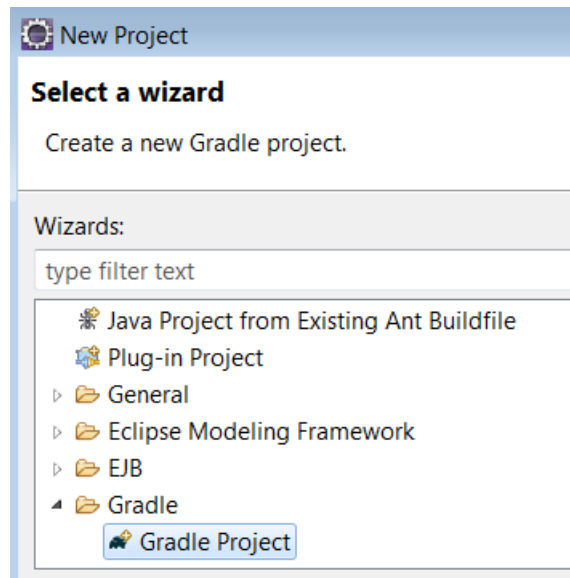
Part 2 - Create a Gradle Project

- __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.



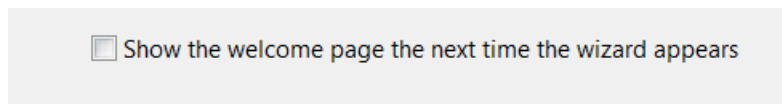
- __4. Click **Apply and Close**.
- __5. From the main menu, select **File** → **New** → **Project...**

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

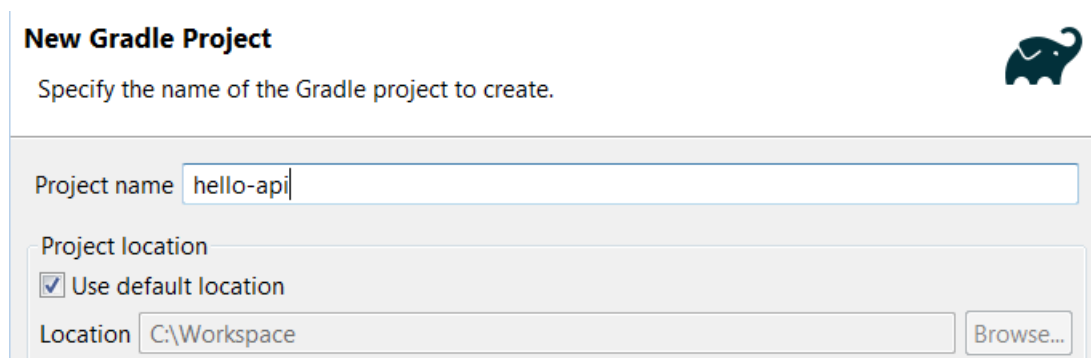


__7. Click **Next**.

__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**.

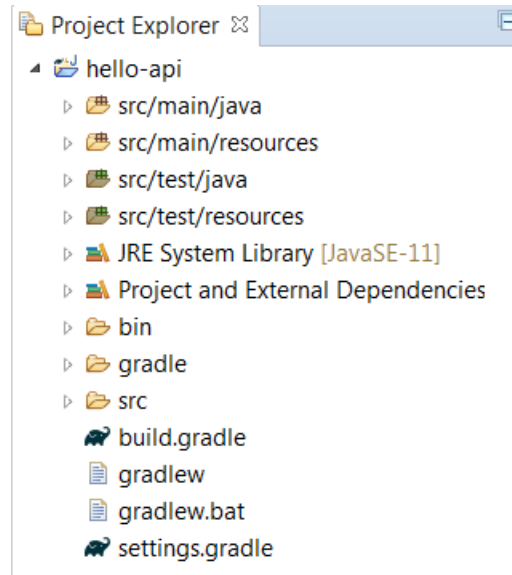


If necessary, wait for Eclipse to finish any background process.

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**.

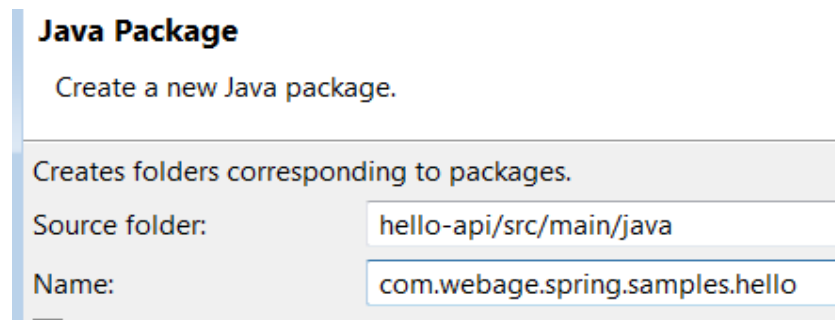
__ 11. Make sure the **Markers** tab doesn't show any error.

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.

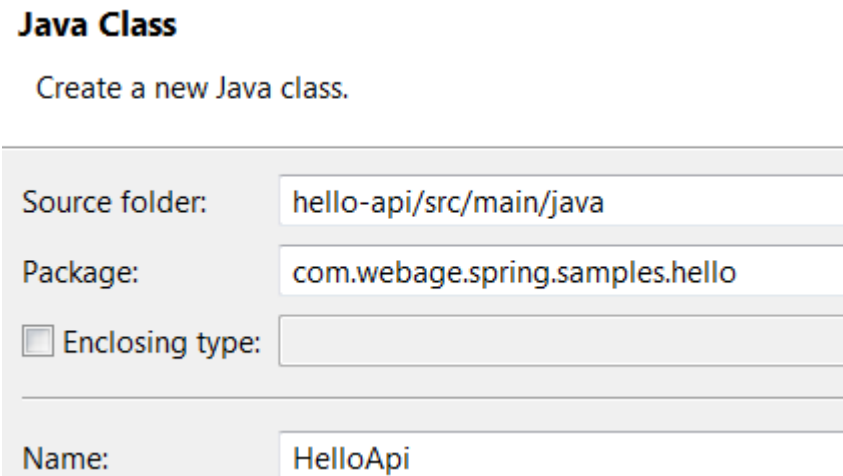
Creates folders corresponding to packages.

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:

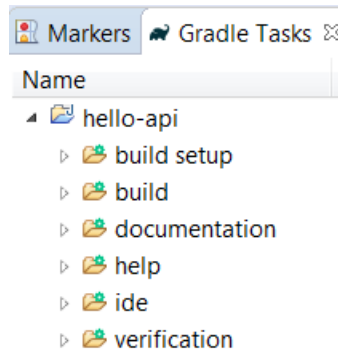
```
@SpringBootApplication  
public class HelloApi {
```

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

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

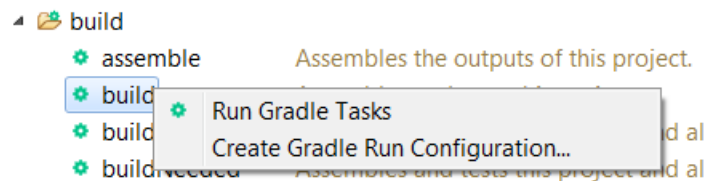
__7. The editor is probably showing errors due to missing 'import' statements. Press **Ctrl-Shift-O** to organize the imports.

- __ 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.

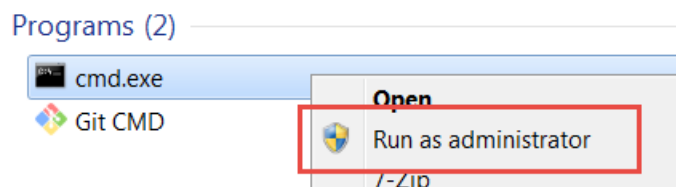
Note. If fails building try again and the second time should works.

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\Jenkins  
C:\Program Files\Jenkins>jenkins stop  
2022-03-30 15:44:26,244 INFO - Stopping the service with id 'jenkins'  
2022-03-30 15:44:26,272 INFO - The service with ID 'jenkins' is not running
```

__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**.

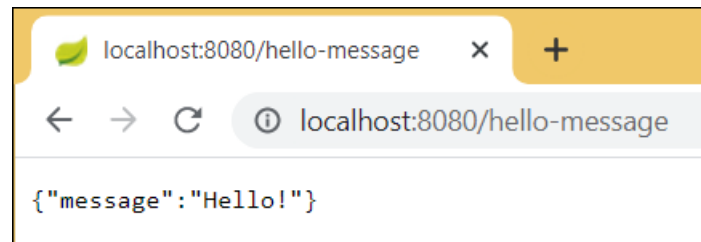
__7. Watch the **Console** panel. At the bottom of it, you should see a message indicating that the **HelloApi** program has started successfully:

```
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:

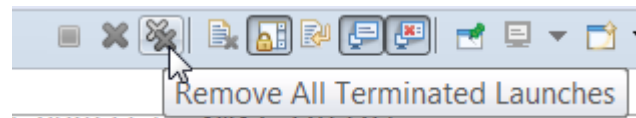


Notice that the response is in the form of a JSON object whose structure matches the 'HelloResponse' class contents.

__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 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).

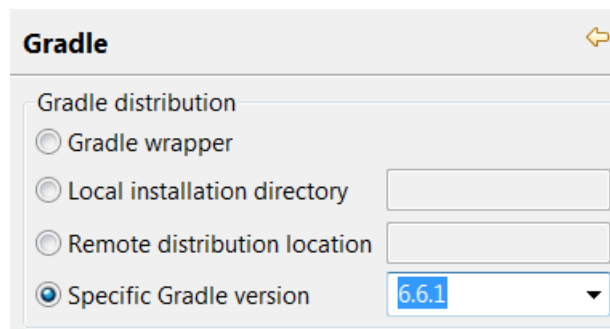
Lab 4 - Use the Spring JDBCTemplate under Spring Boot

Any of Spring's data access techniques can be used with Spring Boot. For convenience, Spring Boot sets up an embedded database by default, which you can override with an external database later on. This is useful for testing and early development on an application.

In this lab you will implement a Data Access Object by creating SQL queries and issuing them with a JDBCTemplate that is autowired by Spring.

Part 1 - Import the Starter Project

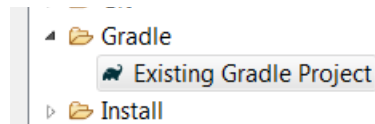
- __1. In eclipse, from the menu, select **Window > Preferences**.
- __2. From the left menu, select **Gradle**.
- __3. Make sure **Specific Gradle version** is set to **6.6.1**, if not change it and click **Apply and Close**.



- __4. Extract the contents of **C:\LabFiles\Gradle\Spring-Boot-JDBC-Starter.zip** to **C:\Workspace** directory without creating extra subdirectories.
- __5. Verify the following folder was created:

C:\Workspace\spring-boot-jdbc2-starter

__6. From Eclipse's main menu, select **File** → **Import** and select **Gradle** → **Existing Gradle Project**, and then click **Next**.



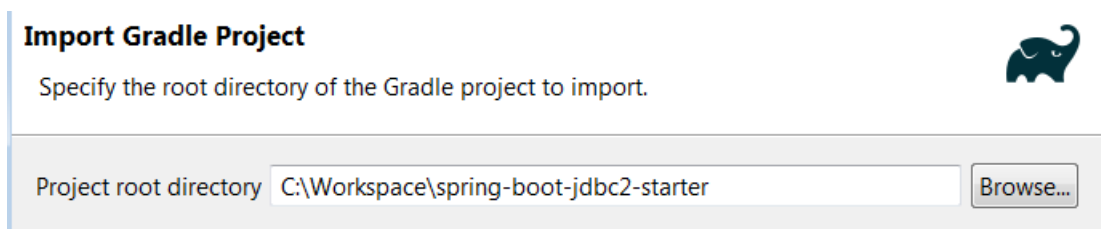
__7. If you see the Welcome page, uncheck the box and click **Next**.

☐ Show the welcome page the next time the wizard appears

__8. Click **Browse** button.

__9. Select **C:\Workspace\spring-boot-jdbc2-starter** and click **Select Folder**.

__10. Make sure you have the right project as shown below:



__11. Click **Finish**.

Wait until the project is imported.

Part 2 - Enable the Default Embedded Database

Spring will auto-configure a database for us, and automatically load a data set if necessary. All we need to do is give it the correct setup information.

__1. In the **Project Explorer**, locate the **build.gradle** file for **spring-boot-jdbc2-starter** project and double-click it to open it.

