Discover Financial Services

Tutorial to Creating a Spring MVC Project with SpringBoot, Gradle, Nexus, GitHub, SonarQube, JUnit, Jenkins, and PCF

INITIAL VERSION 6/30/2016

LAST UPDATED 8/1/2016

CONTRIBUTORS

VICTORIA NISHIMOTO – CMA COLLECTIONS

Summary of revisions:

Version	Edited by	Date	Description of changes
1.0	Victoria Nishimoto	6/30/2016	Initial Version
1.1	Victoria Nishimoto	7/14/2016	Changed templating language from Thymeleaf to Freemarker
1.2	Victoria Nishimoto	7/19/2016	Fixed Freemarker error. Added SonarQube and JUnit section
1.3	Victoria Nishimoto	7/27/2016	Eliminated unnecessary steps
1.4	Victoria Nishimoto	7/29/2016	Improved JUnit Testing section
1.5	Victoria Nishimoto	8/1/2016	Replaced deprecated spring annotation in JUnit Testing section

Contents

Introduction

Before you begin: This guide will walk through the full lifecycle of a Spring MVC application. In this example, we will be creating an application that will list IMS Data Entities. It will also have create, update, and delete capabilities. It will be connected to DB2 using JDBC. It can easily be changed for different datasources, though. It will be committed to GitHub, built with Jenkins using Gradle, uploaded to Nexus, and deployed to the cloud.

Sections:

Section 1: Getting Access

1. In this section, you will find a list of permissions and installs you will need to complete this tutorial
2. Go to https://niqidm.discoverfinancial.com/landing
3. Log in
4. To check your access, click Request Access and then View User Access. Search User using your last name and then click “Show User Access”. Wait for your user access to load. This may take a few minutes
5. You will need the following Application Roles:
    - *PCF (Pivotal Cloud Foundry user) PaaS ~ One Access Level
    - Continuous Integration Users ~ Users and Developers of Continuous Integration
    - GIT ~ GIT Applications
    - GIT ~ GIT Preview 2 6 1 EN
    - Gradleware ~ Gradleware 2 6 EN
    - Internet Access ~ Internet Access
    - Jenkins ~ Developer (writer) of [insert the name of your team]
        i. ex. Jenkins ~ Developer (writer) of CMA Collections Repos
    - Oracle ~ Oracle JDK 1 8 0 20 x64 EN
6. If you don’t have the above accesses, you will need to request them.
7. To request access, click Request Access and then Request User Access. Search User using your last name. You can then search for the above accesses
8. Once your access is complete, you should be able to download the following from your DFS Application Catalog that you can access at http://rwkscm03.rw.discoverfinancial.com/CMApplicationCatalog/#/SoftwareLibrary/AppListPageView.xaml (Copy and paste into Internet Explorer)
    - GIT Extensions 2.47.3

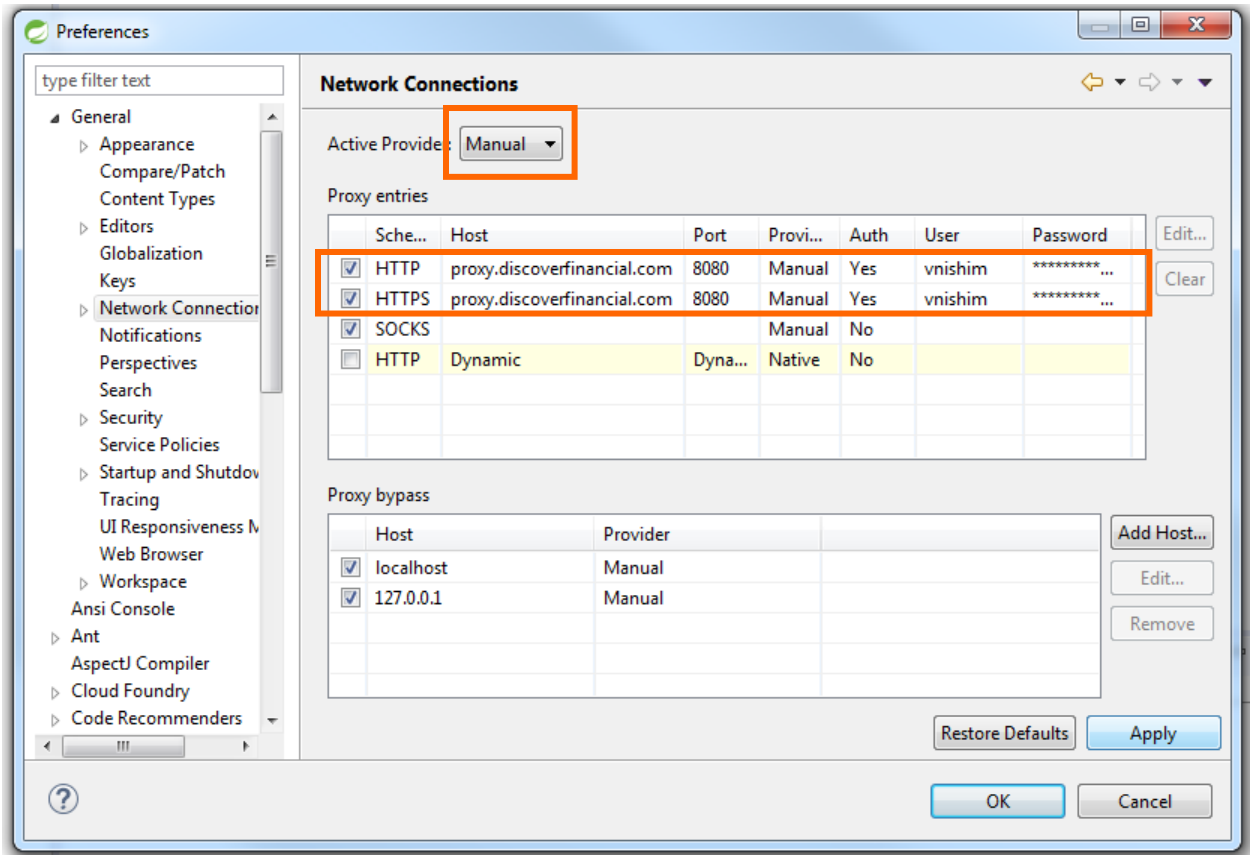
- GIT GUI 1.8.4-msysgit
- GIT Preview 2.6.1 EN
- Gradleware 2.6
- Oracle JDK 1.8.0_20_x64

9. For the PCF request, once submitted PaaS Operations will get in touch with you and ask for some information before you can receive access
    - Foundation: DEV
    - Org Name: your group name, if not sure, email DC-PAASADMIN (ex. cma-collections-org)
    - Space: development
    - Role: Space Developer

Section 2: Setting up STS

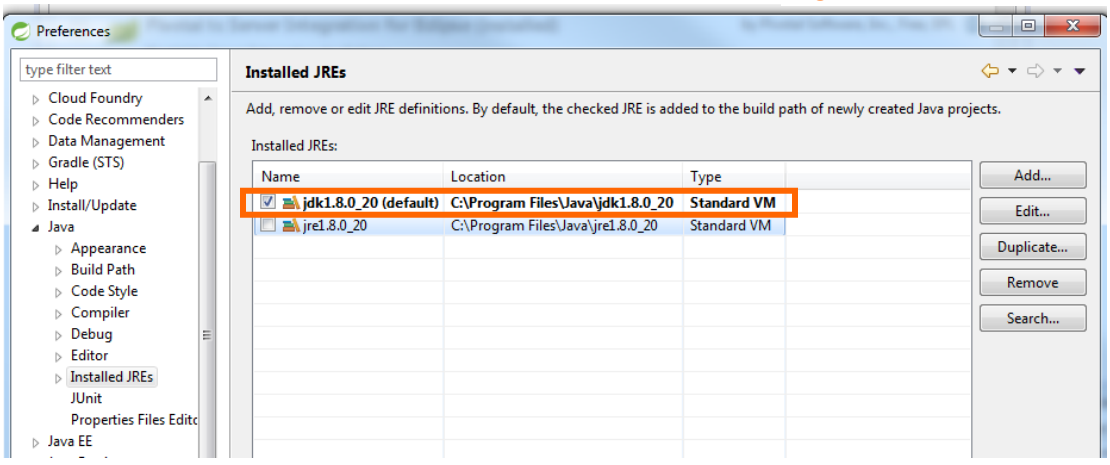
1. In this section, you will download and configure STS
2. Download the zip file of the latest version of STS from https://spring.io/tools/sts/all
3. The zip file will probably then be located in your Downloads folder. Find it and upzip it to your C-Drive
4. Drive down into C:\sts-bundle\sts-3.7.3.RELEASE
5. Run the STS.exe
    - You can pin this to your start menu or desktop for easier access in the future
6. Create a workspace in your preferred location (ex. C:/Users/you/Data/workspace)
7. Inside STS, click the Window option in the menu bar and follow this path:
    - Window – Preferences – General – Network Connections
8. Change Active Provider to Manual (See Figure 1)
9. Edit HTTP, HTTPS with this information:
    - Host: proxy.discoverfinancial.com
    - Port: 8080
    - Check Requires Authentication
    - Enter your discover login info
    - DO NOT change anything else
10. Apply changes
11. Give security questions (optional) for password recovery

