Fleet Management Application using Spring Boot

Table of Contents

[Introduction to Spring Boot 3](#_Toc34419530)

[Spring Boot Primary Goals 3](#_Toc34419531)

[Why New Project Need Spring Boot? 3](#_Toc34419532)

[Getting started with Spring Boot 4](#_Toc34419533)

[Spring Boot and Spring ModelViewController (MVC) 4](#_Toc34419534)

[Spring Boot MVC auto-configuration 4](#_Toc34419535)

[Fleet Management Application using spring boot 4](#_Toc34419536)

[Build with Maven 4](#_Toc34419537)

[Starters 4](#_Toc34419538)

[Servlet containers support 5](#_Toc34419539)

[Template engines support 5](#_Toc34419540)

[Caching support 5](#_Toc34419541)

[Design Patterns used in Spring Framework 5](#_Toc34419542)

[Fleet Management Project Details: 7](#_Toc34419543)

[Requirements for the Fleet Management web application: 8](#_Toc34419544)

[The User should be able to login. 8](#_Toc34419545)

[The User should be able to view the total number of fleets each customer has from the customer search. 8](#_Toc34419546)

[The User should be able to view all the Fleets. 8](#_Toc34419547)

[The User should be able to view all the Customers. 8](#_Toc34419548)

[The User should be able to create Customers. 8](#_Toc34419549)

[The User should be able to create Fleets. 8](#_Toc34419550)

[The User should be able to edit and delete the Customer 8](#_Toc34419551)

[Features: User 8](#_Toc34419552)

[User Module 8](#_Toc34419553)

[Login Details: 8](#_Toc34419554)

[Use Cases: 8](#_Toc34419555)

[Class Diagram: 9](#_Toc34419556)

[Data Access Layer 9](#_Toc34419557)

[Service Layer 10](#_Toc34419558)

[Model 10](#_Toc34419559)

[View 10](#_Toc34419560)

[Controller: 10](#_Toc34419561)

[DAO (Data Access Object) Pattern 11](#_Toc34419562)

[Factory Method Pattern is used 13](#_Toc34419563)

[ModelViewController (MVC) Pattern 13](#_Toc34419564)

[Singleton Pattern 16](#_Toc34419565)

[Note: 16](#_Toc34419566)

[JUnit Testing: 17](#_Toc34419567)

[Integration Testing: 17](#_Toc34419568)

[Challenges during this course of learning and being successful: 19](#_Toc34419569)

[Some of the errors I got during my coding. 19](#_Toc34419570)

[Fleet Management Application Screenshots: 20](#_Toc34419571)

# Introduction to Spring Boot

Spring Boot is the latest innovation from Pivotal Team (The Spring Team) to keep up to date with the changing technology needs. The primary motivation behind developing Spring Boot is to simplify the process for configuring and deploying the spring applications. Spring Boot is not a framework, with Spring Boot, spring applications can be developed with more agility and be able to focus on addressing the application’s functionality needs with minimal (or possibly no) thought of configuring spring itself. Spring Boot can be used to create Java applications using java -jar or more traditional war deployments. It also provides a command line tool that runs “spring scripts”.

# Spring Boot Primary Goals

Spring Boot primary goals are:

* Provide a radically faster and widely accessible getting started experience for all Spring development.
* Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.
* Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).
* Absolutely no code generation and no requirement for XML configuration, to avoid XML Configuration completely
* To avoid defining more Annotation Configuration (It combined some existing Spring Framework Annotations to a simple and single Annotation)
* Spring Boot avoid writing lots of import statements
* Provide some defaults to quick start new projects within no time.

# Why New Project Need Spring Boot?

* To ease the Java-based applications Development, Unit Test and Integration Test Process.
* To reduce Development, Unit Test and Integration Test time by providing some defaults.
* Spring Boot increase Productivity.
* When we talk about defaults, Spring Boot has its own opinions. If you are not specifying the details, it will use its own default configurations. If you want persistence, but don’t specify anything else in your POM file, then Spring Boot configures Hibernate as a JPA provider with an HSQLDB database.
* Provides a bunch of non-functional features/solutions that are very much common to large scale projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).

# Getting started with Spring Boot

Spring Boot project is just a regular Spring project that happens to leverage Spring Boot starters and auto-configuration.

The Spring Team (The Pivotal Team) has provided the following three approaches.

* Using the Spring Boot CLI Tool
* Spring STS IDE
* Using Spring Initializr (Website <http://start.spring.io/>)

Spring Boot and Spring ModelViewController (MVC)

Spring Boot is well suited for web application development. You can easily create a self-contained HTTP server using embedded Tomcat, Jetty, or Undertow. Most web applications will use the spring-boot-starter-web module to get up and running quickly.

Spring MVC lets you create special @Controller or @RestController beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP using @RequestMapping annotations.

# Spring Boot MVC auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications

# Fleet Management Application using spring boot

Fleet Management is a Java-Based Application developed using Spring Boot.

# Build with Maven

This application builds on Maven as most of the IDE’s supports. Pom.xml (Project-Object-Model) is generated. Once the Maven is in the classpath, it is easy to create a Java file.

# Starters

Starters are a set of convenient dependency descriptors that can include in this application. The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported a set of managed transitive dependencies. In order to reduce the burden of searching and configuring all the dependencies required for a framework include spring-boot-starter-batch that will import all the required dependencies for the Spring Batch application.