__2. Add the following 2 dependencies into the **dependencies** block:

```
compile group: 'org.springframework.boot', name: 'spring-boot-starter-jdbc', version: '2.1.8.RELEASE'
compile group: 'org.hsqldb', name: 'hsqldb', version: '2.3.2'
```

__3. Save the file.

__4. Right-click the **spring-boot-jdbc2-starter** project and click **Gradle > Refresh Gradle Project**.

__5. Switch to **Gradle Tasks** pane.

__6. Expand **spring-boot-jdbc2-starter > build**.

__7. Right-click on **build** and click **Run Gradle tasks**.

__8. Switch to **Console** pane and ensure there are no error messages.

Part 3 - Configure the JDBCTemplate

Because we added the dependency for HSQLDB to 'build.gradle', Spring Boot will automatically create a DataSource object for an embedded, memory-based instance of HSQLDB. Also, when the application starts up, Spring Boot will initialize the schema of the embedded database using the 'schema.sql' file that is in the classpath, and then run the 'data.sql' file that's also in the classpath.

To use the database from our Data Access Object, we'll need to configure a JDBCTemplate object that can be injected into the DAO. We'll do that by creating a class and annotating it with '@Configuration'. This is an example of Spring's 'Java-based configuration' mechanism. Spring will find the annotated class in the classpath and use it to instantiate the associated beans.

- ___ 1. In the **Project Explorer**, expand **spring-boot-jdbc2-starter > src/main/java**.
- ___ 2. Right-click on the **com.webage.dao** package and select **New → Class**.
- ___ 3. Enter **DAOConfig** as the new class name and then click **Finish**.

Java Class

Create a new Java class.

Source folder:	spring-boot-jdbc2-starter/src/main/java
Package:	com.webage.dao
<input type="checkbox"/> Enclosing type:	
Name:	DAOConfig

- ___ 4. Add the annotation '@Configuration' to the class, as shown below:

```
@Configuration
public class DAOConfig {
```

- ___ 5. Organize the imports by pressing **Ctrl-Shift-O**.
- ___ 6. Inside the class, create a method 'jdbcTemplate()' as shown below:

```
@Bean
JdbcTemplate jdbcTemplate(DataSource ds) {
    return new JdbcTemplate(ds);
}
```

- ___ 7. Organize the imports by pressing **Ctrl-Shift-O**. Select **javax.sql.DataSource**, if prompted.
- ___ 8. Save the file.

The method we added defines a bean called 'jdbcTemplate' that can be injected into our DAO class.

Part 4 - Complete the DAO Class

Now that we have the configuration complete, all we need to do is flesh out the implementation of the DAO class to actually perform the query. We'll use some convenience classes provided by the Spring Framework to make this relatively painless.

__1. Locate the class called **JDBCPurchaseDAO** in the **com.webage.dao** package. Double-click on the class to open it.

__2. Look for the method that currently looks like below:

```
@Override
public Collection<Purchase> getAllPurchases() {
    // Replace this statement with the call to jdbcTemplate.
    return null
}
```

__3. Replace the line 'return null;' with a call to jdbcTemplate, so that the method appears as below:

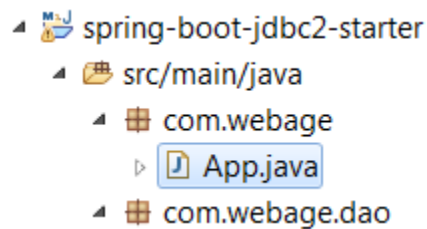
```
@Override
public Collection<Purchase> getAllPurchases() {
    // Replace this statement with the call to jdbcTemplate.
    return jdbcTemplate.query("Select * from PURCHASE", new
        BeanPropertyRowMapper<Purchase>(Purchase.class));
}
```

Here we're using Spring's 'BeanPropertyRowMapper' class to automatically map the database rows to instances of the 'Purchase' class based on the column names. If you look in the 'schema.sql' file, you'll notice that the column names match the property names that are used in the 'Purchase' class. That correspondence allows us to use this convenience class.

__4. Save all files by pressing **Ctrl-Shift-S**.

Part 5 - Compile and Test

- ___ 1. Switch to **Gradle Tasks** pane.
- ___ 2. Expand **spring-boot-jdbc2-starter > build**.
- ___ 3. Right-click **build** and click **Run Gradle tasks**.
- ___ 4. Switch to **Console** pane and ensure there are no error messages.
- ___ 5. Expand **src/main/java > com.webage**
- ___ 6. Right click **App.java** and select **Run As > Java application**.



- ___ 7. Wait until you see this message:

```
main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
main] o.s.b.a.w.s.WelcomePageHandlerMapping : Adding welcome page template: index
main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path
main] com.webage.App : Started App in 5.238 seconds (JVM running for 6.207)
```

- ___ 8. Open a browser and enter the following url:

http://localhost:8080/browse

__9. You should see the results of our database query.

Browse Purchases

[Add Purchase](#)

	Purchase Id	Customer Name	Purchase Date	Product
Edit	1	Bruce	May 12, 2010	Mountain Bike
Edit	2	Paul	Apr 30, 2010	Football
Edit	3	Rick	Jun 5, 2010	Kayak

[Back to Main Menu](#)

Copyright 2011 SportyStuf Inc. Current date/time is Tue Dec 06 13:55:53 EST 2016

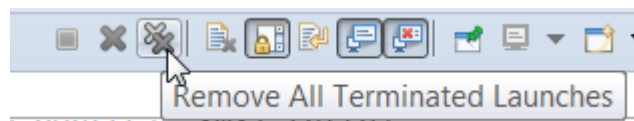
__10. Close the browser.

__11. Close all open files.

__12. Click on the red stop button in the console panel to stop the Spring Boot application:



__13. Click **Remove All Terminated Launches**.



Part 6 - Review

We used the convenient embedded database setup in this lab to demonstrate the use of the Spring JdbcTemplate to carry out a query against a SQL DataSource.

Lab 5 - Use the Spring Data JPA under Spring Boot

Java Persistence Architecture, or JPA simplifies data access by automatically generating SQL queries to manage the storage and retrieval of Java objects. Spring Data takes that idea one step farther to automatically generate data access or "Repository" classes for Java objects. All we need to do is make sure our Java objects are annotated correctly to contain the additional metadata required for database storage. Also, of course, we need to setup the software infrastructure.

In this lab you will annotate a set of domain objects and then use Spring Data to implement a storage repository for those objects.

Part 1 - Import the Starter Project

__ 1. Extract the contents of **C:\LabFiles\Gradle\Spring-Boot-JPA-Starter.zip** to **C:\Workspace** directory.

__ 2. Verify the following folder was created:

C:\Workspace\spring-boot-jpa

__ 3. From Eclipse's main menu, select **File** → **Import** and select **Gradle** → **Existing Gradle Project**, and then click **Next**.

__ 4. Click **Browse** button.

__ 5. Select **C:\Workspace\spring-boot-jpa** and click **Select Folder**.

Import Gradle Project

Specify the root directory of the Gradle project to import.



Project root directory

__ 6. Click **Finish**.

Part 2 - Examine the Starter Project

__ 1. Expand the **spring-boot-jpa** project.

__ 2. Open **build.gradle**.

There are a few items already setup in the starter project that you should take note of for your own projects:

- 'build.gradle' contains two dependencies that are particularly important to this project:

```
compile group: 'org.springframework.boot', name: 'spring-boot-starter-web', version: '2.1.8.RELEASE'
compile group: 'org.springframework.boot', name: 'spring-boot-starter-thymeleaf', version: '2.1.8.RELEASE'
compile group: 'org.hsqldb', name: 'hsqldb', version: '2.4.1'
compile group: 'org.springframework.boot', name: 'spring-boot-starter-data-jpa', version: '2.1.8.RELEASE'
```

- These dependencies pull in the components required for Spring Boot to implement the Spring Data framework, and also to use the embedded HSQLDB instance for testing.
- In 'src/main/resources', there are two files, 'data.sql' and 'schema.sql', which establish the database schema and initial data load for the embedded database that we'll use for testing. These files end up being assembled to the root of the classpath, where Spring Boot can load them at startup.
- Also in 'src/main/resources', there is a file called 'application.properties'. This file contains one line:

```
spring.jpa.hibernate.ddl-auto=none
```

- The line disables Hibernate's automatic schema generation, which would overwrite our test data if we left it enabled.
- The starter project contains a pair of domain classes that we will use as a starting point for our repository.
- The starter project also contains a rudimentary user interface for our demonstration program.

Part 3 - Annotate the Domain Classes

We have a pair of domain classes, 'Customer' and 'Purchase' that are meant to model a set of purchases that might be made through an online store of some kind. There is a "Many-to-One" relationship between these classes; many purchases might be made by one customer. We haven't modeled the reverse relationship, although that might also be useful.

- __ 1. Expand **spring-boot-jpa** > **src/main/java** > **com.webage.domain**.
- __ 2. Open the **Customer** class.
- __ 3. Add the '@Entity' and '@Table' annotations to the class as shown below. These annotations designate that the class should be managed by JPA and tell JPA what database table name to use for the class:

```
@Entity
@Table (name="CUSTOMERS")
public class Customer {
```

- __ 4. Add annotations to the 'id' field so it appears as below. These annotations mark the 'id' field as the primary key of 'Customer' and tell JPA how it's going to be generated:

```
@Id
@GeneratedValue (strategy=GenerationType.IDENTITY)
long id;
```

- __ 5. The 'name' field in 'Customer' is modeled by a column called 'CUSTOMER_NAME' in the database table that we are using. So, the default mapping of field name to column name will not work in this case. Annotate the field as shown below to override the default name mapping:

```
@Column (name="CUSTOMER_NAME")
String name;
```

- __ 6. Organize the imports by pressing **Ctrl-Shift-O**. Select **javax.persistence.Entity**, **javax.persistence.Table** and **javax.persistence.Id**.
- __ 7. Save the file.
- __ 8. Open the **Purchase** class inside 'com.webage.domain'.

- __ 9. Add the '@Entity' and '@Table' annotations to the class as shown below. These annotations designate that the class should be managed by JPA and tell JPA what database table name to use for the class:

```
@Entity
@Table (name="PURCHASES")
public class Purchase {
```

__10. Add annotations to the 'id' field so it appears as below. These annotations mark the 'id' field as the primary key of 'Customer' and tell JPA how it's going to be generated.

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

__11. Annotate the 'customer' field as shown below to tell JPA that it should load the value from an associated table.

```
@ManyToOne
private Customer customer;
```

__12. Organize the imports by pressing **Ctrl-Shift-O**. Select **javax.persistence.Entity**, **javax.persistence.Table** and **javax.persistence.Id**.

__13. Save the file.

We now have a set of domain objects that should be recognized as JPA Entities that can be managed by JPA.

Part 4 - Create the Repository Interfaces

This is where we see the magic of the Spring Data JPA framework!

The starter project has two interfaces declared in the 'com.webage.repository' package. Spring Data uses these interfaces to automatically generate classes that implement the repository functionality. Let's have a look at one of them.

__1. Open the interface called **PurchaseRepository** in the **com.webage.repository** package. For convenience, the contents are reproduced below:

```
package com.webage.repository;

import org.springframework.data.repository.CrudRepository;

import com.webage.domain.Purchase;

public interface PurchaseRepository extends CrudRepository<Purchase,
Long> {

}
```

As you can see, there's really not much to this repository interface. It extends a standard interface called 'CrudRepository' that includes standard Create, Retrieve, Update and Delete methods. It uses Java generic type definitions to indicate what data type the repository holds, and the data type of the primary identifier field.

That information is enough for Spring Data to fully implement this interface. We don't need to write any access code at all!

If you take a look at the 'CustomerRepository' interface you'll find a similar declaration based on the Customer domain class.

Part 5 - Observe the Usage

__1. Open the **PurchaseServiceImpl** class in the **com.webage.service** package. For convenience, the contents are shown below:

```
package com.webage.service;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.webage.domain.Purchase;
import com.webage.repository.PurchaseRepository;

@Service
public class PurchaseServiceImpl implements PurchaseService {
    @Autowired
    private PurchaseRepository repo;

    public void savePurchase(Purchase purchase) {
        repo.save(purchase);
    }

    public Iterable<Purchase> findAllPurchases() {
        return repo.findAll();
    }

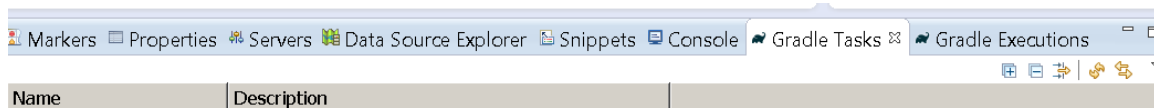
    public Optional<Purchase> findPurchaseById(long id) {
        return repo.findById(id);
    }
}
```

Notice the **@Autowired** field for the **PurchaseRepository**. Spring Data will automatically create a class that implements the **PurchaseRepository** interface and plug it in to the instance of **PurchaseServiceImpl**.

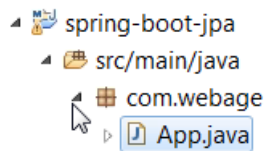
Notice that the service calls the methods 'save(...)', 'findAll()' and 'findOne(...)' on the repository interface. These methods will be implemented by Spring Data's automatic proxy, so as to provide the expected functionality.