Figure 1: Network Connections



12. Do not close this window, now find Java – Installed JREs
13. Make sure jdk1.8.0_20 is selected as default (See Figure 2)

Figure 2: Installed JREs



14. Apply changes
15. Do not close this window, now find Java – Compiler
16. Make sure the Compiler compliance level is also 1.8

17. Restart STS
18. Your Dashboard should now have articles, if not, make sure you have internet access on your normal browser. Also double check the inputs you made in the network window
19. On the Dashboard page, scroll down to the Manage Section on the right side, and click IDE Extensions
20. Install Gradle (STS Legacy) Support
21. Next select Help in the menu bar and select Eclipse Marketplace
22. Type "freemarker" into the search bar and then install FreeMarker IDE from JBoss Tools 1.5
23. Now to configure for GitHub
24. In STS, navigate to Window – Preferences – Team – Git – Configuration
25. Select the button Add Entry and enter the following information
    - Key: user.email
    - Value: youremail@discover.com
26. Add another entry with this information
    - Key: user.name
    - Value: your first and last name
27. Apply Changes
28. Now click your Start icon on your Windows Taskbar
29. Type in the search field "environment", and open Edit environment variables for your account, NOT the system variables (See Figure 3)
30. Add a new user variable with the following information (See Figure 4)
    - Variable name: HOME
    - Variable value: %USERPROFILE%

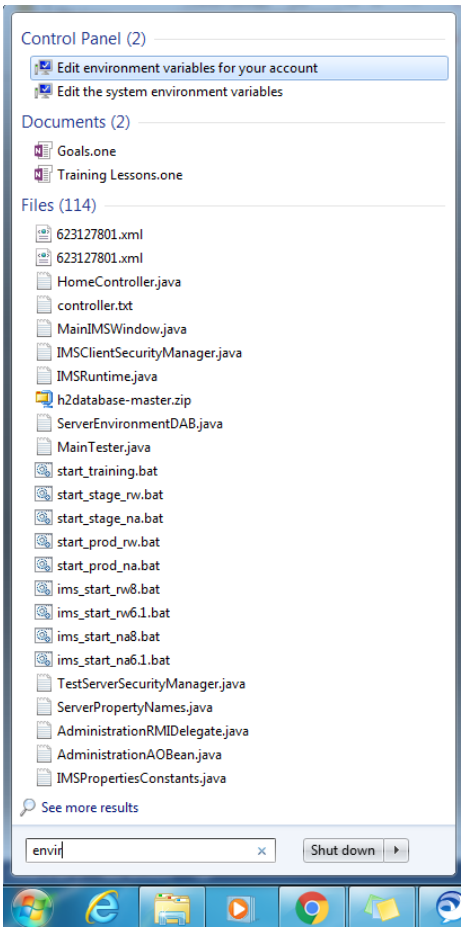


Figure 3: Environment Variables

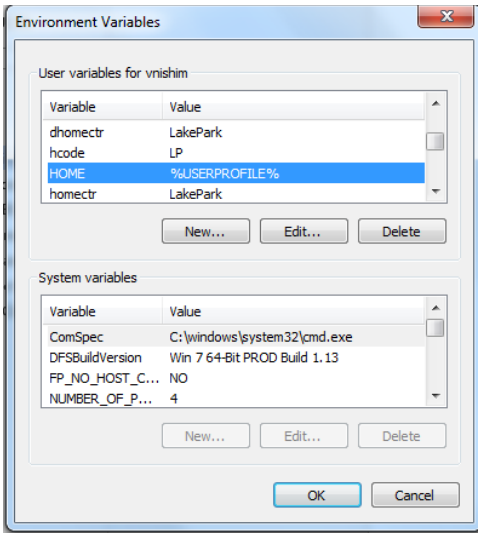


Figure 4: New User Variable

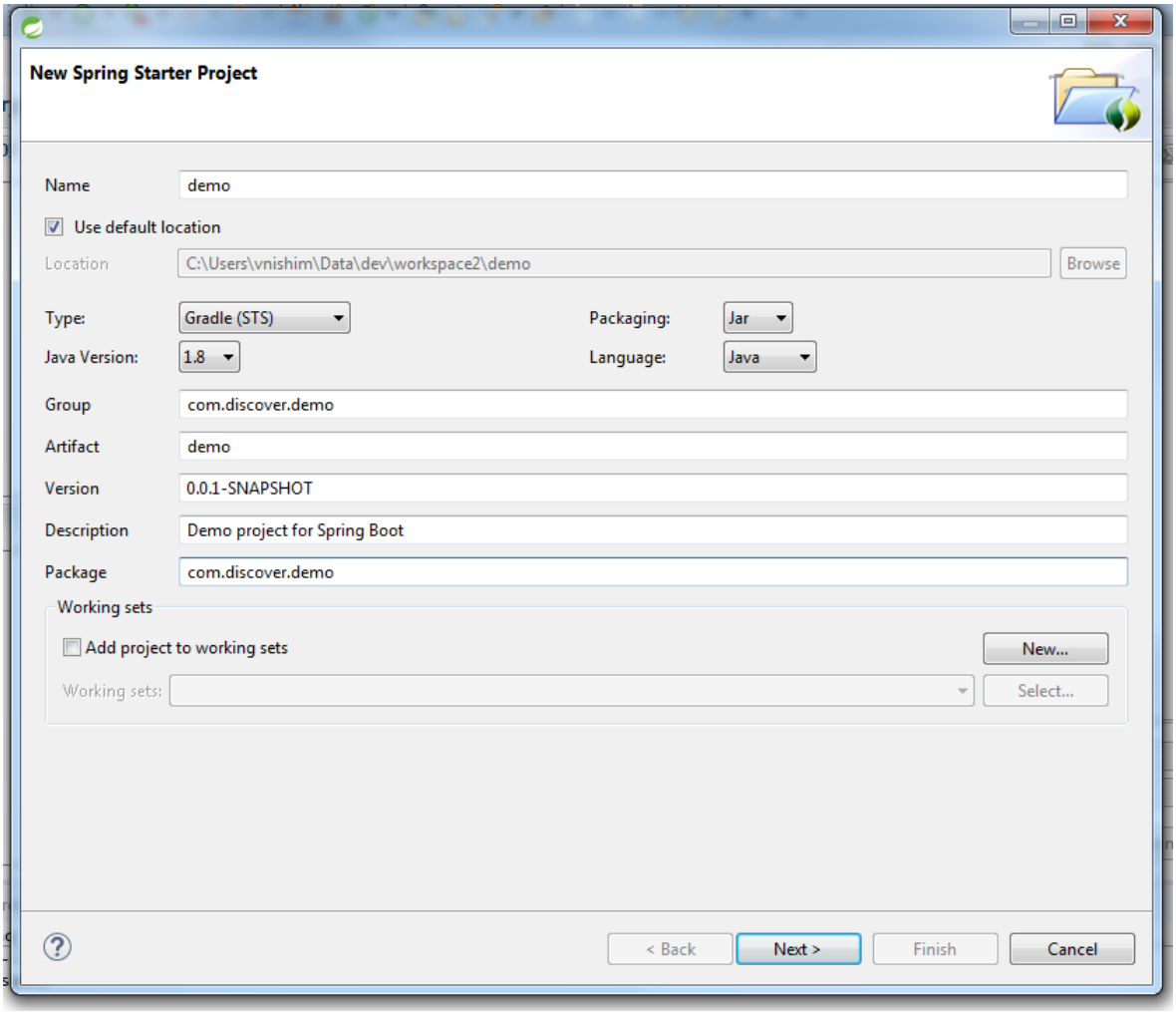
Section 3: Creating a new Spring MVC Project using Gradle

NOTE: alternative is to import existing repository from GitHub to STS, see guide **Using GitHub in STS, Section 2**

1. In this section, you will create a basic MVC project using Gradle, JDBC, and Freemarker.
2. Open STS and click File – New – Spring Starter Project
    - **Note:** If this option is not available, no worries. Select Project instead, a window will appear, scroll down to Spring, expand it and select Spring Starter Project