# Servlet containers support

Tomcat, Jetty, and undertow embedded servlet containers are supported out of box to deploy Spring Boot applications to any Servlet 3.0+ compatible container.

# Template engines support

Spring Boot includes auto-configuration support for Thymeleaf

The main goal of Thymeleaf is to provide an elegant and well-formed way of creating templates. Thymeleaf is a Java library. It is a modern server-side XML/XHTML/HTML5 template engine for both web and standalone applications. It has the ability to plug in your functionality and display data and/or text produced by your applications.

In order to achieve this, it is based on XML tags and attributes that define the execution of predefined logic on the DOM (Document Object Model). Instead of explicitly writing that logic as code inside the template, it is relying on intelligent caching of parsed files in order to use the least possible amount of I/O operations during execution.

# Caching support

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies to cache to methods, reducing thus the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker.

Design Patterns used in Spring Framework

Fleet Management application developed using the Spring Framework encounters the GoF (GangofFour) Design Patterns on a daily basis.

The GoF Design Patterns are broken into three categories:

* Creational Patterns for the creation of objects
* Structural Patterns to provide relationship between objects and
* Behavioral Patterns to help define how objects interact.

|  |  |
| --- | --- |
| **Design Pattern** | **Details** |
| **MVC Pattern** | MVC Design Pattern is a software design that separates the following components of a system or subsystem:   * **Model -** Data about the state of the application or its components. May include routines for modification or access. * **View -** An interpretation of the data (model). This is only limited to a visual representation, but could be audio, derived information (e.g. statistics piped into another model object), etc. Furthermore, a single model may have multiple views. * **Control -** Handles external input to the system invoking modifications on the model. The control/view may be closely related (in the case of a UI). However, other external input (such as network commands), may be processed which are completely independent of the view. |
| Proxy Pattern | Spring uses either JDK proxies or CGLIB proxies (if the target object does not implement any interfaces) to create the proxy for a given target bean. Unless configured to do otherwise, Spring AOP performs run-time weaving Suppose we want to log every method entry and exit. This can be achieved by writing log statements in every method at the start and end. But this will require lot of code work. There are various such tasks like Security which need to be applied across all methods or classes. These are known as cross cutting concerns.AOP addresses the problem of cross-cutting concerns, which would be any kind of code that is repeated in different methods and cannot normally be completely refactored into its own module, like with logging or verification. |
| **Factory Pattern** | **This pattern is used to create objects without specifying the exact class to create**. This pattern is used by spring to load beans using Bean Factory and Application context. |
| **Singleton Pattern** | Beans defined in spring config files are singletons by default. **It ensures only one single object instance** |
| **Template method Pattern** | Template method design pattern is to **define as skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior**. These are used extensively to deal with boilerplate (lot of code is written to accomplish minor functionality) repeated code |
| FrontController Pattern | Front Controller is a controller pattern which provides a centralized controller for managing requests. Each client request must go through and be processed by the Front Controller first, no exceptions. All incoming data is delegated to front controller first. Useful for when your application has multiple entry points which you want to centralize through a single point for standardized processing. Spring implements this design pattern using DispatcherServlet, to dispatch incoming requests to the correct controllers. |
| View Helper Pattern | View Helper arranges view components for the user and delegates processing to other business components so the view component doesn't have to contain any processing logic other than logic to present views. Spring makes use of custom JSP tags etc to separate code from presentation in views. |
| Prototype Pattern | The Prototype pattern is known as a creational pattern,as it is used to construct objects such that they can be decoupled from their implementing systems.  It creates objects based on a template of an exsiting object through cloning. |
| DI/IOC Pattern | Dependency Injection/Inversion of Control design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable.  We can implement dependency injection in java to move the dependency resolution from compile-time to runtime. |

# Fleet Management Project Details:

Fleet Management System is developed and customized for fleet organization that has customers multiple locations. This module helps the Users in the organization to track the fleets in multiple locations (Agriculture, and Construction equipment and other commercial vehicles).

Currently, this application supports only to track the accurate number of fleet available in each location. *This is Developed using the MVC architecture and MY SQLServer database and highlighting the design patterns used in the application.*.

Requirements for the Fleet Management web application:

The User should be able to login.

The User should be able to view the total number of fleets each customer has from the customer search.

The User should be able to view all the Fleets.

The User should be able to view all the Customers.

The User should be able to create Customers.

The User should be able to create Fleets.

The User should be able to edit and delete the Customer

Features: User

* View/Edit/Delete Customer Details
* View/Edit/Delete Fleet Details

# User Module

* Login/Registration of User ( Currently, using Spring boot Login process)
* Google Map Integration- I am not sure if I will be able to complete this functionality within a week.

Customer List

Fleet List

# Login Details:

The Web application user can use the admin portal to login as an admin. In the project, user can login as user and for password, when the application is executed, password is auto generated. Copy the password and in Chrome browser enter the URL: localhost: //8080. Springboot login page shows up:

Enter the user page and Password.

# Use Cases:

Use cases are simple and descriptive diagrams and hence easily understood by both developers and product owners.

User

Actor: In my Project, Actor is User.

Description: This application Fleet Management System is developed in Java using Spring Boot Framework. User has Fleets in multiple locations and in order to track his fleet and responsible person for that fleet list.