Part 6 - Compile and Test

- __1. Right click **spring-boot-jpa** and select **Gradle** → **Refresh Gradle Project**.
- __2. Switch to **Gradle Tasks** pane.



- __3. Expand **spring-boot-jpa** > **build**.
- __4. Right click **build** and click **Run Gradle tasks**.
- __5. Ensure there are no errors in **Console** pane.
- __6. Run the 'App' class as **Java application**.



- __7. Open a browser and enter the following url:

http://localhost:8080/browse

__ 8. You should see the results of our database query.

Browse Purchases

[Add Purchase](#)

	Purchase Id	Customer Name	Purchase Date	Product
Edit	1	Bruce	May 12, 2010	Mountain Bike
Edit	2	Paul	Apr 30, 2010	Football
Edit	3	Rick	Jun 5, 2010	Kayak

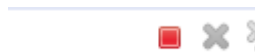
[Back to Main Menu](#)

Copyright 2011 SportyStuf Inc. Current date/time is Tue Dec 06 13:55:53 EST 2016

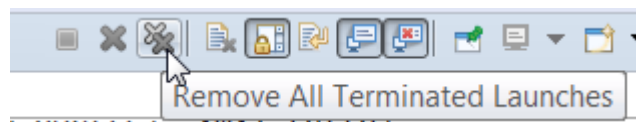
__ 9. Close the browser.

__ 10. Close all open files.

__ 11. Click on the red stop button in the console panel to stop the Spring Boot application:



__ 12. Click **Remove All Terminated Launches**.



Part 7 - Review

In this lab, we used Spring Data's functionality combined with Spring JPA to implement a data repository very rapidly, with a minimum of effort.

Lab 6 - Create a RESTful API with Spring Boot

In this lab you will use Spring Boot to implement a RESTful API for a repository of customers, similar to what you'd need for an online store.

Part 1 - Import the Starter Project

__1. Extract the contents of **C:\LabFiles\Gradle\Spring-Boot-REST-Starter.zip** to **C:\Workspace** directory.

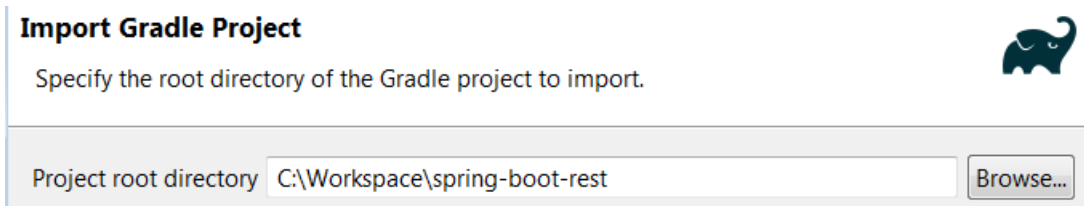
__2. Verify the following folder was created:

C:\Workspace\spring-boot-rest

__3. From Eclipse's main menu, select **File** → **Import** and select **Gradle** → **Existing Gradle Project**, and then click **Next**.

__4. Click **Browse** button.

__5. Select **C:\Workspace\spring-boot-rest** and click **Select Folder**.



__6. Click **Finish**.

Part 2 - Examine the Starter Project

The starter project implements a JPA-based Repository for Customer domain objects using the Spring Data components. If you've just finished the Spring JPA lab, you'll already be familiar with the project.

There are a few items already setup in the starter project that you should take note of for your own projects.

- **build.gradle** contains two dependencies that are particularly important to this project:
compile group: 'org.hsqldb', name: 'hsqldb', version:'2.4.1'
compile group: 'org.springframework.boot', name: 'spring-boot-starter-data-jpa', version:'2.1.8.RELEASE'

- These dependencies pull in the components required for Spring Boot to implement the Spring Data framework, and also to use the embedded HSQLDB instance for testing.
- In **src/main/resources**, there are two files, **data.sql** and **schema.sql**, which establish the database schema and initial data load for the embedded database that we'll use for testing. These files end up being assembled to the root of the classpath, where Spring Boot can load them at startup.
- Also in **src/main/resources**, there is a file called **application.properties**. This file contains one line:

```
spring.jpa.hibernate.ddl-auto=none
```

- The line disables Hibernate's automatic schema generation, which would overwrite our test data if we left it enabled.
- The starter project contains a pair of domain classes that we will use as a starting point for our repository.
- The starter project also contains a rudimentary user interface for our demonstration program.

- ___ 1. Right click **spring-boot-rest** and select **Gradle → Refresh Gradle Project**.
- ___ 2. Switch to **Gradle Tasks** pane.
- ___ 3. Expand **spring-boot-rest > build**.
- ___ 4. Right click **build** and click **Run Gradle tasks**.
- ___ 5. Ensure there are no errors in **Console** pane.

Part 3 - Create the Customer API Class

Although we're running under the Spring Boot environment, the basic technology for creating RESTful services is contained in the Spring MVC framework. To create an API under this framework, we will create one or more classes that have methods to serve out responses to HTTP requests. In this particular case, there is no real business logic involved in the API, so the methods will essentially delegate all their work to the Repository class. As such, the methods are short, and a full implementation for the four core HTTP methods will be small enough to fit into one API class. We'll implement GET, POST, and PUT methods (we'll assume that we never delete a customer).

- ___ 1. In the **Project Explorer**, locate the **spring-boot-rest > src/main/java** node.
- ___ 2. Right-click on **src/main/java** and select **New → Package**.

___3. Enter **com.webage.api** as the package name and then click **Finish**.

Java Package	
Create a new Java package.	
Creates folders corresponding to packages.	
Source folder:	spring-boot-rest/src/main/java
Name:	com.webage.api

___4. Right-click on the newly-created package and select **New** → **Class**.

___5. Enter **CustomerAPI** as the class name and then click **Finish**.

Java Class	
Create a new Java class.	
Source folder:	spring-boot-rest/src/main/java
Package:	com.webage.api
<input type="checkbox"/> Enclosing type:	
Name:	CustomerAPI

___6. We need to flag this class as a resource controller and assign a URL path for it. To do that, add annotations to the class as shown below:

```
@RestController
@RequestMapping("/customers")
public class CustomerAPI {
```

These annotations will make this resource controller serve out the "/customers" resource path.

__7. Inside the body of the class, add an '@Autowired' reference to the 'CustomersRepository' implementation, as shown below:

```
public class CustomerAPI {  
  
    @Autowired  
    CustomersRepository repo;  
}
```

Part 4 - Implement a GET method

First, let's implement the **GET** method against the **/customers** resource path. This should return all the customers that are in the repository.

__1. In the body of the 'CustomerAPI' class, add the following method. It will use the repository proxy to retrieve an 'Iterable' object that iterates through all the customers:

```
public Iterable<Customer> getAll() {  
    return repo.findAll();  
}
```

__2. We want this method to respond to a 'GET' request on the '/customers' path. Since the API class is already annotated with the required path, all we need to do is flag this class as a 'GET' method. To do so, add the '@GetMapping' annotation to the method as shown below:

```
@GetMapping  
public Iterable<Customer> getAll() {
```

__3. Organize the imports by pressing **Ctrl-Shift-O**.

__4. Save the file.

Part 5 - Test the GET Method

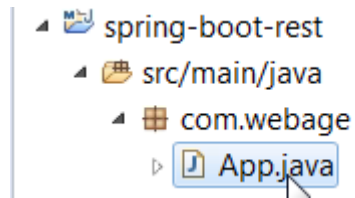
__1. Switch to **Gradle Tasks** pane.

__2. Expand **spring-boot-rest > build**.

__3. Right click **build** and click **Run Gradle tasks**.

__4. Ensure there are no errors in **Console** pane.

__5. Run the '**App**' class as **Java Application**.



The system should start up without errors.

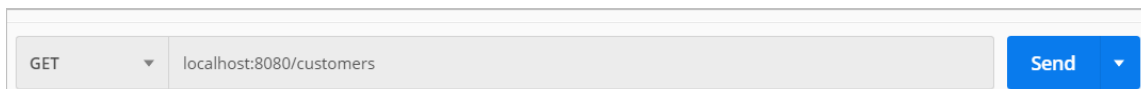
__6. From the Start menu, find and open **Postman**.

Do not update if prompt.

__7. Click the + to add a new tab.

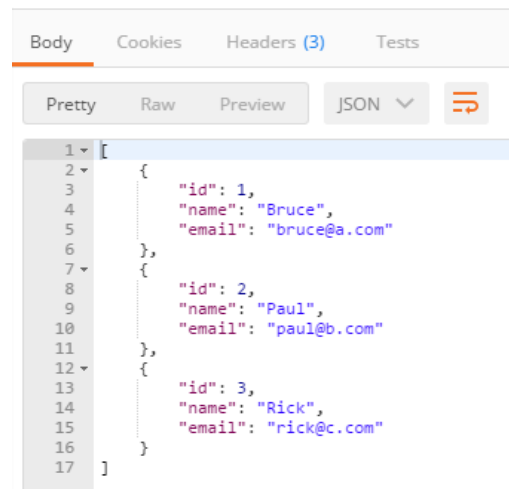


__8. Select the **GET** request using the drop-down box, and enter **localhost:8080/customers** as the URL:



__9. Click the **Send** button.

The result should look like below (you may need to select the 'Body' tab to see the result):



So far, so good! Leave **Postman** open. We'll be using it for the next test.

Part 6 - Lookup a Specific Customer by ID

Looking up a particular customer could be modeled as a "GET" request to a path like `"/customers/2"`, where **2** is the customer ID.

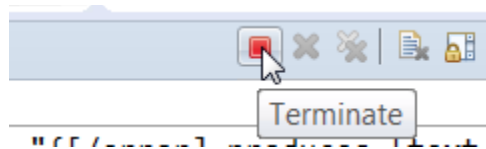
__1. Back in eclipse, in the body of the **CustomerAPI** class, add the following method:

```
@GetMapping("/{customerId}")
public Optional<Customer> getCustomerById(@PathVariable("customerId")
long id) {
    return repo.findById(id);
}
```

__2. Organize imports. Select **java.util.Optional**.

__3. Save the file.

__4. If you still have 'App.java' running from the previous test, stop it using the red stop button in the console window.



__5. Switch to **Gradle Tasks** pane.

__6. Expand **spring-boot-rest** > **build**.

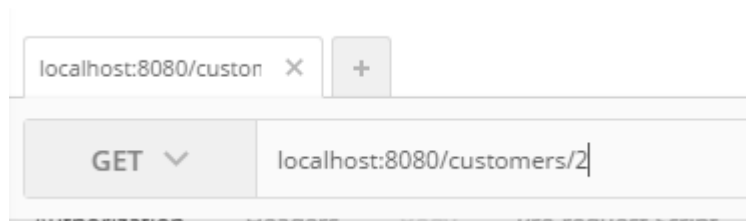
__7. Right click **build** and click **Run Gradle tasks**.

__8. Ensure there are no errors in **Console** pane.

__9. Run **App.java** as Java Application.

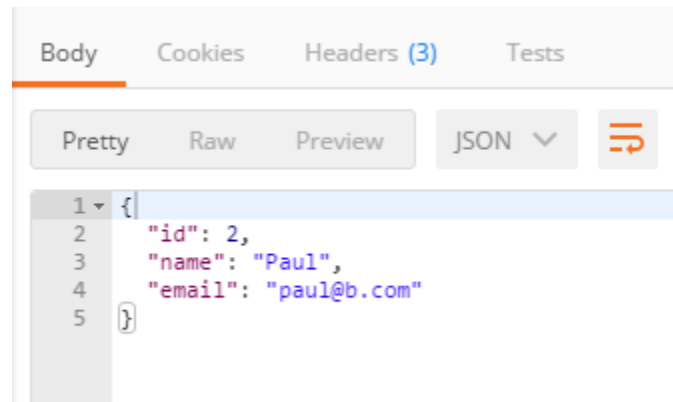
Wait until the app is started.

__10. In the **Postman** tool, add an identifier to the end of the URL:



__11. Click **Send**.

__12. We should get a single result:



__13. Once again, leave **Postman** open, but stop 'App.java'.

__14. You may see a message when stopping the process, just click OK.

Part 7 - Add a POST Method Handler

The POST method is intended to add an entity to a collection of entities. For instance, adding a new customer to the set of customers. This action is accurately modeled as a "POST" to the "/customers" path.

__1. In the '**CustomerAPI**' class, add the following method.

```
@PostMapping
public ResponseEntity<?> addCustomer(@RequestBody Customer newCustomer,
                                     UriComponentsBuilder uri) {
    if (newCustomer.getId() != 0
        || newCustomer.getName() == null
        || newCustomer.getEmail() == null) { // Reject - we'll assign the
customer id
        return ResponseEntity.badRequest().build();
    }
    newCustomer=repo.save(newCustomer);
    URI location=ServletUriComponentsBuilder.fromCurrentRequest()
        .path("/{id}").buildAndExpand(newCustomer.getId()).toUri();
    ResponseEntity<?> response=ResponseEntity.created(location).build();
    return response;
}
```

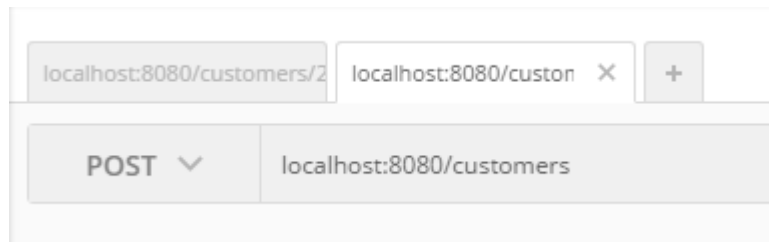
__2. Organize imports.