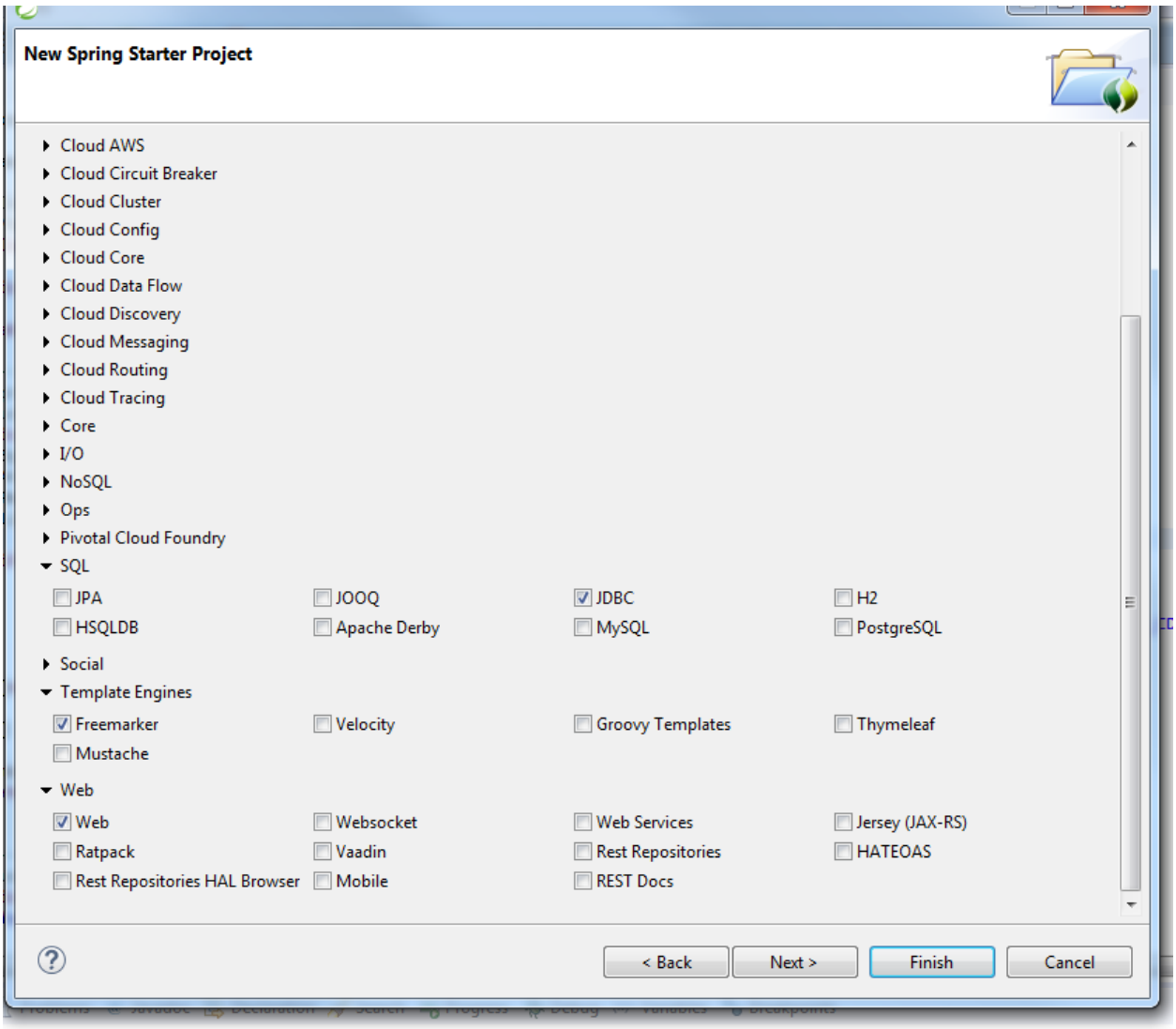
3. A window should appear where you can enter the following information (See Figure 5)
    - Name: demo
    - Type: Gradle (STS)
    - Java Version: 1.8
    - Packaging: Jar
    - Language: Java
    - Group: com.discover.demo [you should probably change]
    - Artifact: demo [this will be the name of your project]
        i. **Note**: I belong to CMA Collections team and I was working on a product called EIM so for mine my group was com.discoverfinancial.cma and my artifact was EIMS. Go to https://nexus.discoverfinancial.com:8443/content/groups/public/com/ to see where you might put your project
    - Version: 0.0.1-SNAPSHOT
    - Description: Demo project for Spring Boot
    - Package: com.discover.demo
    - Click Next

Figure 5: New Spring Starter Project



4. The window will now change to a list of dependencies, open and select the following (See Figure 6)
    - SQL: JDBC
    - Template Engines: Freemarker
    - Web: Web

Figure 6: New Spring Project Dependencies



5. Click Finish
6. Awesome! Now we have our project. There is probably an error, though. Let's fix it.
7. Open your build.gradle and delete the following section.

```
eclipse {
        classpath {
                containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')
```

```
                containers
'org.eclipse.jdt.launching.JRE_CONTAINER/org.eclipse.jdt.internal.debug.ui.launcher.StandardVM
Type/JavaSE-1.8'
        }
}
```

8. Now change the following bolded orange lines to your build.gradle. This will integrate Nexus. The maven {url nexusPublicRepoURL} will retrieve resources from Nexus and the uploadArchives will upload your jar file to Nexus. You also need to comment out the JDBC dependency because we are not ready for it yet. DON'T delete it, though.

```
buildscript {
        ext {
                springBootVersion = '1.3.5.RELEASE'
        }
        repositories {
                maven {url nexusPublicRepoURL}
        }
        dependencies {
                classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
        }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'spring-boot'
apply plugin: 'maven'

jar {
        baseName = 'demo'
        version = '0.0.1-SNAPSHOT'
}

sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
        maven {url nexusPublicRepoURL}
}


dependencies {
        compile('org.springframework.boot:spring-boot-starter-freemarker')
        //compile('org.springframework.boot:spring-boot-starter-jdbc')
        compile('org.springframework.boot:spring-boot-starter-web')
        testCompile('org.springframework.boot:spring-boot-starter-test')
}

//***START Nexus Integration **

//*
//** enhances the Gradle uploadArchives task to inject nexus username and passwords
//*
uploadArchives {

    //* variables needed for script execution, but (IMO) uploadArchives should not be run
locally
    def nxUsername = System.getenv("nexusUsername") ?: "nexusUsernameNotSet"
```

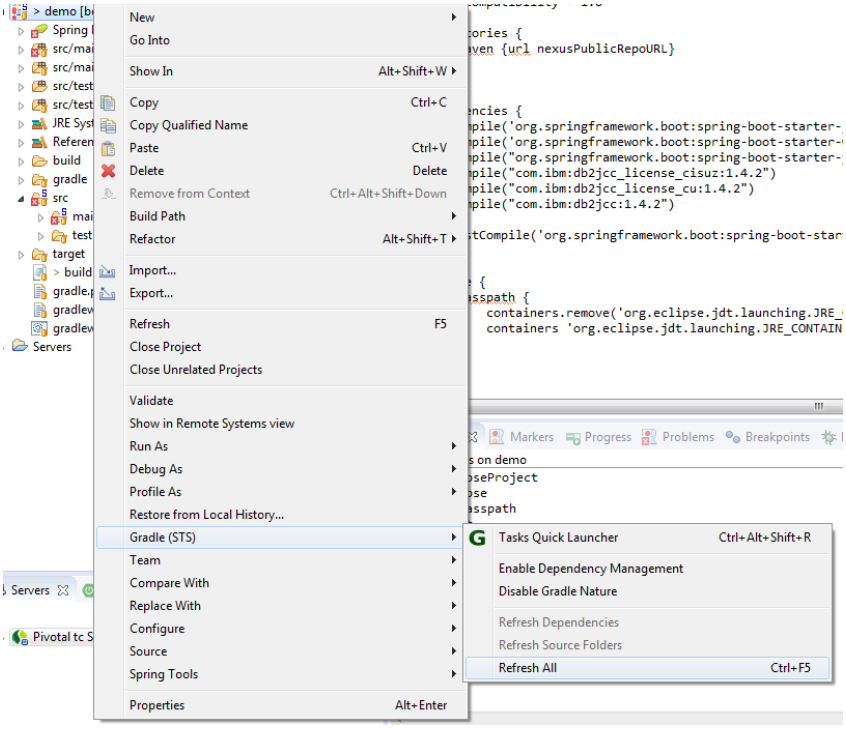
```
    def nxPassword = System.getenv("nexusPassword") ?: "nexusPasswordNotSet"

    repositories {
        mavenDeployer {
            repository(url: System.getenv("nexusReleaseRepoURL")) {
                authentication(userName: nxUsername, password: nxPassword)
            }
            snapshotRepository(url: System.getenv("nexusSnapshotRepoURL")) {
                authentication(userName: nxUsername, password: nxPassword)
            }
        }
    }
}

//***END Nexus Integration
```