Here the User can login with the help of MVC pre built security. Home page has two links – One for Customers and one for Fleet. User can Add New customer, Delete a customer. Edit is partially working currently.

User can Search on a customer id and find out the Fleets that belong to him.

User can also find out all the customers and fleet in his Organization. With Further enhancements with the project, Admin will be able to add more users, Geo tracking, and Monitor performances.

# Class Diagram:

Data Access Layer– Fleet Management application consists of Customer, Fleet and CustomerFleet

Data Access objects (Dao**) interface and its implementations** for

CustomerDaoImpl Class, FleetDaoImpl, and CustomerFleetDaoImpl class

This class has @repository annotation which is enabled for performing CRUD operations.

JdbcTemplate and a DataSource object is injected via the constructor.

RowMapper to map arrow in the result set to a POJO object.

Need to make sure that the class fields declared must match the column names in the database table.

Use of ResultSetExtractor is to extract a single row as a POJO.

getDataSource() method returns a configured DataSource bean. Here I entered MySQL Server login details.

The getCustomerDao() method returns an implementation of the CustomerDao interface which is the CustomerDaoImpl class. This bean is injected to the HomeController class.

Service Layer **consist** of one Service interface and its implementations class

@Service annotation is used. This layer is optional. It is recommened to perform additional business logic if any. It is generally used to connect with repository for CRUD operations.

FleetApplication.java is a startup class in the package Fleet. This class is configured and start up boot application. @SpringBoot application apples the default config settings for my Spring boot Application. **run will start an embedded Tomcat Web application server**.

Modelclass: Customers, Fleet- Customer.java,FleetInfo.java, CustomerFleet.java and FleetApplication.java. Model class maps a row in the table Customer, Fleetinfo and CustomerFleet to a plain-old-java –object (POJO).

View Template .jsp – Source code of the .jsp page is written to display the customer list, fleet list and the action links for performing CRUD operations. Depending on your applications (.jsp) and html files are created. Here the JSP (Java Server Pages) uses spring form tags to bind the values of the form to a model object.

View Resolver: finds the physical view files from the logical names.

Controller: In the Controller package, create a HomeController class which will have all the methods we need for our CRUD operations.

Here, annotation @Autowired is used to let Spring inject an instance of the CustomerDao implementation into this controller automatically. Each handler method uses this customerDao object to perform necessary CRUD operations.

Handler method is used for retrieving details of a particular customer for viewing, inserting/updating, editing or deleting.

Handler mapping is responsible to find appropriate controllers that handle specific requests.

DAO (Data Access Object) Pattern**:**

Customer Object acting as a Model Object and CustomerDao is the Data access object interface. CustomerDaoImpl is concrete class implementing Data access object interface. DAO provides some specific data operations without exposing to the details of the database.

**Data Access Object Pattern for Customer is shown here. In project it is for Customer, Fleet and Customer Fleet.**

Controller

CustomerDao

+get All(CustomersList)