__3. Save the file.

- __4. Switch to **Gradle Tasks** pane.
- __5. Expand **spring-boot-rest > build**.
- __6. Right click **build** and click **Run Gradle tasks**.
- __7. Ensure there are no errors in **Console** pane.
- __8. Run **App.java**

Wait until the app is started.

- __9. In the **Postman** tool, open a new tab by clicking on the '+' icon in the tab headers.
- __10. Click the 'method' drop-down box and select **POST**.
- __11. Enter **localhost:8080/customers** as the request URL.

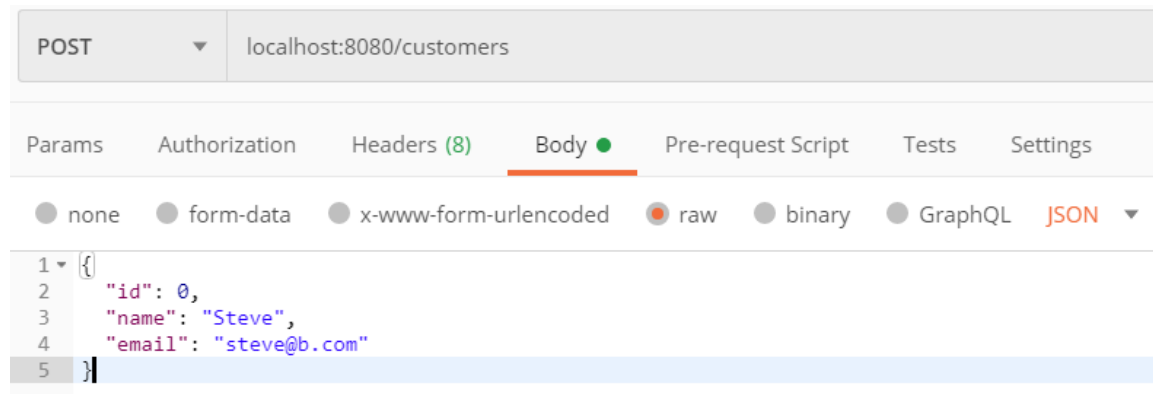


- __12. Select the **Body** tab and then click the radio button for **raw**.
- __13. Expand the drop-down box and select **JSON**
- __14. Enter the following new customer object in the body text area (hint: you might find it convenient to copy and edit an object from the response to the 'GET' request that we issued earlier - it's in the other tab in **Postman**):

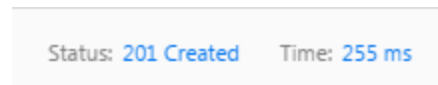
```
{
  "id": 0,
  "name": "Steve",
  "email": "steve@b.com"
}
```

Note: Make sure you set the 'id' value to 0.

__15. When it looks like below, click **Send**.



The response should show a '201 Created' status. Look at the bottom section.



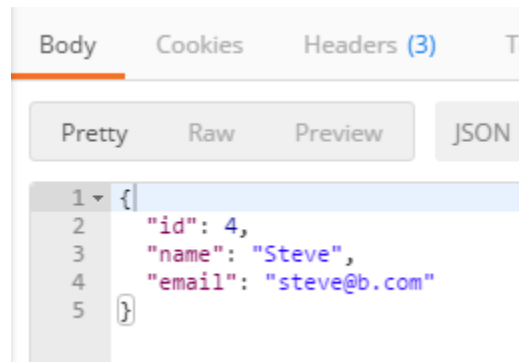
__16. Click on the **Headers** tab in the response area [bottom section]. You should see the 'Location' header:

Body	Cookies	Headers (3)	Test Results	Status: 201 Created	Time: 567ms	Size:
		KEY	VALUE			
		Location ⓘ	http://localhost:8080/customers/4			
		Content-Length ⓘ	0			
		Date ⓘ	Mon, 06 Apr 2020 20:16:47 GMT			

The location header gives us the URL for the newly created customer object. Notice that the repository assigned the next customer id for us.

__17. Copy the contents of the 'Location' header into the URL area and issue a GET request.

You should see the new Customer object similar to the following:



__18. Leave **Postman** open, but stop 'App.java'

Part 8 - Add a PUT Method Handler

The PUT method models an update to a particular customer. For instance to change the customer's email for customer id 4, we would PUT an updated object to '/customers/4'.

__1. Add the following method to the 'CustomerAPI' class:

```
@PutMapping("/{customerId}")
public ResponseEntity<?> putCustomer(@RequestBody Customer newCustomer,
    @PathVariable("customerId") long customerId) {
    if (newCustomer.getId() != customerId
        || newCustomer.getName() == null
        || newCustomer.getEmail() == null) {
        return ResponseEntity.badRequest().build();
    }
    newCustomer=repo.save(newCustomer);
    return ResponseEntity.ok().build();
}
```

__2. Organize the imports by pressing **Ctrl-Shift-O**.

__3. Save the file.

Notice that we're verifying the customer id in the supplied object matches the id that's in the path variable. If there's a difference, we return a 'badRequest' response.

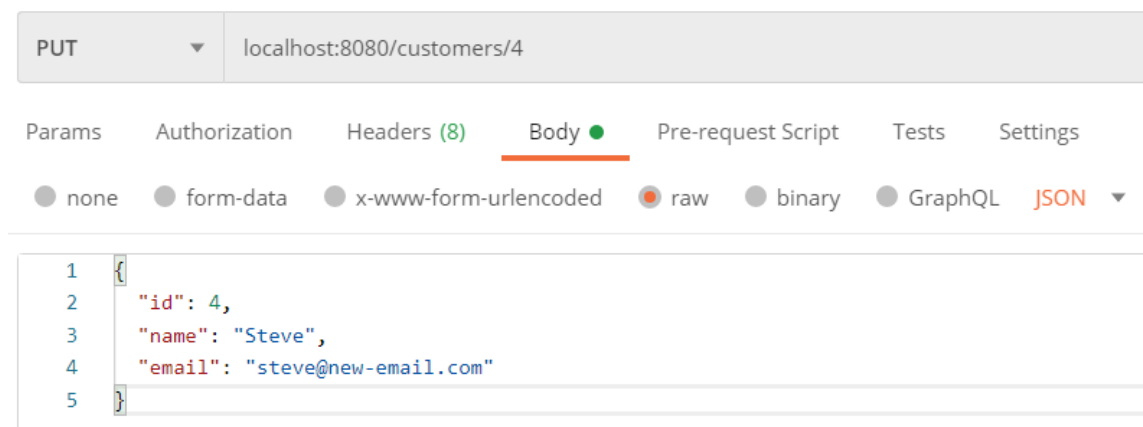
- __4. Switch to **Gradle Tasks** pane.
- __5. Expand **spring-boot-rest > build**.
- __6. Right click **build** and click **Run Gradle tasks**.
- __7. Ensure there are no errors in **Console** pane.
- __8. Run **App.java**

Wait until the app is started.

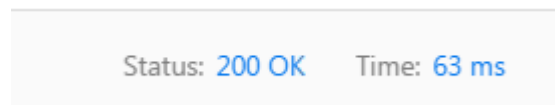
- __9. In the **Postman** tool, open a new tab by clicking on the '+' icon in the tab headers.
- __10. Click the 'method' drop-down box and select **PUT**.
- __11. Enter **localhost:8080/customers/4** as the request URL.
- __12. Select the '**Body**' tab and then click the radio button for **raw**.
- __13. Expand the drop-down box and select **JSON**.
- __14. Enter the following new customer object in the body text area:

```
{
  "id": 4,
  "name": "Steve",
  "email": "steve@new-email.com"
}
```

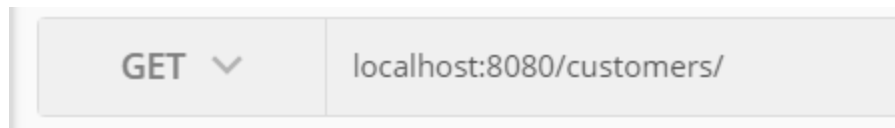
- __15. When looks like below, click **Send**.



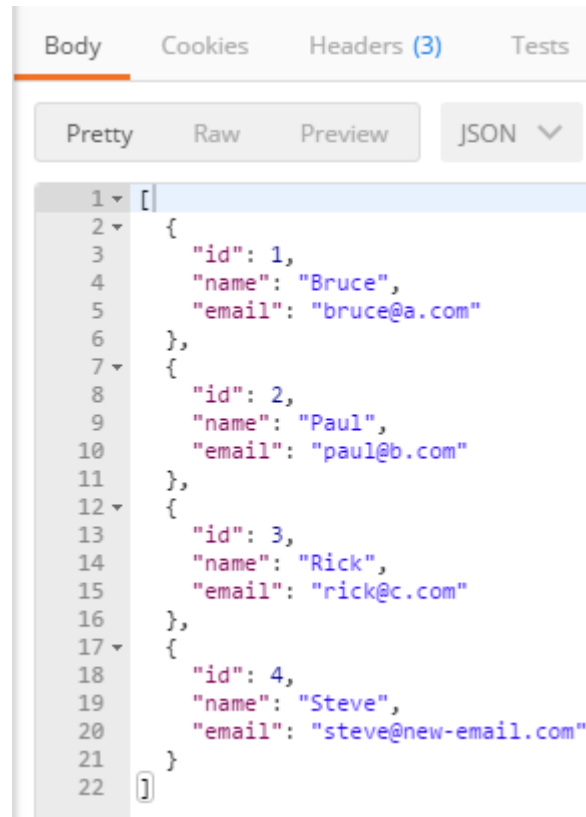
You should see a '200 OK' response.



__16. In a different tab in **Postman**, do a **GET** request to **/customers**

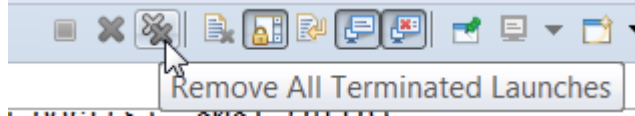


Notice that the record for customer id 4 has been updated with the new email address we supplied.



__17. Close Postman.

- __18. Back in eclipse, close all open files.
- __19. Stop 'App.java' by clicking on the red stop button.
- __20. Click **Remove All Terminated Launches**.



Part 9 - Review

In this lab, we used Spring MVC to implement a CRUD service that stores Customer objects. Thanks to Spring's automatic mapping to JSON, we had to do very little coding to realize the API.

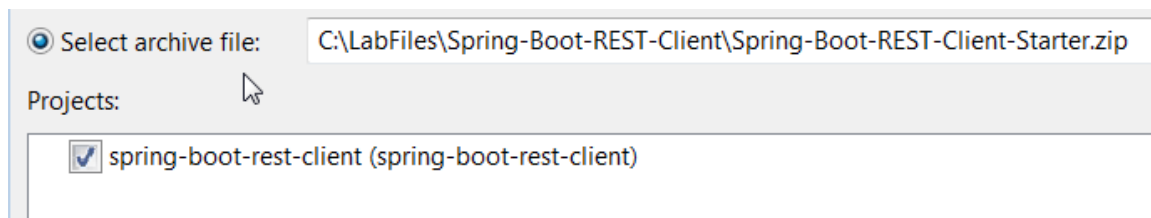
Lab 7 - Create a RESTful Client with Spring Boot

In this lab you will use Spring's RestTemplate class to make a call to a RESTful API to retrieve a list of Customers.

Note: This lab requires that the 'spring-boot-rest' application is running. If necessary, load the project from the solutions and run 'App.java'.

Part 1 - Import the Starter Project

- ___ 1. From Eclipse's main menu, click **File** → **Import** and then select **General** → **Existing Projects into Workspace**, and then click **Next**.
- ___ 2. Click the radio button for **Select archive file**:
- ___ 3. Click the **Browse** button and navigate to select **C:\LabFiles\Spring-Boot-REST-Client\Spring-Boot-REST-Client-Starter.zip** and then click **Open**.
- ___ 4. On the **Import** dialog, leave the defaults as-is and click **Finish**.



Part 2 - Examine the Starter Project

The starter project implements a web page to display a list of Customer objects. There is a CustomerDAO interface provided, but no implementation of that interface.