9. Save your changes.
10. Right click on the project in your Project Explorer and select New – File
    - Enter File name: gradle.properties
    - Click Finish
11. Enter the following into your newly created gradle.properties file. These are the values of the variables that you use in your build.gradle.

```
nexusPublicRepoURL=https://nexus.discoverfinancial.com:8443/content/groups/public/
nexusReleaseRepoURL=https://nexus.discoverfinancial.com:8443/content/repositories/releases
nexusSnapshotRepoURL=https://nexus.discoverfinancial.com:8443/content/repositories/snapshots
```

12. Save your changes.
13. Now right click on your project and go to Gradle (STS) – Refresh All (See Figure 7)

Figure 7: Gradle Refresh All



14. Now drill into your project and open src – main – java – com – discover – demo
15. Inside the demo folder, create a new folder named “controller”
    - **Note:** If you cannot see your new folder, right click on your project and click Refresh
    - Also, make sure you’re in the Project Explorer, not the Package Explorer
16. Inside this folder, create a file and call it “HomeController.java”
17. In your HomeController, copy and paste in the following:

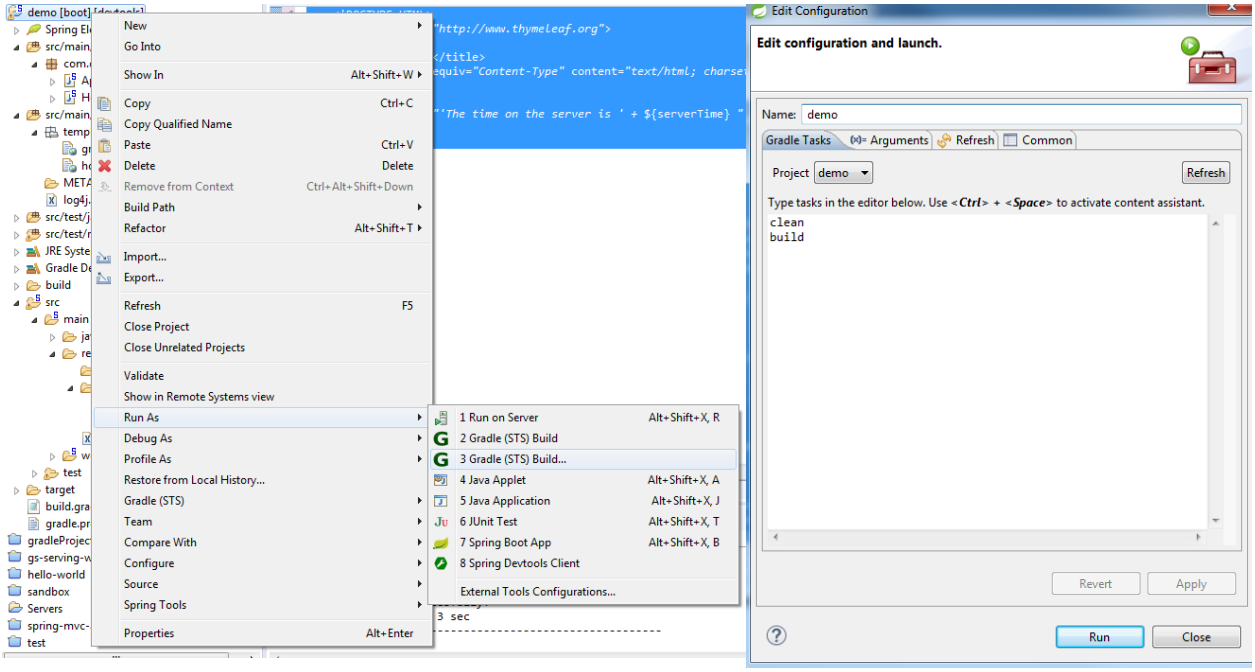
```
package com.discover.demo.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
public class HomeController {
	private static final Logger logger = LoggerFactory.getLogger(HomeController.class);

	@RequestMapping(value = "/", method = RequestMethod.GET)
	public String home(Model model) {
		logger.info("Welcome to index");
		model.addAttribute("name", "[enter your name]");
		return "index";
	}
}
```

18. Now go into src – main – resources – templates and create a file called “index.ftl” and add the following code:

```
<!DOCTYPE HTML>
<html>
<head>
<title>HOME</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
	<p>hello ${name}</p>
</body>
</html>
```

19. Notice there is a static folder also inside resources. This is for your css, js, img files
20. **At this point, you shouldn’t have any errors.** Now right click on the project and select Run As – Gradle (STS) Build… (See Figure 8)
21. A window will appear, type in the following tasks:
    - clean
    - build

Figure 8: Gradle Build…

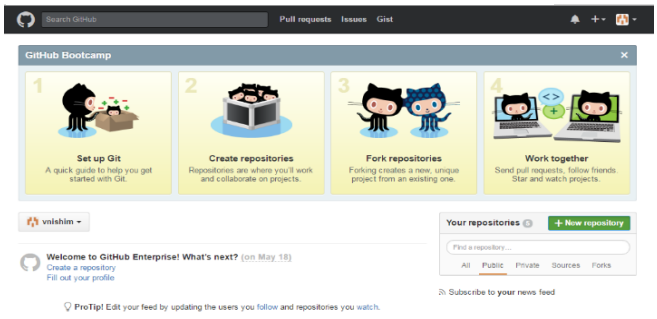


22. Run!
23. Now, right click on your project and go to Debug As – SpringBootApp
24. Open your browser and go to http://localhost:8080/
25. You should get a page that says “hello YourName”
26. Click the red square in the STS console to stop it from running.
27. Awesome! We have created a basic web app!

Section 4: Creating a new repository on GitHub & connecting to STS

1. In this section, you will create a GitHub repository, generate an SSH key, and make your first commit and push to GitHub.
2. Go to this site https://github.discoverfinancial.com/ in your browser (Chrome is preferred)
    - Should log you in automatically, but if not, trying logging in or create an account
3. Create a new repository by clicking the green button on the right hand side (See Figure 9)

Figure 9: GitHub - New Repository



4. Enter the following information:
    - Owner: you
    - Repository name: demo
    - Description: Demo Repository
    - Check Public
    - Create!
5. Leave this open
6. Now Search for Git Bash in your window Start bar. (See Figure 10) Open Git Bash console. We are going to generate an SSH key.
    - **Note:** You only need to do this once. After you put the SSH key in GitHub, you shouldn't have to generate a new one unless you want to.
7. Enter the following into the console, putting in your GitHub email
    - ssh-keygen –t rsa –b 4096 –C you@discover.com
8. It will then generate your key, Wait for it to load
9. It will then prompt you to enter a file to save the key, don't put anything and simply press enter
10. It will then prompt you for a passphrase, Enter a password you will remember
11. Next you need to ensure ssh-agent is enabled.
12. Stay in Git Bash and enter the following command:
    - eval "$(ssh-agent –s)"
13. Now to copy the key into GitHub
14. Stay in Git Bash and enter the following command:
    - Clip < ~/.ssh/id_rsa.pub
    - This copies the key into your clipboard
15. Now go to your GitHub account and edit your settings (See Figure 11)

Figure 10: Git Bash

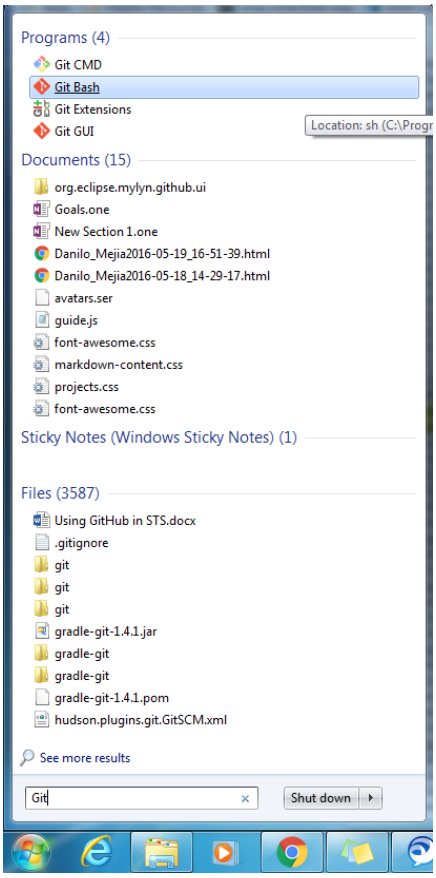
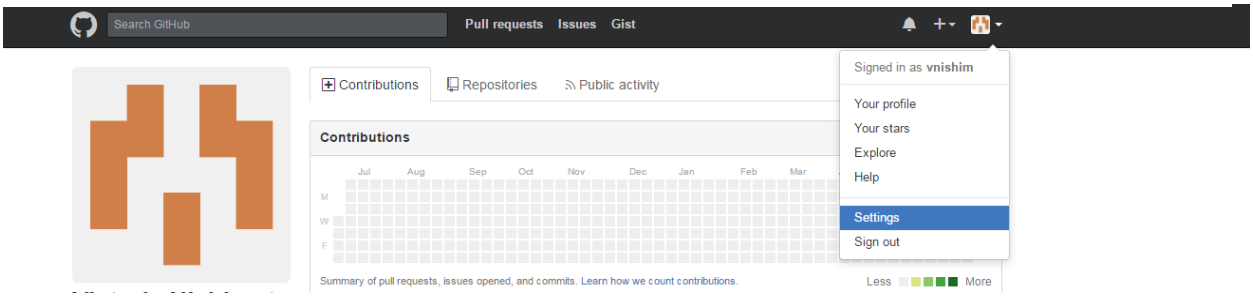