+update +delete: void

+edit +create

Customer

+get

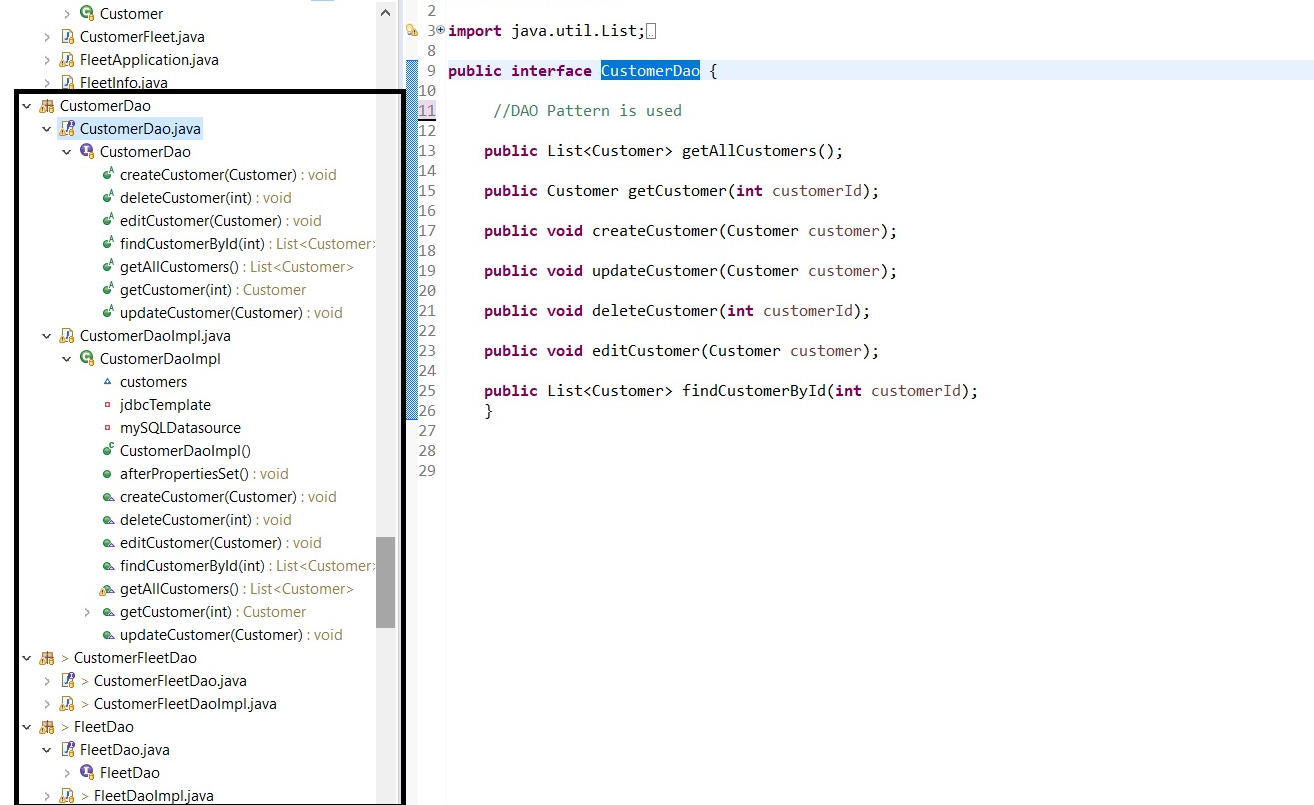
+set

uses

uses

Implements

CustomerDaoImpl



The whole JDBC architecture is actually a [Bridge](https://en.wikipedia.org/wiki/Bridge_pattern), it is an abstract concept which holds other abstractions that could be replaced separately. ResultSet follows the [Iterator](https://en.wikipedia.org/wiki/Iterator_pattern) pattern.

Factory design pattern is creational design pattern. It is useful whenever we want to hide the object creation logic from the outside world. It is hiding object instantiation due to various reason like –

* Object is complex to instantiate manually.
* Object is having logic to instantiate object based on input parameter.
* External API and don’t want to expose logic to outside.

Most of the time Factory design pattern return various object based on it’s input. In simple term, we can say that it is a collection of object.

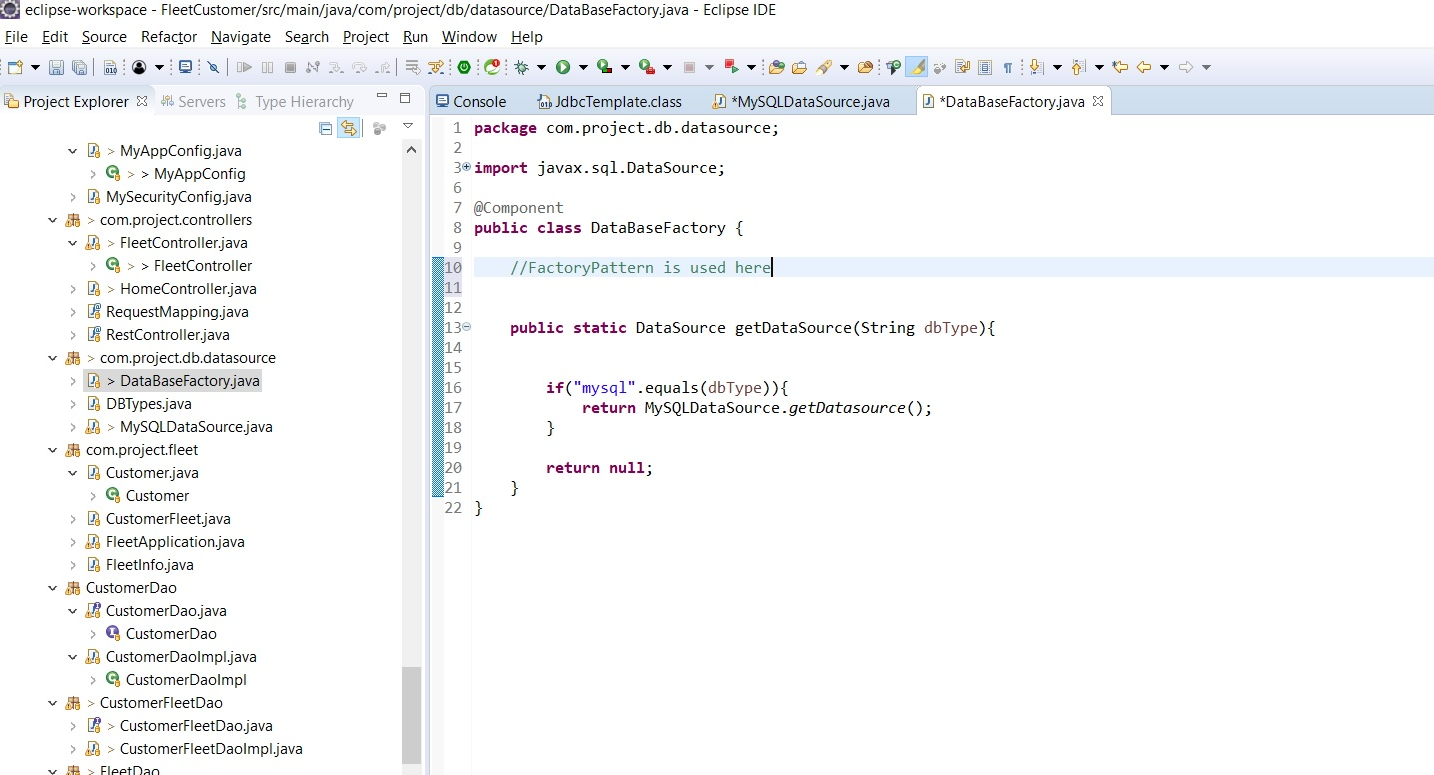
Factory Method Pattern is used to decouple the behavior of executing SQL queries from the type of database used.