There are a few items already setup in the starter project that you should take note of for your own projects

- Also in 'src/main/resources', there is a file called 'application.yml'. This file contains the following content:

```
---
server:
  port: 8081
```

- The line disables sets up the embedded web server to run on port 8081. We need to configure this for our test, because the API that we'll be hitting is already running on port 8080.
- The starter project contains a domain class that we will use as a starting point for our repository.
- The starter project also contains a rudimentary user interface for our demonstration program.

__ 1. Right click **spring-boot-rest-client** and select **Maven → Update Project**.

__ 2. Make sure **spring-boot-rest-client** is selected and click OK.

Part 3 - Complete the CustomerDAO

The repository class 'CustomerDAO' implements the CustomersDAO interface, which is called by the user interface's controller object. In the starter project, the implementation is incomplete; it just returns an empty ArrayList of Customers. We'll change that to get the list of customers from a RESTful API.

__ 1. In the **Project Explorer**, locate the **APIClientCustomersDAO** class in the **spring-boot-rest-client/src/main/java/com.webage.dao** package.

__ 2. Double-click on the file to open it.

__ 3. Add the following line inside the class to provide the connection URL as an instance variable:

```
String customersAPIbase="http://localhost:8080/customers";
```

__ 4. Look for the 'Insert code here...' comment inside the 'getAllCustomers' method and edit the method to look like this:

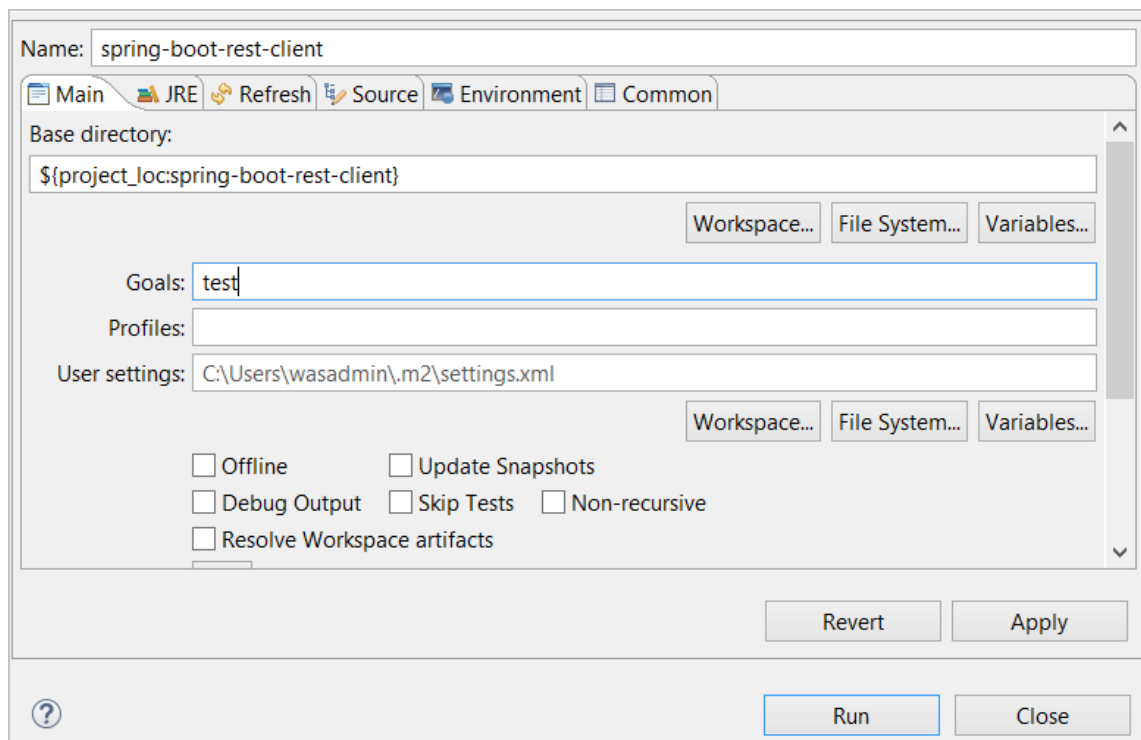
```
@Override
public Collection<Customer> getAllCustomers() {
    // Construct a GET request to the CustomersAPI base url
    // Insert code here..
    RestTemplate template=new RestTemplate();
    Customer[] customers=template.getForObject(customersAPIbase,
Customer[].class);
    return Arrays.asList(customers);
}
```

The method creates an instance of RestTemplate, and then calls the 'getForObject(...)' method to retrieve and convert a JSON response from the server.

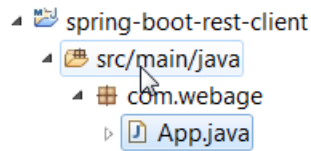
- __ 5. Organize imports by pressing **Ctrl-Shift-O**. Select **java.util.Arrays**.
- __ 6. Save the file.
- __ 7. Right click **spring-boot-rest-client** and select **Run as** → **Maven install**.

Wait until you see BUILD SUCCESS in the Console.

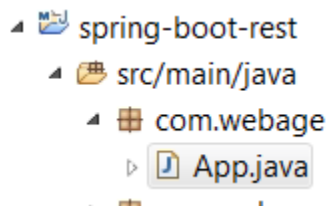
- __ 8. Right click **spring-boot-rest-client** and select **Run as** → **3 Maven Build**.
- __ 9. Enter **test** as **Goals** and click **Run**.



__10. Run the **App.java** as **Java Application** to startup the server on the **spring-boot-rest-client** project.



__11. Run the **App.java** as **Java Application** to startup the server on the **spring-boot-rest** project. [Project from previous Lab].



__12. Open a Web Browser and enter:

http://localhost:8081/browseCustomers

You should see a response similar to:

Browse Purchases

[Add Purchase](#)

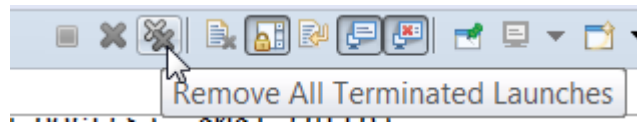
Customer Id	Customer Name	Email
1	Bruce	bruce@a.com
2	Paul	paul@b.com
3	Rick	rick@c.com

[Back to Main Menu](#)

__13. Close the browser.

__14. Stop 'App.java' by clicking on the red stop button.

__15. Click **Remove All Terminated Launches**.



__16. Repeat the previous 2 steps as many times as needed until your console is clean.

__17. Close all open files.

Part 4 - Review

In this lab, we used Spring's RestTemplate object to quickly implement a client to a RESTful service.

Lab 8 - Enable Basic Security

In this lab we will enable basic security on a Spring Boot web application.

Part 1 - Import the Starter Project

__1. Extract the contents of **C:\LabFiles\Gradle\Spring-Boot-Basic-Security-Starter.zip** to **C:\Workspace** directory.

__2. Verify the following folder was created:

C:\Workspace\spring-boot-basic-security

__3. From Eclipse's main menu, select **File** → **Import** and select **Gradle** → **Existing Gradle Project**, and then click **Next**.

__4. Click **Browse** button.

__5. Select **C:\Workspace\spring-boot-basic-security** and click **Select Folder**.

Import Gradle Project

Specify the root directory of the Gradle project to import.

Project root directory **C:\Workspace\spring-boot-basic-security**

__6. Click **Finish**.

Part 2 - Enable Security

There actually isn't much work required to enable security. Spring Boot looks in the classpath for various modules when it starts up, and enables them if they are present. So all we need to do to enable basic security is to put Spring Security on the classpath. The Spring Boot project provides a starter package that does exactly that.

__1. In the **Project Explorer**, expand **spring-boot-basic-security**.

__2. Locate '**build.gradle**' and double-click on it to open the file.

___3. Add the following dependency element just before the closing dependencies block {:

```
compile group: 'org.springframework.boot', name: 'spring-boot-starter-security', version: '2.1.8.RELEASE'
```

___4. Save and close the file.

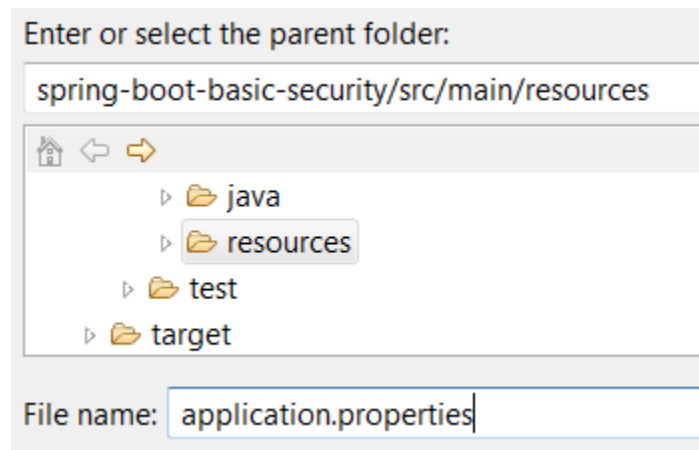
Part 3 - Configure the User Password

___1. In the **Project Explorer**, locate the node for **src/main/resources**.

___2. Right-click on **src/main/resources** and select **New** → **Other**.

___3. Select **General** → **File** and click **Next**.

___4. Enter **application.properties** as the filename and click **Finish**.



___5. In the new file, enter the following lines:

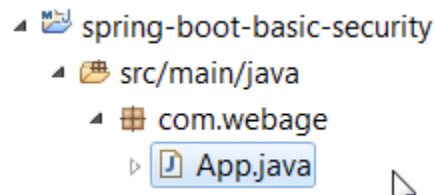
```
spring.security.user.name=user  
spring.security.user.password=Pa$$w0rd
```

___6. Save and close the file.

Part 4 - Build and Test

Using the same technique as in previous labs, you will run Gradle Refresh and then run App.java as a Java Application.

- ___ 1. Right click **spring-boot-basic-security** and select **Gradle** → **Refresh Gradle Project**.
- ___ 2. Switch to **Gradle Tasks** pane.
- ___ 3. Expand **spring-boot-basic-security** > **build**.
- ___ 4. Right click **build** and click **Run Gradle tasks**.
- ___ 5. Ensure there are no errors in **Console** pane.
- ___ 6. Right-click **App.java** under **src/main/java** → **com.webage** and click **Run As** → **Java Application**.



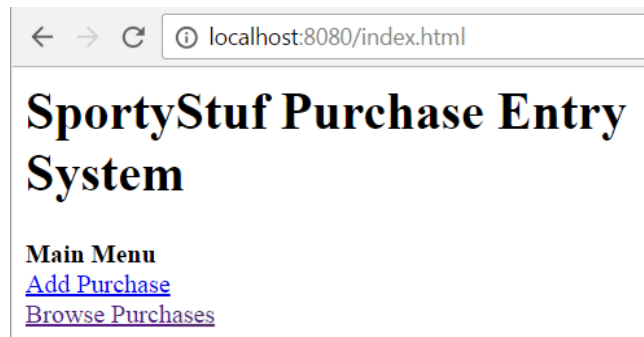
- ___ 7. Open a browser and navigate to:

http://localhost:8080

- ___ 8. You will be prompted for a user name and password. Enter **user** as the username and **Pa\$\$w0rd** as the password and then click **Sign in**.

A screenshot of a web application's sign-in page. The page has a light gray background. At the top, the text 'Please sign in' is displayed in a large, bold, dark blue font. Below this text are two input fields. The first field contains the text 'user'. The second field contains a series of dots, indicating a password. Below the input fields is a large blue button with the text 'Sign in' in white.

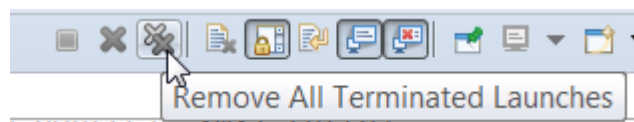
__9. Once you've entered your password, you will have full access to the application.



__10. Close the browser.

__11. Stop 'App.java' by clicking on the red stop button.

__12. Click **Remove All Terminated Launches**.



__13. Close all open files.

__14. Close Eclipse.

Part 5 - Review

This was a very short lab demonstrating how simple it is to add basic security to an application that is built using Spring Boot.

Lab 9 - 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:

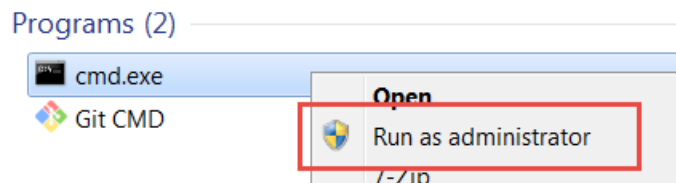
1. Verify Jenkins is running
2. Configure tools in Jenkins

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.



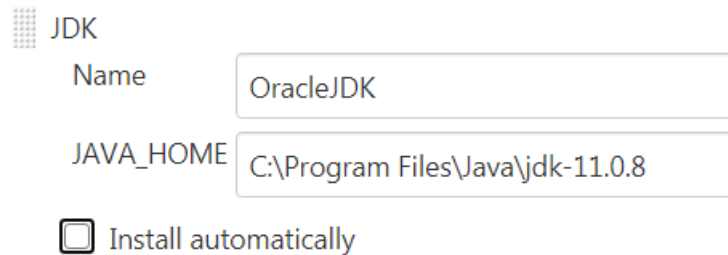
Don't worry about any warning.