Figure 11: GitHub Settings

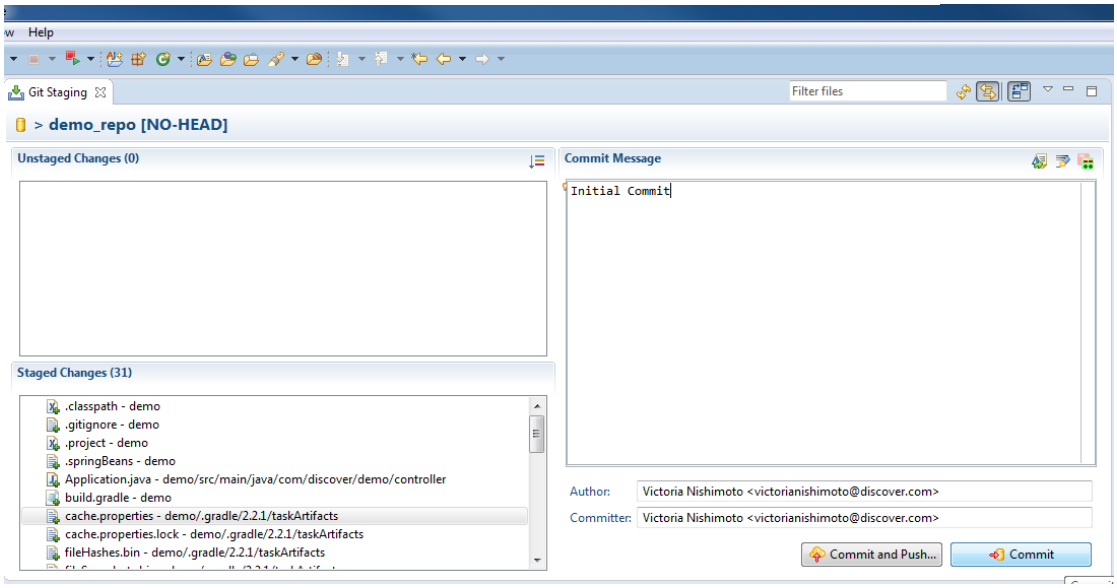


16. Click SSH keys on the left hand side and click Add SSH key
17. Give it any title you want
18. Paste your key into the textbox and click Add key
19. Enter your discover username and password at the prompt

- **Note:** You will get an email confirmation that you added an SSH key

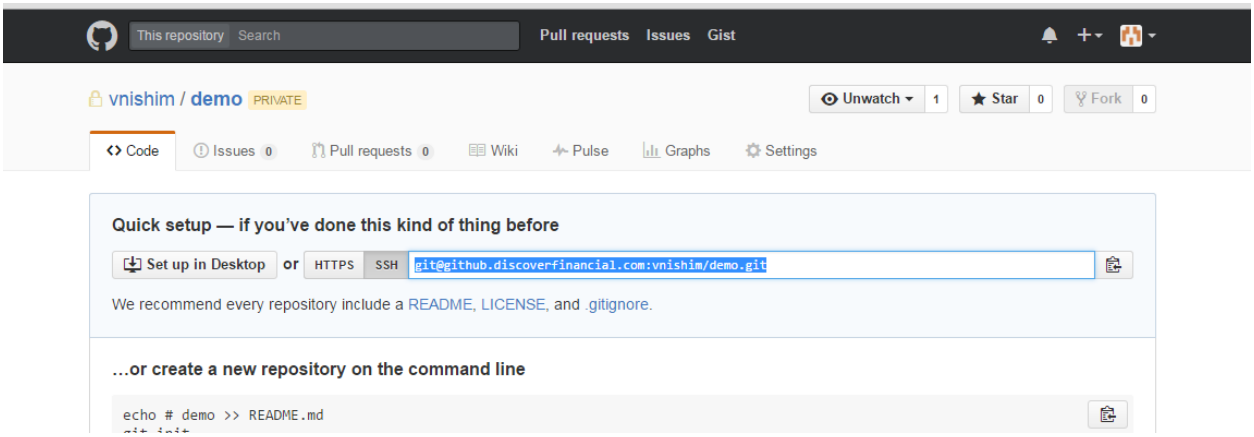
20. Now in STS, right click on your project and select Team – Share Project
21. Click the checkbox at the top that says, "Use or create repository in parent folder of project"
22. Click on your project and then Create Repository button
23. Click Finish
24. Right click on your project and select Team – Add to Index
25. Click the down arrow on your Package Explorer window filters and uncheck "*.resources"
26. Right click on your project, select Team – Commit
27. Grab any Unstaged Changes and drag them into Staged Changes (See Figure 12)
28. Type "Initial commit" as the message

Figure 12: Git Staging



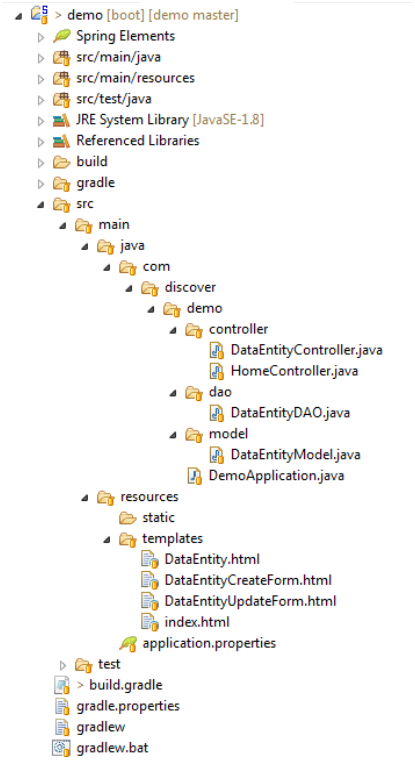
29. Click commit
30. Go back to your demo repository in GitHub and select the SSH option and copy the URL (See Figure 13)

Figure 13: GitHub SSH URL



31. Now go back to STS, right click on your project and select Team – Remote – Push
32. In the Destination Git Repository window, paste your link in the URI: text field, the rest of the fields should then auto populate.
33. Don't worry about the Authentication
34. Press Next and it will ask you for your passphrase, this is the password (probably your discover password) that you put into Git Bash for your ssh key
35. Press enter and click these buttons:
    - Add All Branches Spec
    - Add All Tags Spec
36. Click Finish
37. Click OK
38. Now go back to your GitHub demo repository, refresh the page, and your first commit should be there! GitHub should have auto-created a master branch for you, as well.
39. One more step. Before we begin any work or make any changes to our code, we need to create a new branch. **You should almost never make changes straight to your master branch.**
40. In STS, right click on your project and select Team – Switch To – New Branch
41. You can call your branch anything, but I would recommend calling it "development"
42. Do not change anything else, and click Finish

Figure 14: Final File Tree



Section 5: Using MVC with JDBC & CRUD

1. In this section, we will be creating a JDBC template that will use a datasource to connect to DB2. You can easily change the datasource, though. We will also create a Controller, View, and Model that will connect to the datasource to create, read, update, and delete from.
2. For references, at the end, your Project Explorer should look like Figure 14

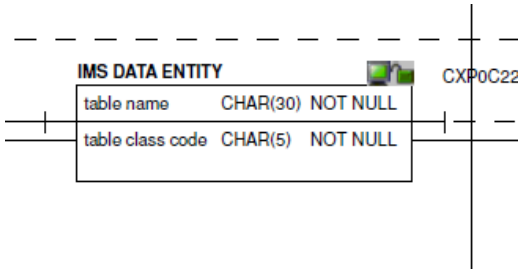
Creating the DAO and connecting the datasource