Driver Manager +get Connection -----( creates)

My SQL connection Connection +createStatements () -----(creates ) add another DB ’s

Statement (+execute)

My SQL Statements



ModelViewController (MVC) Pattern allows separation of views and business logic. I used Spring boot framework and implemented MVC and other patterns and built my Fleet Management application.

Model contains only the application data and has no logic describing how to present the data to a user.

View represents the model’s data to the user. The view knows how to access the data, but it does not know what this data means or how this data is used by user.

Controller exists between the view and the model. It listens to events triggered by the view (or any other external source like MySQL) and executes the appropriate reactions to the events.

Model and views are connected through notification mechanism and hence the result of this action is then automatically reflected in the view.

ModelViewController Pattern

Model

Customer, Fleet

Updates Manipulates

Controller

Customer- CRUD operations

View

CustomerList, FleetList, Add New Customer, Edit

Fleet Management

Uses

Sees

Fleet Management application is build using Spring Framework and the MYSQL Server database. Updated to postgres. Client is DBeaver. DB is stored in RDS Instance using postgres.

Java Persistence API (JPA).

Spring is a versatile, quick and transparent framework that allows building a simple MVC application. This integrates easily with a database.

Starter Web is used for developing Spring Boot Web applications or RESTful services. Starter Test provides unit testing and integration test capabilities with Spring Test, Mockito, and JUnit

The application follows a standard MVC architecture. It will have a controller (Customer Controller class), views (based on Thymeleaf templates), and a model (a Java map object) and retrieve data from database while the application is running. Once a new project has been created, we will need to edit the Maven configuration file, “pom.xml”, and add the following dependencies:

@repository allows for easy create, read, update, and delete (CRUD) operations.

@Controller maps the request URLs to view templates and perform all necessary processing.

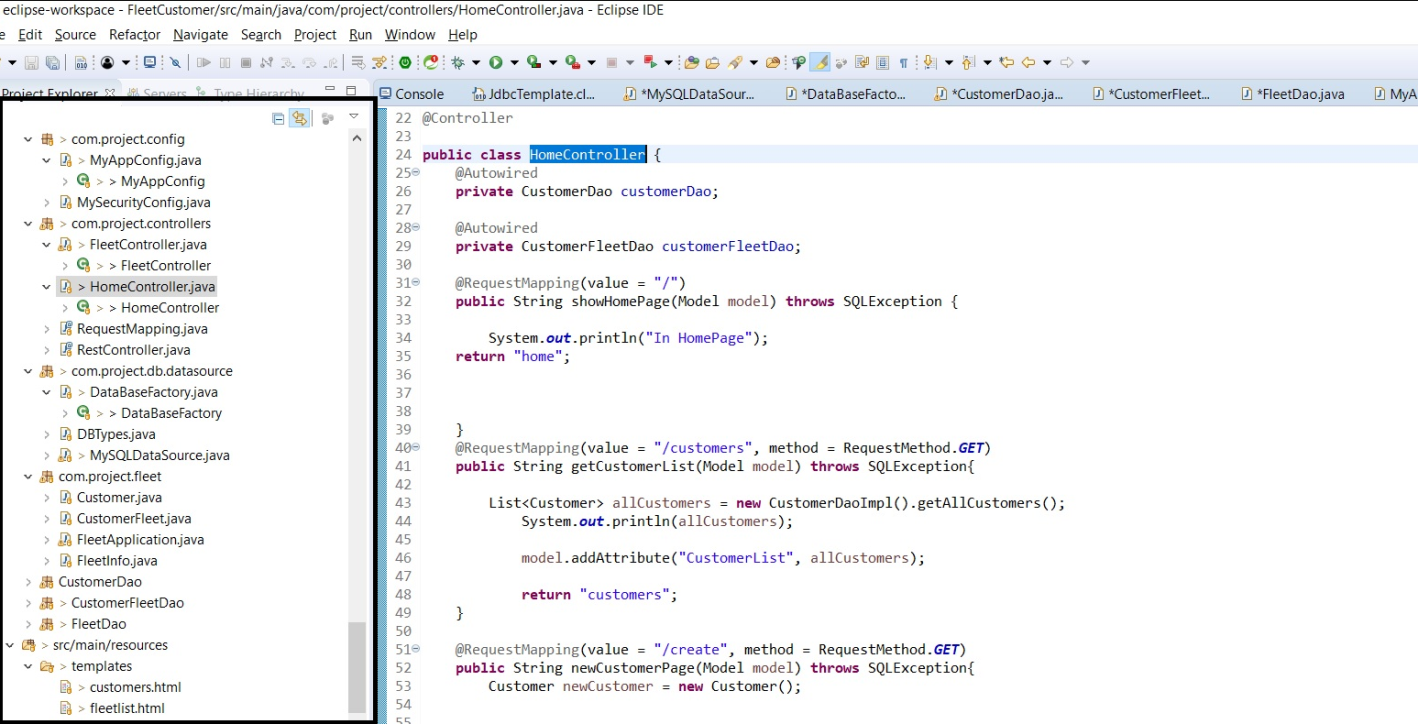
@RequestMapping annotation is used to mapping of URLs to methods. Every method of the controller is mapped to URLs

The model parameter of these methods allows the data to be passed to the view.