- __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 than above, contact your instructor or search for the right path and use it as JAVA_HOME.

- __15. Verify your settings look as below:



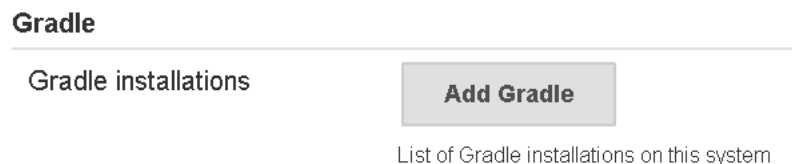
JDK

Name

JAVA_HOME

☐ 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 10 - 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.

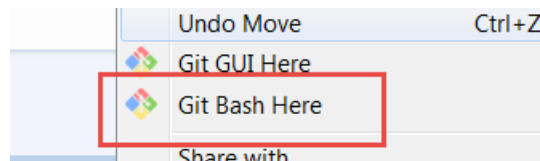
Part 1 - Create a Git Repository

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`

You will see:

```
$ 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
```

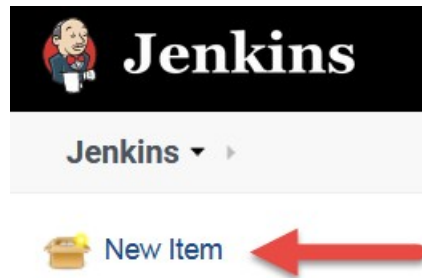
__7. Enter the following to clone the current Git repository into a new remote repository.

```
git clone --bare . /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.

The image is a screenshot of the Jenkins 'Enter an item name' form. It has a light gray background. At the top, the text 'Enter an item name' is displayed. Below this is a text input field containing the text 'SimpleGreeting'. Underneath the input field, the text '» Required field' is shown. Below the input field is a section titled 'Freestyle project' with a small icon of a box and a globe. To the right of the icon, the text reads: '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.'

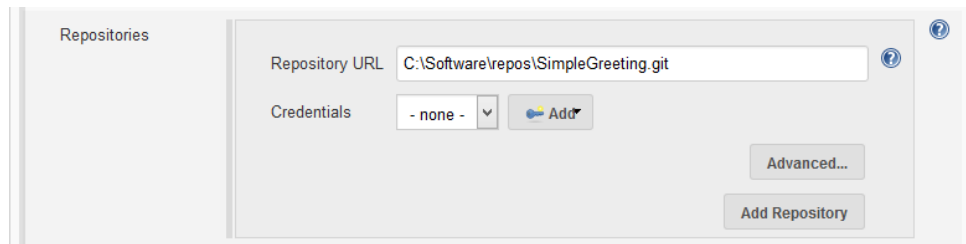
__4. Click **OK**, to add a new job.

After the job is created, you will be on the job configuration page.

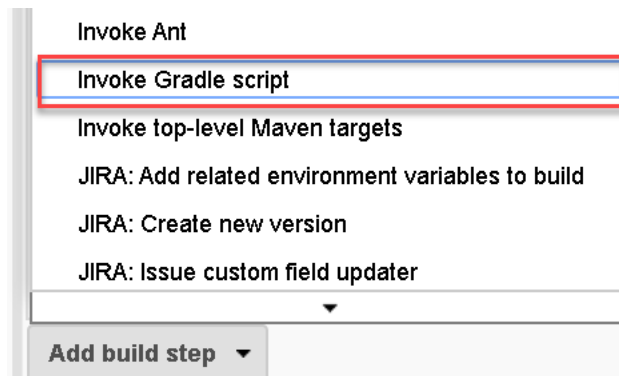
__5. Scroll down to the **Source Code Management** section and then select **Git**.

The image is a screenshot of the Jenkins 'Source Code Management' section. It has a light gray background. The title 'Source Code Management' is at the top. Below the title are two radio button options: 'None' and 'Git'. The 'Git' option is selected, indicated by a blue dot in the center of the radio button.

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



__7. In **Build Steps** section, click **Add build step** and select **Invoke Gradle script**

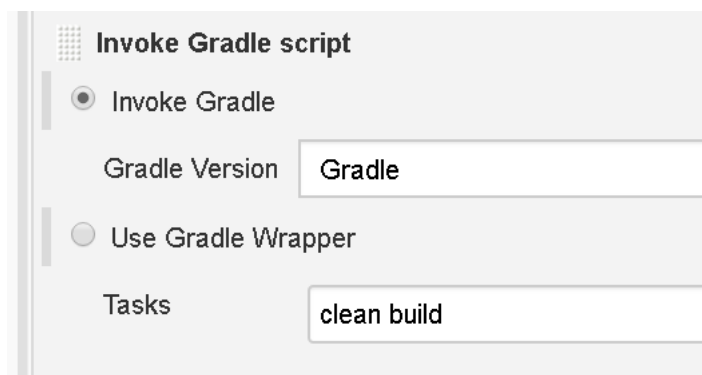


__8. Ensure **Invoke Gradle** radio button is selected.

__9. In **Gradle Version**, from the drop-down select **Gradle**

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

__11. The **Invoke Gradle script** configuration should look like this:



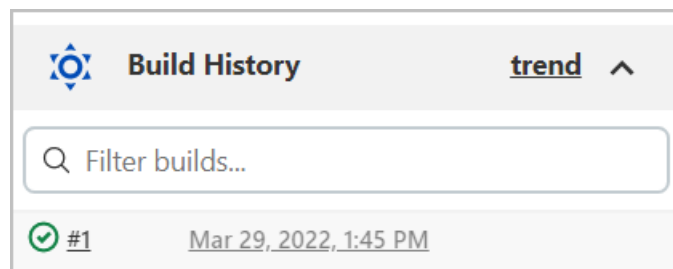
__12. Click **Save**.

You will see the Job screen.

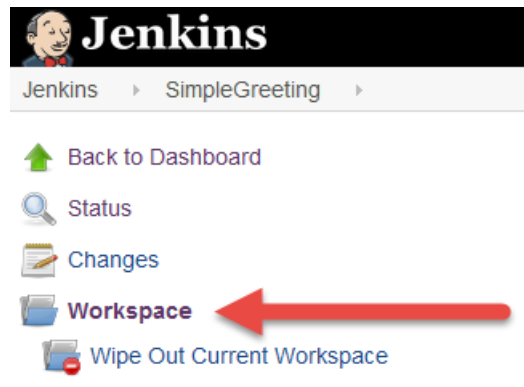
Project SimpleGreeting

__13. From the left menu, click **Build Now**.

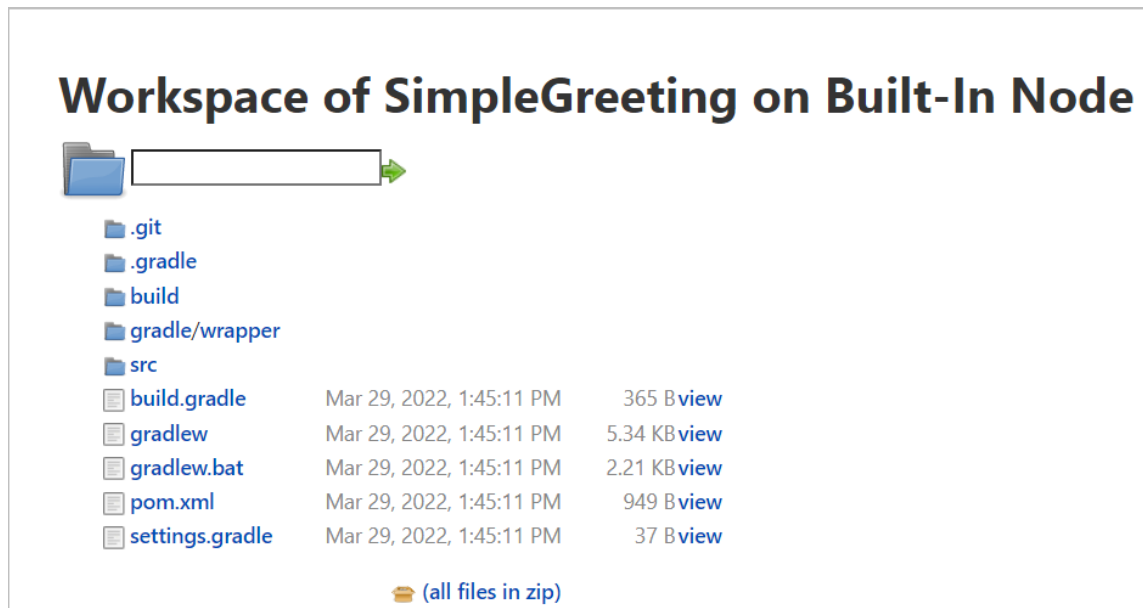
You should see the build in progress in the **Build History** area.



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

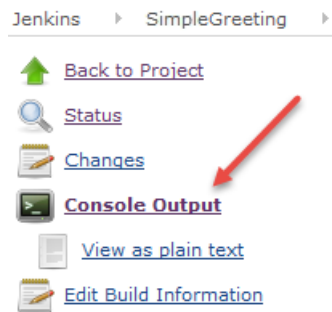


You will see that the directory is populated with the source code for our project.



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

__16. Click the **Console Output** from the left menu.



__17. 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.

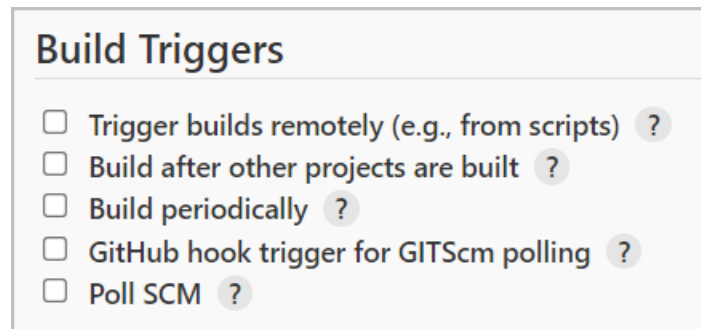
Part 3 - Enable Polling on the Repository

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.

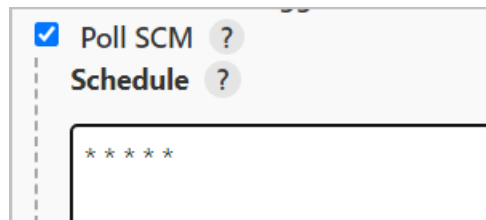
__2. Click the **Configure** link.

__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`

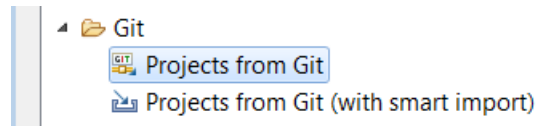
Note: By default, Gradle stores test results in the folder mentioned above. In this case, the actual file name is TEST-com.simple.TestGreeting.xml but you can use * wildcard to specify the file name.

__ 8. Click **Save**.

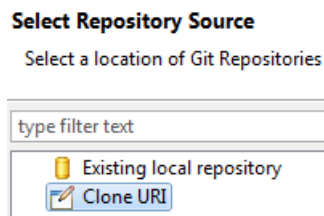
Part 4 - Import the Project into Eclipse

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
- __ 2. Select C:\Workspace as Workspace and click **Launch**.
- __ 3. From the main menu, select **File** → **Import...**
- __ 4. Select **Git** → **Projects from Git**.

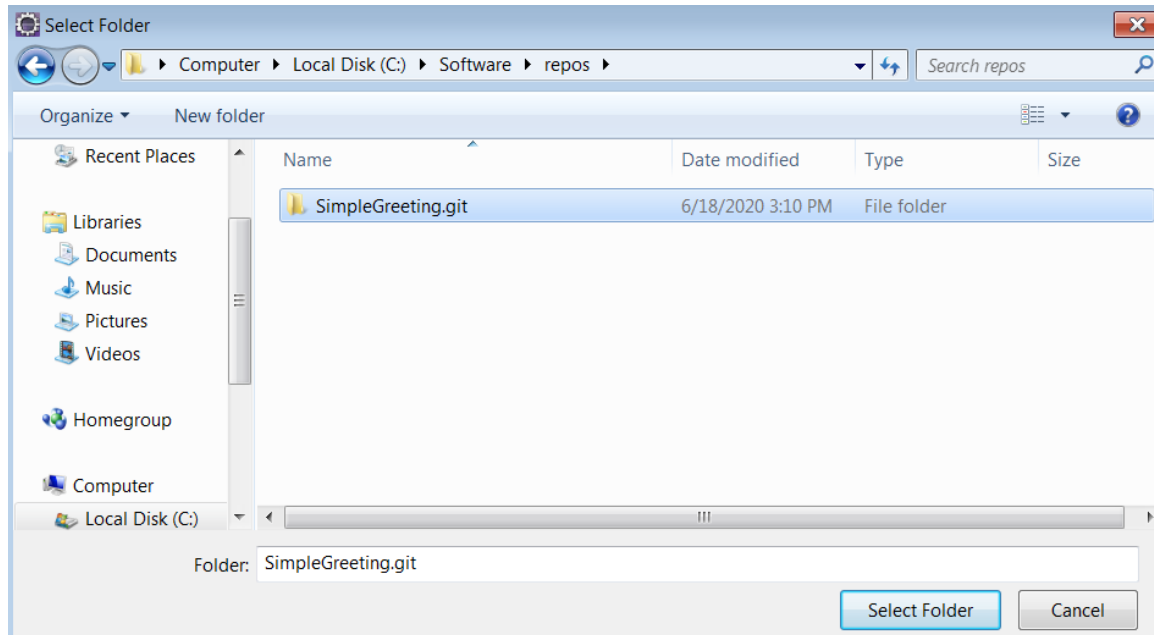


- __ 5. Click **Next**.
- __ 6. Select **Clone URI** and then click **Next**.



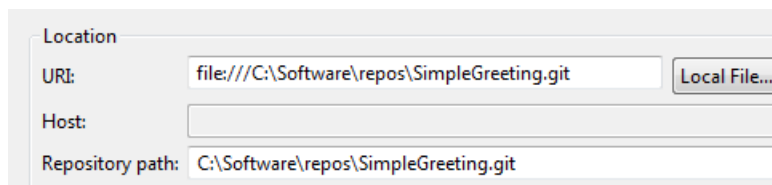
You might think that 'Existing local repository' would be the right choice, since we're cloning from a folder on the same machine. Eclipse, however, expects a "local repository" to be a working directory, not a bare repository. On the other hand, Jenkins will complain if we try to get source code from 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.

__7. Click on **Local File...** and then navigate to **C:\Software\repos\SimpleGreeting.git**

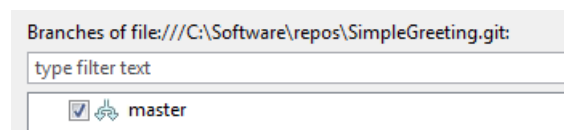


__8. Click **Select Folder**.

__9. Back in the **Import Projects** dialog, click **Next**.

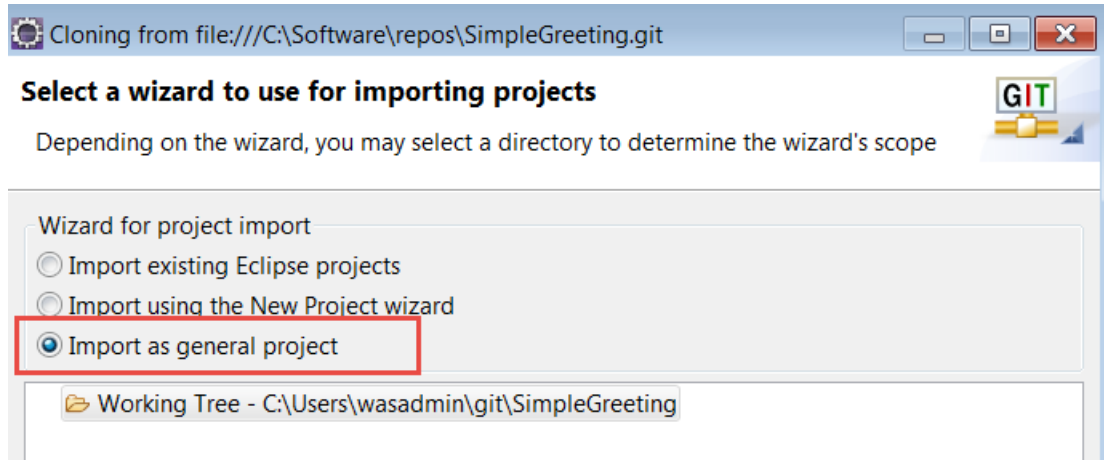


__10. Click **Next** to accept the default 'master' branch.

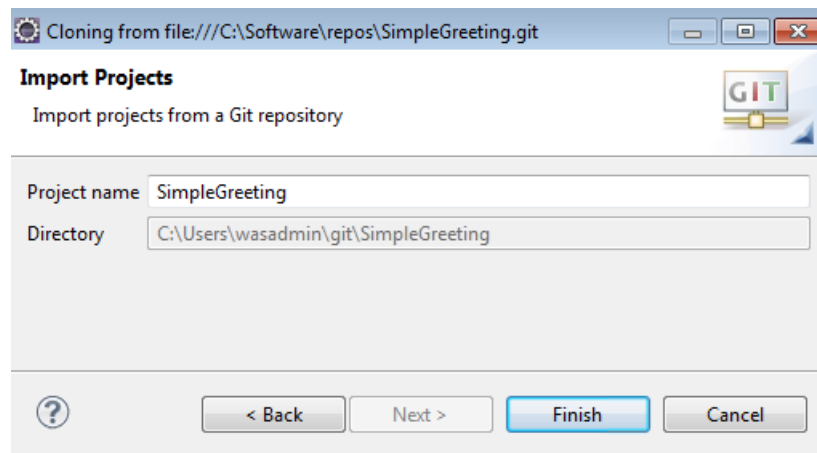


__11. In the **Local Destination** pane, leave the defaults and click **Next**.

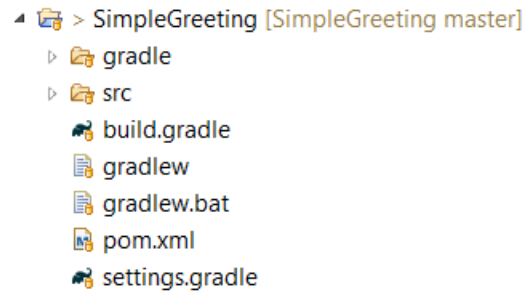
__12. Select **Import as a general project** and click **Next**.



__13. Click **Finish**.



__14. 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";'

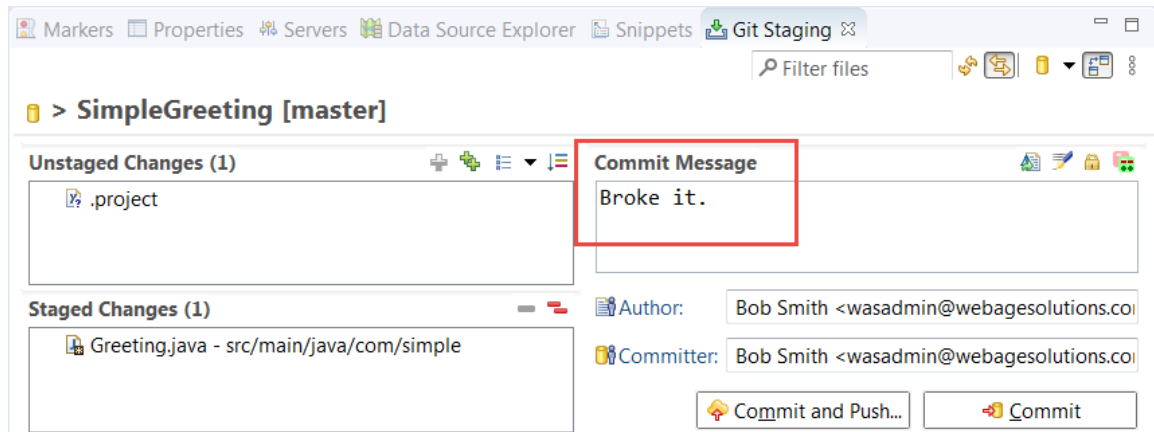
```
public String getStatus(){  
    return "BAD";  
}
```

__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**.

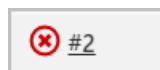


__7. Click **Close** in the status dialog that pops up.

__8. Now, flip back to the web browser window that we had Jenkins running in.

__9. 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. You can launch a new build if it's taking too long by clicking **Build Now**. You may need to refresh the page.

__10. 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'.

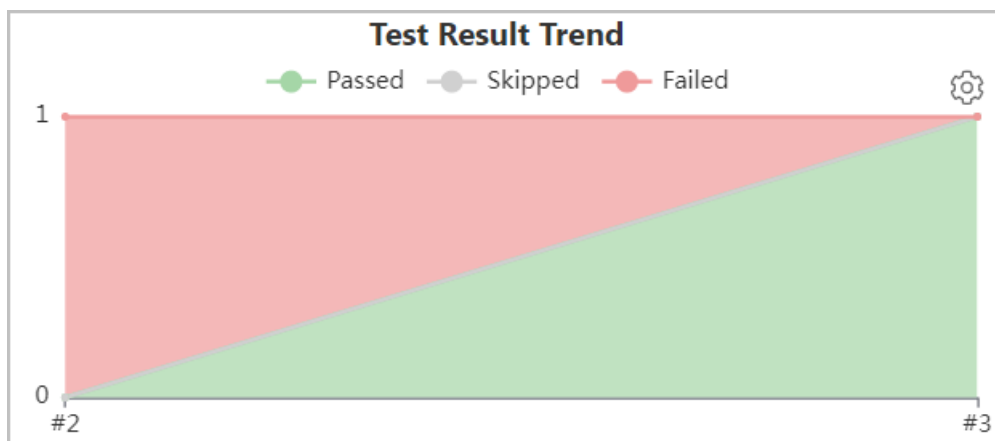
```
public String getStatus(){  
  