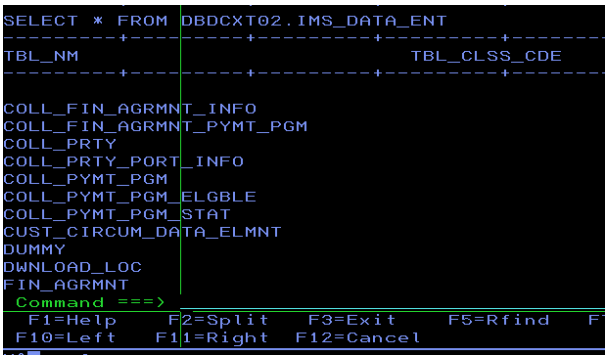
1. First you need to make the following bolded orange changes to your build.gradle dependencies

```
dependencies {
        compile('org.springframework.boot:spring-boot-starter-freemarker')
        compile('org.springframework.boot:spring-boot-starter-jdbc')
        compile('org.springframework.boot:spring-boot-starter-web')
        testCompile('org.springframework.boot:spring-boot-starter-test')
        compile("com.ibm:db2jcc_license_cisuz:1.4.2")
        compile("com.ibm:db2jcc_license_cu:1.4.2")
        compile("com.ibm:db2jcc:1.4.2")
}
```

2. Now save your build.gradle and right click on your project and select Gradle (STS) – Refresh All
3. Now just for your understanding, here is the logical model view and a query of the table from db2 (See Figure 15)

Figure 15: Example Table Information





4. Next is to create the DAO. Right click on your com – discover – demo folder and select New – Folder
5. Name it "dao" and click Finish
6. Now right click on this folder and select New – File
7. Name the file DataEntityDAO.java. Enter the following code into this file:

```
package com.discover.demo.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import com.discover.demo.model.DataEntityModel;

@Repository
public class DataEntityDAO {

	@Autowired
	private JdbcTemplate jdbcTemplate;

	//create
	public void create(DataEntityModel entity){
		String sql = "INSERT INTO DBDCXT02.IMS_DATA_ENT ( TBL_NM, TBL_CLSS_CDE ) VALUES
('"
				+ entity.getTBL_NM() + "','" + entity.getTBL_CLSS_CDE() + "');";
		System.out.println("CREATE DATA ENTITY: " + sql);
		jdbcTemplate.update(sql);
	}

	//read
	public DataEntityModel get(String id) {
```

```
		String sql = "SELECT * FROM DBDCXT02.IMS_DATA_ENT WHERE TBL_NM= '" + id + "'";
		System.out.println("GET A DATA ENTITY: " + sql);
		return jdbcTemplate.query(sql, new ResultSetExtractor<DataEntityModel>() {

			@Override
			public DataEntityModel extractData(ResultSet rs) throws SQLException,
			DataAccessException {
				if (rs.next()) {
					DataEntityModel d = new DataEntityModel();
					d.setTBL_NM(rs.getString("TBL_NM").trim());
					d.setTBL_CLSS_CDE(rs.getString("TBL_CLSS_CDE").trim());
					return d;
				}
				return null;
			}
		});
	}
	public List<DataEntityModel> list(){
		String sql = "SELECT * FROM DBDCXT02.IMS_DATA_ENT";
		System.out.println("LIST ALL DATA ENTITIES: " + sql);

		List<DataEntityModel> listDataEntities = jdbcTemplate.query(sql, new
RowMapper<DataEntityModel>(){
			public DataEntityModel mapRow(ResultSet rs, int rowNum) throws
SQLException{
				DataEntityModel d = null;
				int index = 0;
				d = new DataEntityModel(rs.getString(++index).trim(),
rs.getString(++index).trim());
				return d;
			}
		});
		return listDataEntities;
	}

	//update
	public void update(DataEntityModel entity){
		String sql = "UPDATE DBDCXT02.IMS_DATA_ENT "
				+ "SET TBL_CLSS_CDE='" + entity.getTBL_CLSS_CDE() + "' "
				+ "WHERE TBL_NM='" + entity.getTBL_NM() + "'";
		System.out.println("UPDATE DATA ENTITY: " + sql);
		jdbcTemplate.update(sql);
	}

	//delete
	public void delete(DataEntityModel entity){
		String sql = "DELETE FROM DBDCXT02.IMS_DATA_ENT WHERE TBL_NM='"
				+ entity.getTBL_NM() + "' AND TBL_CLSS_CDE='" +
entity.getTBL_CLSS_CDE() + "';";
		System.out.println("DELETE DATA ENTITY: " + sql);
		jdbcTemplate.update(sql);
	}

}
```

8. You will have errors in your file. Ignore then for now. They will be resolved in a few steps.
9. Last thing is to edit our properties file. Inside the src – main – resources folder, open the file called “application.properties”

10. Inside this file, put the following:

```
spring.datasource.url=jdbc:db2://db00dtst-h261.discoverfinancial.com:1300/DB94
spring.datasource.driver-class-name=com.ibm.db2.jcc.DB2Driver
spring.datasource.username=[insert database username here]
spring.datasource.password=[insert database password here]
```

11. For this example, we are using a DB2 user id with access to the DBTCXT02 schema
12. This will change depending on your datasource. You can change this to use MySQL, Postgres, Oracle, etc. This allows your @Autowired annotation to use these values in the JdbcTemplate.

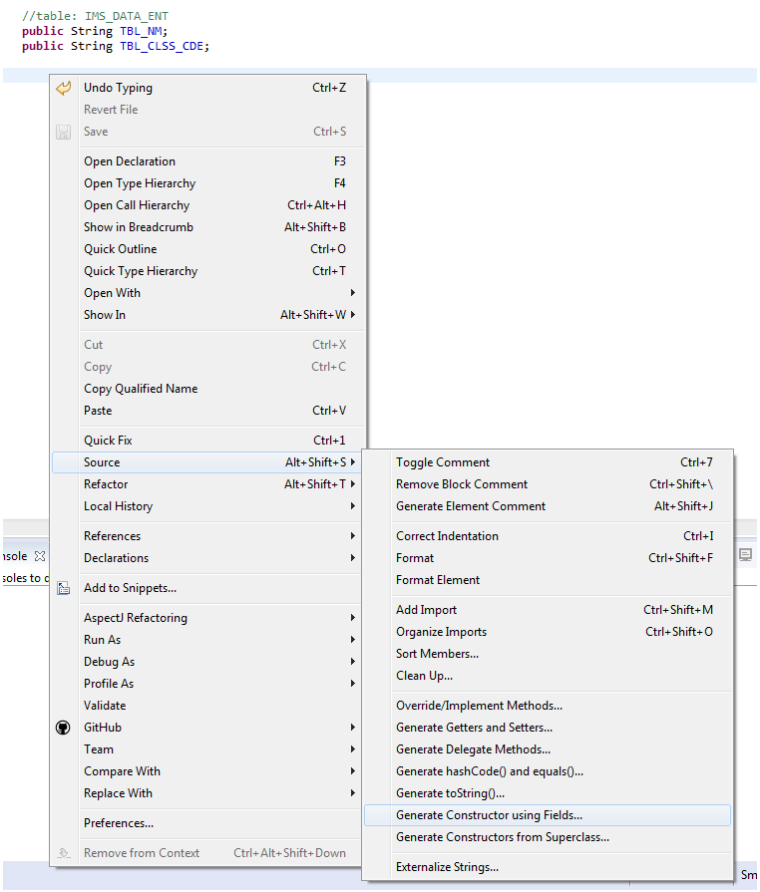
Creating the Model, View, and Controller

1. In STS, drill down into your project to src – main – java – com – discover – demo
2. Right click on the demo folder and select New – Folder
3. Name the folder "model"
4. Refresh your project if the folder doesn't show up
5. Now right click on the model folder and select New – File
6. Name the file "DataEntityModel.java"
7. Copy and paste the following code into your DataEntityModel.java

```
package com.discover.demo.model;

public class DataEntityModel{

        //table: IMS_DATA_ENT
        private String TBL_NM;
        private String TBL_CLSS_CDE;
}
```

8. As you can see, our model matches the table in DB2 from the above section. This is important for CRUD.
9. Okay, next let's generate our constructor and getters and setters
10. Underneath the attributes, right click and select Source – Generate Constructor using Fields (See Figure 16)
11. The fields should already be selected, so just press OK. You should now have a constructor.
12. Also create an empty constructor from superclass, this will be important later.

Figure 16: Generate Constructor using Fields...