Each controller method either returns the name of the Thymeleaf template to be used as view, or a URL in a specific pattern (“redirect :/”) to redirect to.

Within the Controller, @Autowired annotation automatically assigns a valid instance of the defined repository in the corresponding field. This allows the access to relevant data from within the controller without having to deal with a lot of boilerplate code.

Views – Templates can be defined for the views to be generated. Thyme leaf is a simple templating engine. The model we used in controller methods is available directly within the templates. Thymeleaf contains some special elements and attributes that controls generation of HTML.

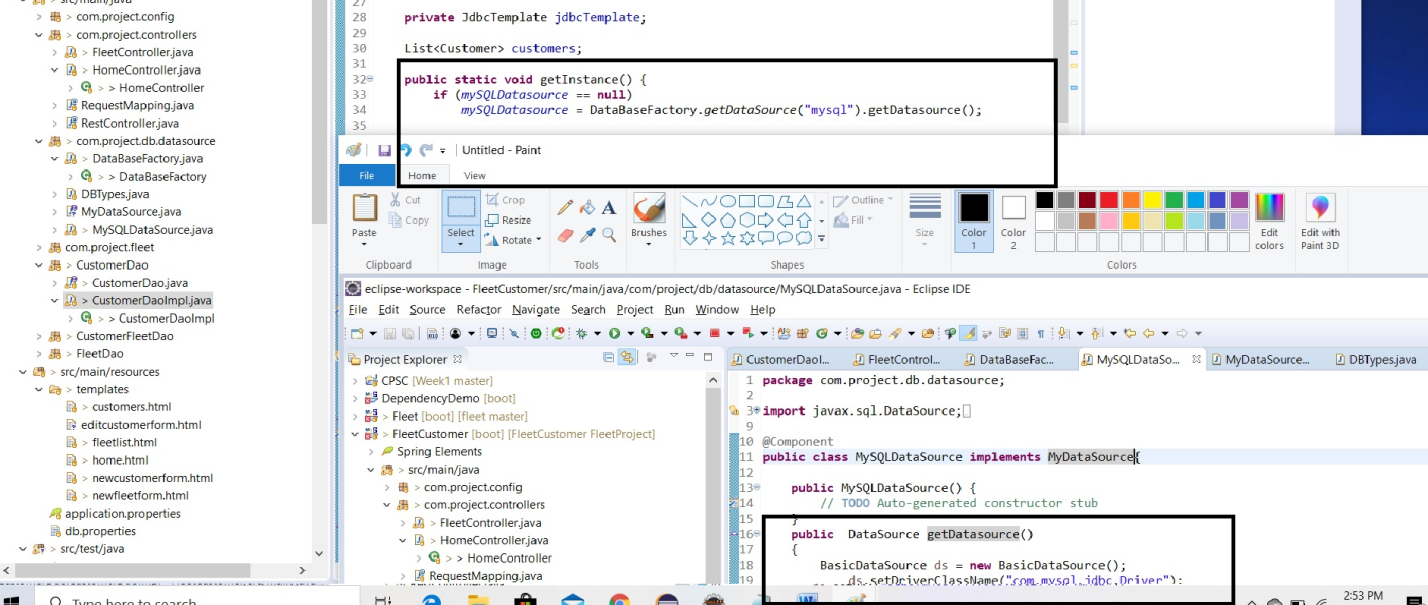


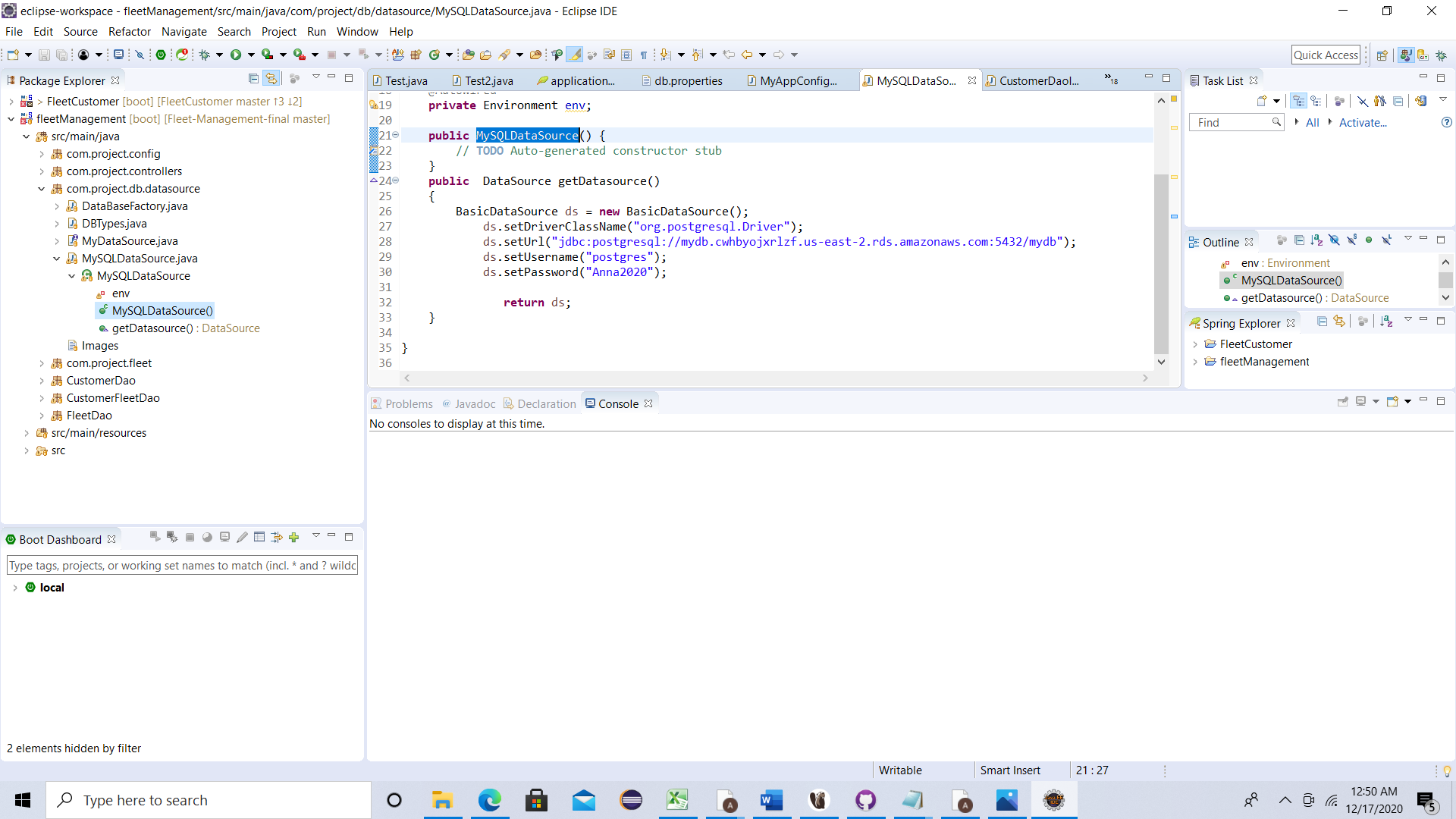
Singleton Pattern is used for calling the data source once and when the ds (data source) is not null, it does not need to call every time.