    return "GOOD";  
  
}
```

__2. As above, "Save", 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**.

__4. This time the build will pass, refresh the page when the build is done.

A graph will be shown.



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 11 - 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

SimpleGreetingPipeline

» 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).

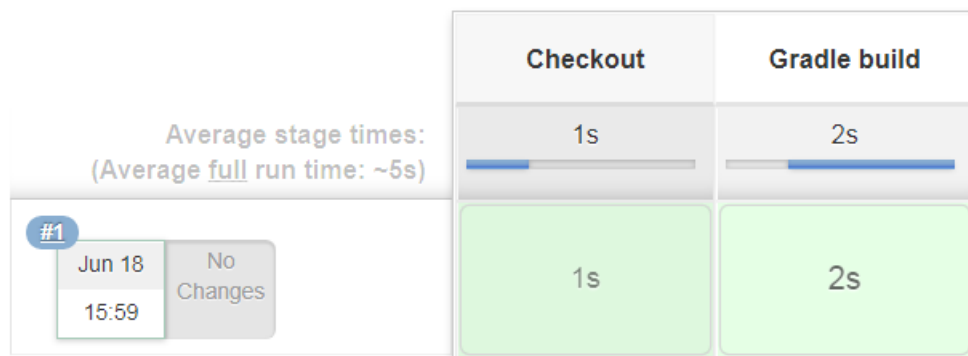
__5. Click on **Save** to save the changes and return to the project page.

__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.

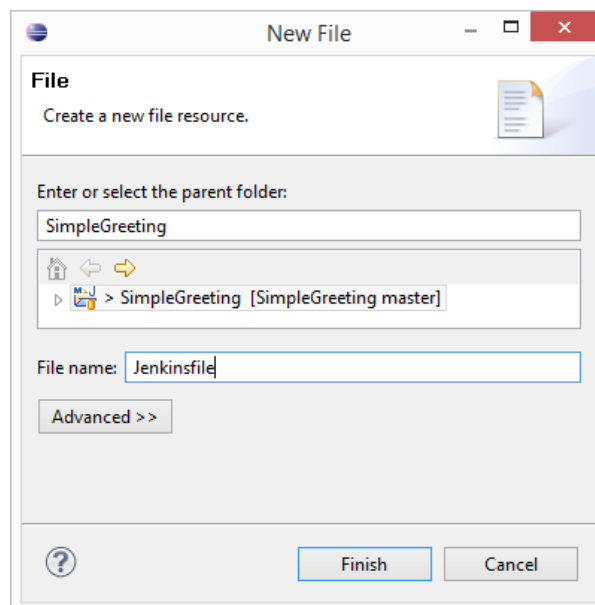
In the following steps, we'll create a Jenkinsfile and create a pipeline job that uses it.

__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.

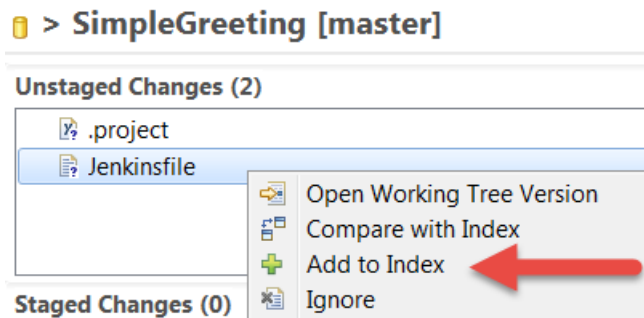


__4. Click **Finish** to create the new file.

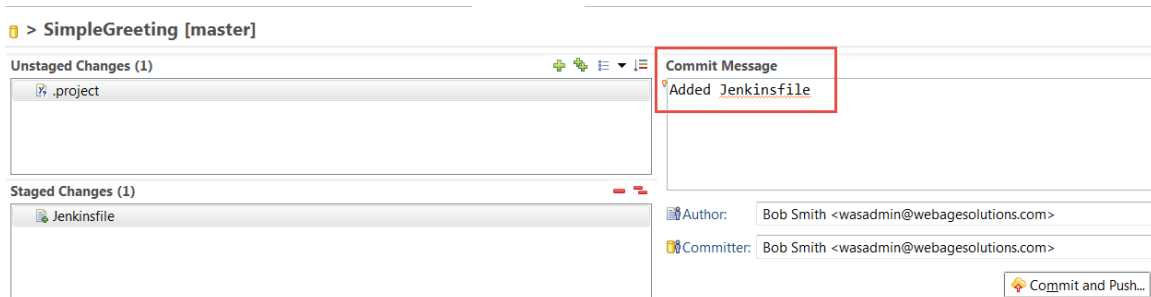
- __5. You may see the 'Editors available on the Marketplace' window, just click **Cancel**.
- __6. You may see a compatibility error, just click OK.
- __7. Enter the following text into the new file (Note: this is the same script that we used above, so you could copy/paste it from the Jenkins Web UI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

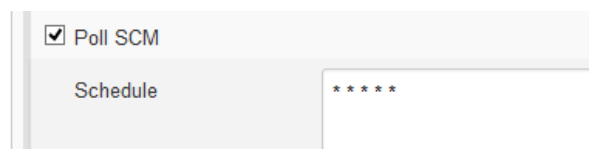
__14. Select **Pipeline** as the item type.

__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)

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

The screenshot shows a web-based configuration interface for a pipeline. It has a light gray background with white panels for each section. The 'Pipeline' section is at the top, followed by 'Definition' which contains a text box with 'Pipeline script from SCM'. Below that is the 'SCM' section with a dropdown menu showing 'Git'. The 'Repositories' section follows, containing a 'Repository URL' text box with the value 'C:\Software\repos\SimpleGreeting.git'. At the bottom is the 'Credentials' section, which has a dropdown menu showing '- none -' and an 'Add' button with a key icon.

__21. Click **Save** to save the new configuration.

__22. Click **Build Now** to launch the pipeline.

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";  
  
}
```