13. Underneath the constructor, right click and select Source – Generate Getters and Setters
14. Then check both boxes and click OK. You will now have getters and setters.
15. Next is to create the controller. Right click on your controller folder and select New – File
16. Name it “DataEntityController.java”
17. Inside your new file, copy and paste the following code:

```
package com.discover.demo.controller;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

import com.discover.demo.dao.DataEntityDAO;
import com.discover.demo.model.DataEntityModel;

@Controller
@RequestMapping("/DataEntity")
public class DataEntityController {

	@Autowired
	private DataEntityDAO dao;

	private static final Logger logger =
LoggerFactory.getLogger(DataEntityController.class);

	//-------------------Retrieve All Data Entities-------------------//
	@RequestMapping(value = "", method = RequestMethod.GET)
	public String list(Model model) {
		logger.info("List DataEntity Items");

		List<DataEntityModel> dataEntityList = dao.list();

		model.addAttribute("list", dataEntityList);
		return "DataEntity";
	}

	//-------------------Create a Data Entity-------------------//
	@RequestMapping(value= "/create", method = RequestMethod.GET)
	public String create(Model model){
		logger.info("Create a DataEntity Item Form");

		return "DataEntityCreateForm";
	}

	@RequestMapping(value= "/create", method = RequestMethod.POST)
	public String create(DataEntityModel DEM, Model model){
		logger.info("Create a DataEntity Item Submit");

		dao.create(DEM);

		return "redirect: ";
	}
```

```
	//-------------------Update a Data Entity-------------------//
	@RequestMapping(value= "/update", method = RequestMethod.GET)
	public String update(@RequestParam(value="id") String id, Model model){
		logger.info("Update a DataEntity Item Form");

		DataEntityModel DEM = dao.get(id);

		model.addAttribute("obj", DEM);
		return "DataEntityUpdateForm";
	}

	@RequestMapping(value= "/update", method = RequestMethod.POST)
	public String update(DataEntityModel DEM, Model model){
		logger.info("Update a DataEntity Item Submit");

		dao.update(DEM);

		return "redirect: ";
	}

	//-------------------Delete a Data Entity-------------------//
	@RequestMapping(value= "/delete", method = RequestMethod.GET)
	public String delete(@RequestParam(value="id") String id, Model model){
		logger.info("Delete a DataEntity Item");

		DataEntityModel DEM = dao.get(id);
		dao.delete(DEM);

		return "redirect: ";
	}
}
```

18. As you can see, there are GET and POST methods. When you have @RequestParam in the method signature, it is probably going to be a GET method. When you have an object in the signature, it will probably be a form submission, which is a POST method. To pass something from the controller to the view, you add it to the model.
19. Next we need to make some Views. So inside your resources – templates folder, create a new file called DataEntity.ftl
20. Inside the file, put the following:

```
<!DOCTYPE HTML>
<html>
<head>
<title>HOME</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
	<div>
		<a href="DataEntity/create">Create</a>
		<table style="width: 100%">
			<thead>
				<tr>
					<th>Table Name</th>
					<th>Table Class Code</th>
					<th>Actions</th>
				</tr>
			</thead>
```

```
                        <tbody>
                        <#list list as l>
                                <tr>
                                        <td>${l.getTBL_NM()}</td>
                                        <td>${l.getTBL_CLSS_CDE()}</td>
                                        <td>
                                                <a href="update?id=${l.getTBL_NM()}">Edit</a> |
                                                <a href="delete?id=${l.getTBL_NM()}">Delete</a>
                                        </td>
                        </#list>
                        </tbody>
                </table>
        </div>
</body>
</html>
```

21. This view will display a table. For each row in the table, you get the DataEntity object from the model value that was passed and get its values. There is also a link that will go to the create method in the controller. This is using Freemarker to get and pass the data.
22. Create another view called "DataEntityCreateForm.ftl" and put the following code in it:

```
<!DOCTYPE HTML>
<html>
<head>
<title>Form</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
        <div>
                <form action="create" method="post">
                        <label>Name: </label>
                        <input name="TBL_NM" />
                        <label>Code: </label>
                        <input name="TBL_CLSS_CDE" />
                        <input type="submit" value="Save" />
                </form>
        </div>
</body>
</html>
```

23. This view will have fields that match the DataEntityModel. In the controller, the GET create method will pass an empty DataEntityModel object and in the view, you will fill in the values for the attributes. When you submit it, it will then save it to the database.
24. Create another view and call it "DataEntityUpdateForm.ftl"
25. Copy and paste the following:

```
<!DOCTYPE HTML>
<html>
<head>
<title>Form</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
        <div>
                <form action="update" method="post">
                        <label>Name: </label>
```

```
                    <input name="TBL_NM" value=${obj.getTBL_NM()} />
                    <label>Code: </label>
                    <input name="TBL_CLSS_CDE" value="${obj.getTBL_CLSS_CDE()}" />
                    <input type="submit" value="Save" />
                </form>
        </div>
</body>
</html>
```

26. This is very similar to the create method, except the form will autofill because you're not passing an empty object.
27. Alright, you're all set! Let's test it out to make sure everything is working.
28. Right click on your project and select Debug As – Spring Boot App
29. Now go to your browser and navigate to http://localhost:8080/DataEntity
30. You should have a list of Data Entities with create, edit, and delete capabilities. Yay!

Section 6: Integrating SonarQube & JUnit Testing

1. In this section, you will learn how to add the SonarQube plugin to your project. Also, you will add some JUnit tests that specifically test SpringMVC.
2. Open your project in STS and open your build.gradle
3. Add the following classpath to your buildscript
    - `classpath("com.discover.esqmci:discover-project-plugin:latest.release")`
4. Now add the following plugin
    - `apply plugin: 'discover'`
5. Save your build.gradle
6. Now for the JUnit testing.
7. Make sure you have the following dependency in your build.gradle (you should)
    - `testCompile('org.springframework.boot:spring-boot-starter-test')`
8. Next, drill down into your project to src – test – java – com – discover – demo – Demo ApplicationTests.java
9. Inside your DemoApplicationTests.java file, add the following orange text:

```
package com.discover.demo;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.TestPropertySource;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = DemoApplication.class)
```

```
@WebAppConfiguration
@TestPropertySource("/application.properties")
public class DemoApplicationTests {

	private MockMvc mvc;

	@Autowired
	private WebApplicationContext ctx;

	@Before
	public void contextLoads() throws Exception {
		System.out.println("TEST: Set up Web App Context");
		mvc = MockMvcBuilders.webAppContextSetup(ctx).build();
	}

	@Test
	public void list() throws Exception {
		System.out.println("TEST: List DataEntity Items");
		mvc.perform(get("/DataEntity"))
			.andExpect(view().name("DataEntity"))
			.andExpect(model().attributeExists("list"))
			.andExpect(status().isOk());
	}

	@Test
	public void CRUD() throws Exception {
		create();
		update();
		delete();
	}

	public void create() throws Exception {
		System.out.println("TEST: Create a DataEntity Item");
		mvc.perform(post("/DataEntity/create").param("TBL_NM",
"name").param("TBL_CLSS_CDE", "code"))
			.andExpect(status().isFound())
			.andExpect(redirectedUrl(" "));
	}

	public void update() throws Exception {
		System.out.println("TEST: Update a DataEntity Item");
		mvc.perform(get("/DataEntity/update?id=name"))
			.andExpect(status().isOk());
		mvc.perform(post("/DataEntity/update").param("TBL_NM",
"name").param("TBL_CLSS_CDE", "test"))
			.andExpect(status().isFound());
	}

	public void delete() throws Exception {
		System.out.println("TEST: Delete a DataEntity Item");
		mvc.perform(get("/DataEntity/delete?id=name"))
			.andExpect(status().isFound())
			.andExpect(redirectedUrl(" "));
	}
}
```

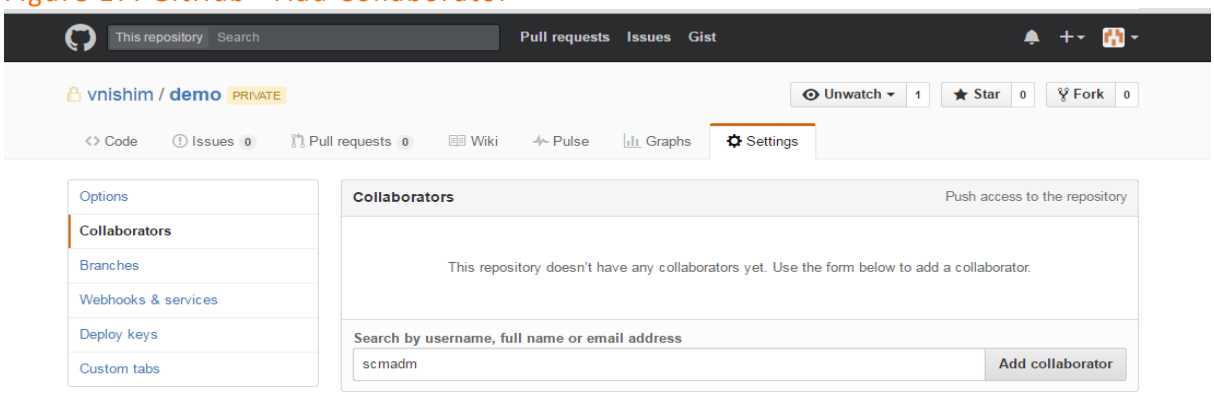
10. To then test this on your localhost, right click on your project and select Debug As – JUnit Test
11. A window will then appear that will tell you how many tests were run, how many errors were found, and how many failures occurred.