uses Singleton New

CustomerDaoImpl implement s**tatic** DataSource

CustomerDao getInstance()

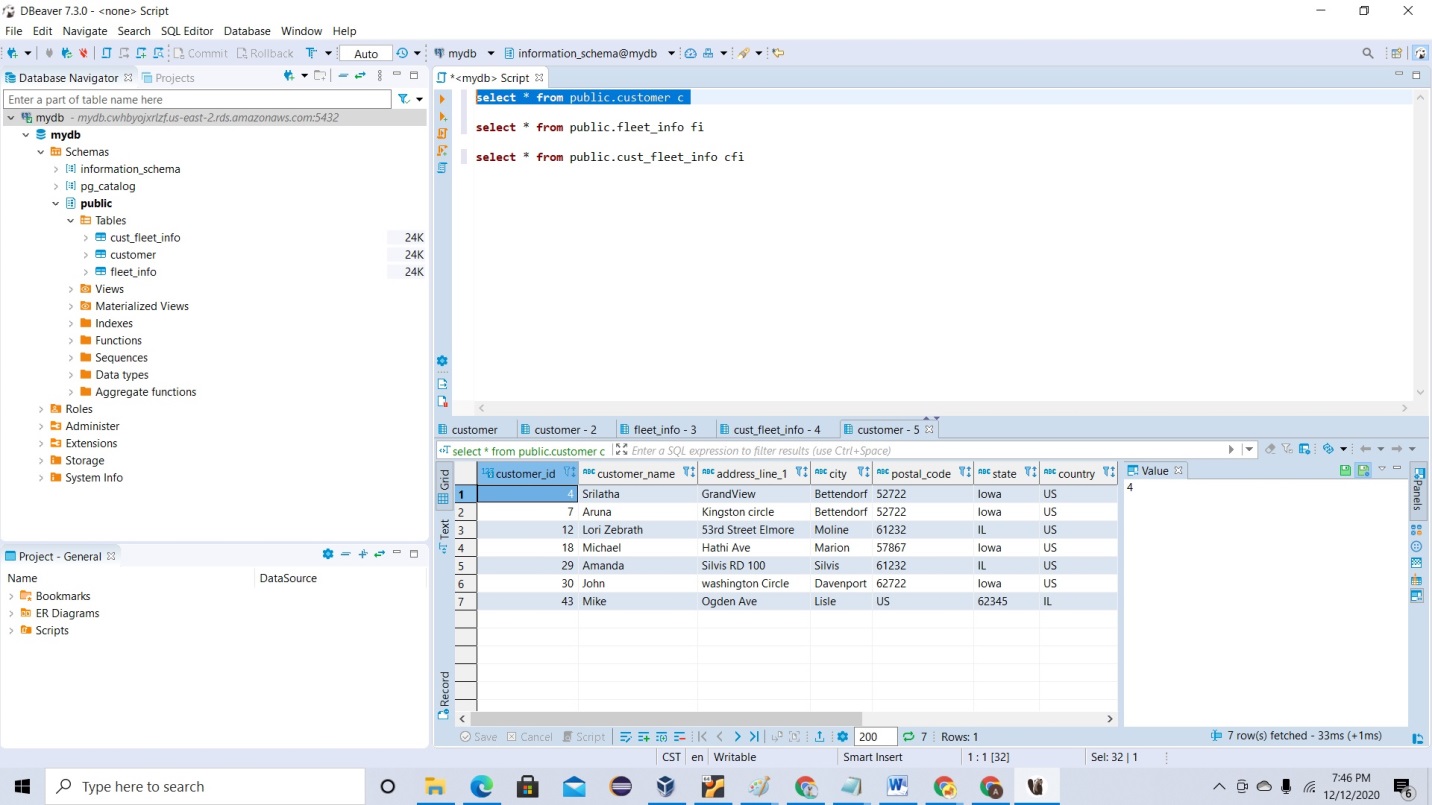




Note:In this Application Admin can Join the Fleets for the customer only in My SQL Table updated to DBEAVER client and He should be able to search the list of fleet for the customer by searching by Customer Id in the web application. Because of the time constraint, I am able to provide a link to view all the Fleets list and create a new fleet list.