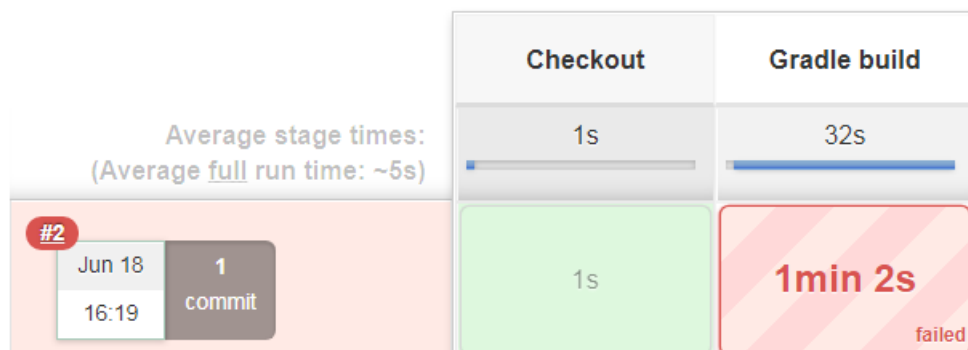
__4. Save the file.

__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.

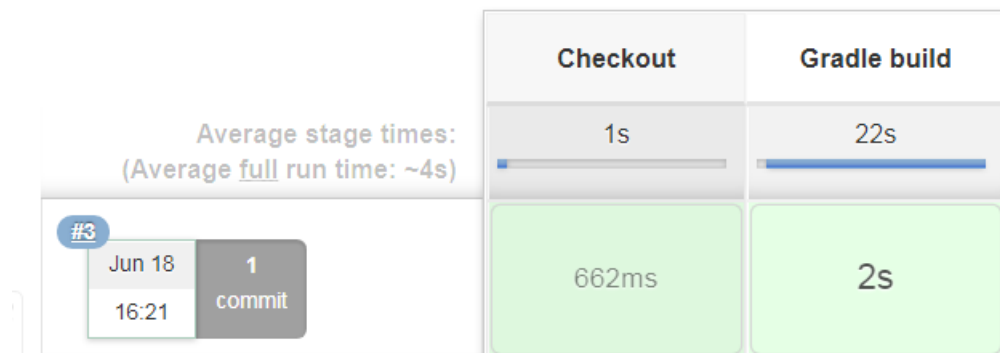
___ 10. Alter the 'bat "gradle"' line to read as follows:

```
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**.
- __5. 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".

The screenshot displays the Jenkins Pipeline console interface. At the top, a table shows stage performance metrics:

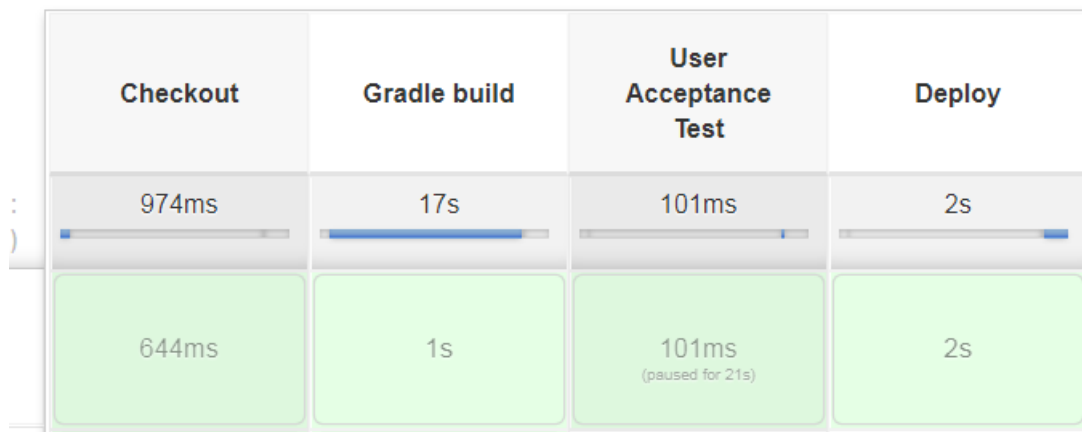
Stage	Checkout	Gradle build	User Acceptance Test
Average stage times:	990ms	17s	373ms
(Average full run time: ~4s)			

Below the table, the pipeline history shows two builds. Build #4 is highlighted, showing a commit at 16:23 on Jun 18. The main console view shows the 'User Acceptance Test' stage is currently paused. A yellow dialog box is overlaid on the console, asking 'Is this build good to go?' with a 'Pass' status. The dialog includes a dropdown menu set to 'Yes' and two buttons: 'Proceed' and 'Abort'. A blue dashed box on the right indicates the stage is '(paused for 10s)'.

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 delete the SimpleGreetings project if this happen.

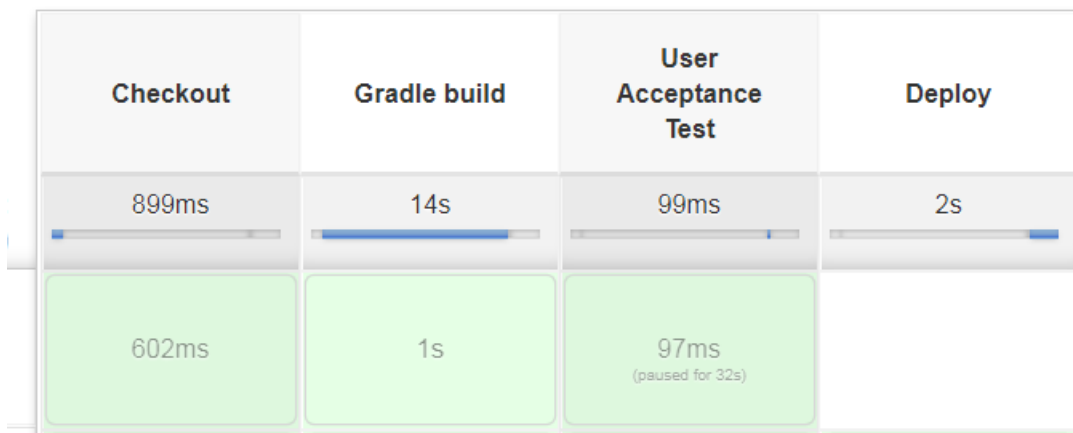
__6. Select **Yes** and click **Proceed**.

You should see the job run to completion.



__7. Click **Build Now** to run the pipeline again but this time select **No** on the **User Acceptance Test** and then 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.

__ 8. 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 12 - OPTIONAL: 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.



__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.



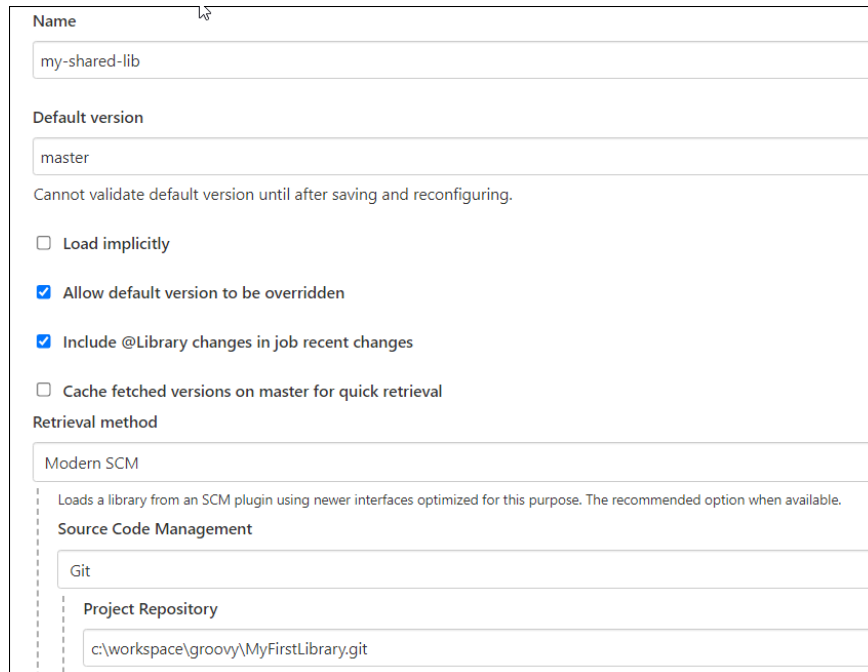
__ 10. In **Name** text box, enter "**my-shared-lib**" (without quotes).

__ 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.

A screenshot of the Jenkins Global Pipeline Libraries configuration form. The form is divided into several sections. The 'Name' section has a text box containing 'my-shared-lib'. The 'Default version' section has a text box containing 'master' and a message 'Cannot validate default version until after saving and reconfiguring.' Below this are three checkboxes: 'Load implicitly' (unchecked), 'Allow default version to be overridden' (checked), and 'Include @Library changes in job recent changes' (checked). The 'Retrieval method' section has a dropdown menu set to 'Modern SCM' with a description 'Loads a library from an SCM plugin using newer interfaces optimized for this purpose. The recommended option when available.' The 'Source Code Management' section has a dropdown menu set to 'Git'. The 'Project Repository' section has a text box containing '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	<input type="text" value="my-shared-lib"/>
Default version	<input type="text" value="master"/> <div>Currently maps to revision: 2ea0ae0b71303c9300a92c795297473bf2113e49</div>
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**
- __ 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**.

__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:

```
[Pipeline] Start of Pipeline
[Pipeline] echo
Hi, Bob
[Pipeline] echo
Square is: 25
[Pipeline] echo
Date is: 09/25/2021 10:30:36
[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**
- __ 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.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.

- __ 7. Click **Apply** button.

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

- __ 8. Click **Save** button.

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

- __ 9. Open eclipse in the same workspace.
- __ 10. Go to the **Project Explorer** and locate the file 'Greeting.java'. It will be under **src/main/java** in the package 'com.simple'.
- __ 11. Open 'Greeting.java'.
- __ 12. Locate the line that reads 'return "BAD";'. Change it to read 'return "**GOOD**";'
- __ 13. Save the file.
- __ 14. In the **Project Explorer**, right-click on **Greeting.java** and then select **Team** → **Commit...** (This is a shortcut for committing a single file).
- __ 15. Enter an appropriate commit message and then click **Commit and Push**.

- __16. Click Close.
- __17. Go back to Jenkins.
- __18. Click **Build Now**.
- __19. 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

- __20. After all stages are completed, notice a new build shows up under **Build History**.
- __21. Click the latest build.
- __22. Click **Console Output**.
- __23. 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

```

- __24. In **File Explorer** go to C:\Workspace\dev and notice SimpleGreeting-*.jar is copied here by the shared library script upon user acceptance.
- __25. Close all.

Part 8 - Review

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