12. Okay you are now set up to use SonarQube and JUnit in Jenkins. Continue to the next section, and we will talk about these a little more.

Section 7: Building your project with Jenkins

1. Log in to Jenkins at https://jenkm1.discoverfinancial.com:8443/login?from=%2F
2. After you log in, it should display a list of folders you have developer rights to. Click the folder you want to put your project in
3. On the left-hand-side there are a variety of options you can select, pick "New Item"
4. Enter the name of your project and select Freestyle Project
    - **Note:** you might get an error message that says access denied. Just ignore it and click OK
5. It will then redirect you to the configuration page, you can change this anytime by clicking Configure on the left-hand-side
6. Enter the following information into your configurations:
    - Project Name: <enter the name of your project>
    - Check Discard Old Builds
        i. Strategy: Log Rotation
        ii. Days to keep builds: <enter the number of days you want to keep your builds>
        iii. Max # of builds to keep: <enter how many builds you want to keep>
    - Check GitHub project
        i. Go back to your GitHub account and select the SSH option and copy (See Figure 13)
        ii. Project url: (ex. git@github.discoverfinancial.com:vnishim/demo.git/)
    - JDK: jdk8
        i. Select the JDK you use in your project
    - Select Git
        i. Before you can fill this out, you need to edit your repository settings a little.
        ii. Open your repository on GitHub and click the settings button
        iii. Then select Collaborators on the right-hand-side and enter "scmadm" into the Search bar
        iv. Add collaborator (See Figure 17)

Figure 17: GitHub - Add Collaborator



v. Now in Jenkins, you can enter your information and select scmadm as your credentials
vi. Repository URL: (ex. git@github.discoverfinancial.com:vnishim/demo.git/)
vii. Credentials: scmadm (scmaadm)
viii. Branches to build: */master
ix. Repository browser: Auto

- Check Inject environment variables to the build process
- Check Inject passwords to the build as environment variables
  i. Global passwords: checked
  ii. Mask password parameters: checked
- Click the Add build step dropdown and select “Invoke Gradle script”
- Select “Use Gradle Wrapper”
  i. Make gradlew executable: checked
  ii. Tasks: clean build sonarqube
- Add post-build action from the drop down: Archive the artifacts
  i. Files to archive: **/build/**/*
  ii. Note: you will get an error, this is ok, just ignore it.
- Add post-build action from the drop down: Publish JUnit test result report
  i. Test report XMLS: **/build/test-results/*.xml
  ii. Health report amplification factor: 1.0

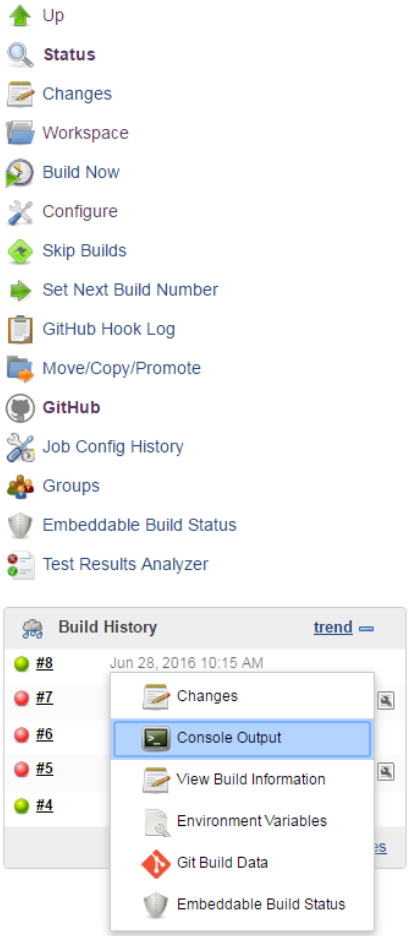
7. Apply and Save
8. Next, go to your project in STS. We need to add a gradle wrapper and then merge with the master
9. To do this, right click on your project and select Gradle (STS) – Tasks Quick Launcher and then type in “wrapper”
10. Press enter, wait for it to run, and now you should have a wrapper.
11. Go into your build.gradle and now add this to configure your wrapper

```
task wrapper(type: Wrapper) {
     gradleVersion = '2.12'
```

```
    distributionUrl =
"https://nexus.discoverfinancial.com:8443/content/groups/public/org/gradle/gradle/2.12/gradle-
2.12-all.zip"
}
```

12. Okay so right now if you've been following this tutorial, you're probably on the development branch. We want Jenkins to build our master branch though, so we need to merge our changes. To do this, make sure you've pushed your latest commits to the development branch. (See Section 4, #25-28)
13. Right click on your project and select Team – Switch To – master
14. Then right click on your project and select Team – Merge.
15. A window will appear. Click on the development branch and then click Merge
16. Now push just like before. (See Section 4, #30-36)
17. Go back to Jenkins and click the "Build Now" option on the left-hand-side.
18. Click the little dropdown arrow next to the build time and select Console Output (See Figure 18).
19. Near the bottom of your build there should be a line that says "ANALYSIS SUCCESSFUL, you can browse [link]"
20. Click the link and it will redirect your SonarQube results.
13. Also if you go Back to Project, you should be able to see your test results by clicking the Test Results Analyzer option, on the left-hand-side
14. You can also see your test results when you open Workspace – build – reports/tests – index.html
21. Before you continue, make sure you get a successful build.

Figure 18: Jenkins - Console Output



Section 8: Integrating Pivotal Cloud Foundry

1. In this section, you will integrate PCF into your build.gradle and your Jenkins build configurations
2. First we need to change our Jenkins configuration a little
3. In Jenkins, click on your project and click Configure on the right-hand-side.
4. Scroll down to the Build section where you invoke your Gradle script. There will be a text-field labeled "Switches". In this field, enter the following:

```
-PnexusPublicRepoURL=${nexusPublicRepoURL}
-PnexusReleaseRepoURL=${nexusReleaseRepoURL}
-PnexusSnapshotRepoURL=${nexusSnapshotRepoURL}
-PnexusUsername=${nexusUsername}
-PnexusPassword=${nexusPassword}
-PcfUsername=${cfUsername}
-PcfPassword=${cfPassword}
```

5. Then change your Tasks to
    - clean

- build
- cfPush
- uploadArchives

6. Apply and Save the changes
7. To deploy your application to the cloud foundry, you also need to make a few changes to your gradle configuration, so go back to your project in STS
8. Right click on your project and select Team – Switch To – development
9. Now open your build.gradle and add the following dependency to your buildscript
    - `classpath("org.cloudfoundry:cf-gradle-plugin:1.1.0")`
10. Next you need to add the following plugin after the other plugins
    - `apply plugin: 'cloudfoundry'`
11. This next step can differ from project to project. You need to make sure you have access to the Jenkins and PCF. You don't need this until you are ready to build and deploy with Jenkins. This is to simply create and push your project to an application in the development space. You will potentially need to replace most of information, except the username, password, trustSelfSignedCerts. Put it at the end of your build.gradle
    - Note: After adding the cloudfoundry to your build.gradle, if you try to build in STS, it will fail. You can still run your app through SpringBoot but you can't build it.

```
cloudfoundry {
    target = "https://api.cfd2.discoverfinancial.com"
    host ="demo"
    domain = "cfd2.discoverfinancial.com"
    username = cfUsername
    password = cfPassword
    application="demo"
    trustSelfSignedCerts = "true"
    buildpack = "java_buildpack_offline_discover_OracleJRE8_jbp3_1"   //don't let CF
determine...specify exact version
    space = "development"
    organization = "this-is-just-a-test-org"
    memory = 1024
    instances = 1
}
```

12. Okay now to commit and push these changes to the development branch and then merge it back into the master and then push the master to GitHub.
13. To commit, See Section 4, #26-29
14. To push, See Section 4, #30-37
15. To merge, See Section 6, #12-16
16. If it does not build automatically, click the "Build Now" option on the right-hand-side in Jenkins
17. Click the little dropdown arrow next to the build time and select Console Output. (See Figure 18)
18. If your build succeeded there will be a link that you can then see your application on, congratulations!
19. If it failed, you can use the Console Output to try and see what's wrong. Here is a couple of possible errors and how to fix them.

- If the Console Output displays a bunch of “0 of 1 instances running (1 crashed)” listed at the bottom, abort the build and go to Pivotal Apps Manager. Look at the logs and your error will probably be found there.
- If the Console Output displays a message like “Error calling Cloud Foundry: 400 Bad Request: The host is taken” trying changing the name of the host in your build.gradle
- If your build gets stuck on a download or your gradlew, abort the build and look at your build.gradle, your error will probably be there.

Congratulations!

You’re all done! You have created a Spring MVC project, integrated Gradle, built it with Jenkins, uploaded to Nexus, and deployed to PCF. You can see my repository at https://github.discoverfinancial.com/vnishim/demo