From Customer List page, when **user enters 4 and click on Search**, all the fleets belonging to that customer shows up.

Fleet web Application for Fleet list- **Longitude and Latitude are not correct** as I did not implement the google maps for location.



# JUnit Testing:

Testing the application at class level or method level helps to understand if each of the building blocks are functioning as expected so that there would not be any show stopping issues when each of the these individual elements collaborate to perform the high level functionalities of the system.

In this project, the test is performed for data access methods in home controller (Validating CRUD) and factory pattern.

I completed Junit Testing with the help of my friend, and I refactored the code for singleton Pattern as it failed. While validating the CRUD methods.

# Integration Testing:

After integrating the data with MYSQL server and the Java Code using DAO and My SQL as Data source, the application is run as Springboot. Here the embedded server is Tomcat.

The application is developed only for Positive flow. No logging error code is developed

|  |  |  |
| --- | --- | --- |
| Test Cases | Expected Result | Actual Result. |
| Login with invalid user ID or PW. [Url:localhost](file:///C:\Users\T143a\Desktop\CPSC-OOD\Final%20Project-%20Notes\localhost):8080 | Bad Credential message shows up | Working as expected |
| Login with Valid user ID and PW  <Url:localhost>:8080 | User is directed to Fleet Management Login page | Login page shows up as expected. |
| From the Home page, Verify Customer List link and Fleet List link | Both the link s should be available. Customer and Fleet highlighted | Customer and Fleet links highlighted as expected |
| From the Home Page, Click on the Customers link | Verify Search Text box along with New Customer and New Fleet List hyperlinked and below all the list of Customers shows up along with Edit and Delete links for each Customer | Working as expected |
| From the Customer page, Enter a Customer ID in Search and Click on Search | Verify all the Fleet details of the Customer shows up. | Fleet details of the Customer is showing as expected |
| Click on New Fleet List hyperlink | New Fleet List page shows up to enter the Fleet details | Fleet Details are entered |
| Click on New Fleet List Hyperlink, enter the details of the Fleet and click on Save | New Fleet List page shows up to enter the Fleet details  Enter an existing Fleet ID | Whitelabel error page shows up for duplicate entry |
| Click on Go back arrow and change the Fleet ID and click on save | Saved Fleet Details shows up on Fleet List page along with Customers and New Fleet List hyperlinks | Working as expected |
| Click on Customers from Fleet link page | Page is redirected to home page  Customer List and Fleet List hyperlinks shows up | Working as expected |
| Click on Customer List hyperlink | Verify Search Text box along with New Customer and New Fleet List hyperlinked and below all the list of Customers shows up along with Edit and Delete links for each Customer | Working as expected |
| Click on New Customer List page from Customers page | Page is directed to /create to create a new customer with his details and save button shows up | Working as expected |
| Enter the New Customer details and click on Save | Enter all details of the customer and save the details.  Enter Existing customer ID | White label error page shows up for duplicate entry |
| Click on Go back arrow and change the Customer ID and click on save | Page is redirected to home page.  Customer List and Fleet List hyperlinks shows up | Working as expected |
| Click on Customer List to verify the details of the newly created customer | Saved Customer Details shows up on Customer List page along Search Fleet List search box, and with New Customers and Fleet List hyperlinks | Working as expected |
| Click on the Edit Button from Action to update the details of the customer | Edit Customer List shows up with Edit Save button |  |
| Click on the Delete Button from the Action Column to delete a customer | Page is Redirected to home page. Customer List and Fleet List hyperlinks shows up | Working as expected |
| Click on Customer List to verify whether deleted customer info is removed from the customer list | Deleted Customer is removed from the Customer list | Working as expected |

**At this point, this application works only for Positive flow.**

# Challenges during this course of learning and being successful:

**Lesson Learnt:** While doing the coding, lot of errors occurred. With the help of google search, I am able to debug and one of my friend guided me in the Request mapping and Junit testing. I think this is one of my best experiences learning from scratch doing a project with database connection (MY SQL Server) and Installing Spring boot application and implementing the design patterns. Now, I am able to understand why companies are leaning towards Junit testing and Test Driven Development before the code is deployed to production. I now have at least basic understanding on how to debug the code and start doing the root cause analysis more frequently. I spent some couple of hours for simple typo mistakes with the database labeling and the html pages.

# Some of the errors I got during my coding.

When I missed @controller, build program failed.

Missing @ bean configuration

View templates and .jsp pages not addressed correctly.

Issues with Data base column names not matching within the Java classes. Primary and Foreign key

@Request Mapping

# Fleet Management Application Screenshots:

