Spring Boot REST Services using JPA Part 5

Description: Spring Boot is an open source, micro service-based Java web framework. The Spring Boot framework creates a fully production-ready environment that is completely configurable using its prebuilt code within its own codebase.

Project: This project expands on the program built using documents "Spring Boot REST Services using JPA" Part 1 thru 4 which is based on Building REST services with Spring. The Part 1 program created a simple payroll service that manages the employees of a company. Part 2 updated the controller class adding links to the response to make it more RESTful. Part 3 simplified adding links to the response messages. Part 4 updated the employee entity model and add customer order processing. This document Part 5 works on refactoring using better software engineering practices.

Technology: This project uses the following technology:

Integrated Development Environment (IDE):

Spring Tool Suite 4 (Version: 4.15.0.RELEASE)

Java Development Kit (JDK):

Oracle's JDK 8 (1.8)

Other Technology:

Postman – a web and desktop application used for API testing.

Table of Contents

Glossary of Terminology

For a list of key terms and definitions used throughout this and various Spring Boot demo documents see the document titled “Appendix 01 Glossary”.

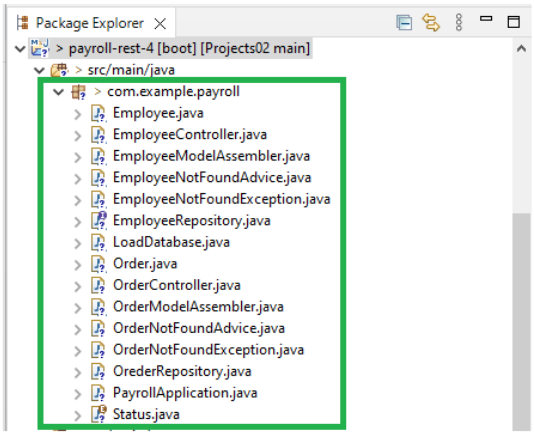
Copy Existing Project: payroll-rest-3

This project builds on the project created using “Spring Boot 300 REST Services Part 4” which was based on the website Building REST services with Spring. The payroll-rest-3 project must exist to following the details in this document. The project built in Part 4 is used as the starting point for this project. The overall project is applying software engineering practices refactoring the project.

Right click on the project “payroll-rest-3” ➔ select “Copy”
Right click in open area in “Package Explorer” ➔ select “Paste”
Enter “Project name:” **payroll-rest-4** ➔ Click “Copy”

Project Discussion

This project is solely for the purpose of refactoring the finished project “payroll-rest-3” based on the website “Building REST services with Spring”. After coping the “payroll-rest-3” the initially structure of project “payroll-rest-4” is as follows where all Java classes are under one package:



In the Model View Controller (MVC) software architectural pattern commonly used for structuring software products. It is a division of responsibility and division of code in software packages. The view component is typically a front-end that sends requests to a back-end where the controller component receives the request then interacts with the model component to control processing of the data in the model. The controller is also responsible for rendering the data back to the view component for presentation back to the requestor.

What does this mean for refactoring project “payroll-rest-4”?

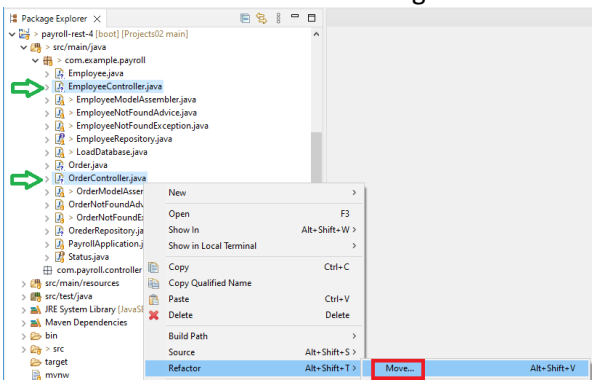
- View – the project relies on Postman to send request and display results. There is no change to the view component.

- Controller – the two controller components are refactored into their own package.
- Model – the domain entity / model classes are refactored into their own package.
- Other refactoring is also performed separating exception, assembler, repository, and main classes into separate packages as well.

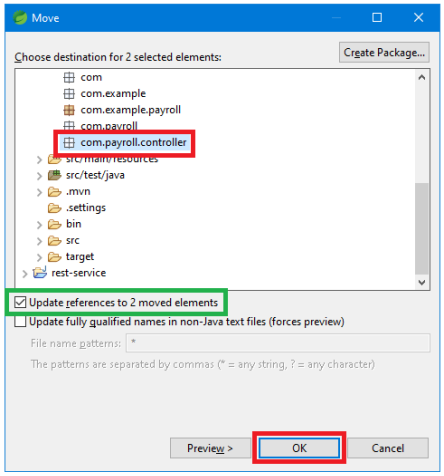
Refactoring is usually more extensive than creating packages for separation of duties. Characteristics of refactoring found in various definition has refactoring as a process of restructuring code. This restructuring of code happens while not changing its original functionality. The end results of refactoring is to improve code maintainability by making repeating minimal changes without altering the code's existing outward behavior.

Refactor Controller Classes

This refactoring creates a “com.payroll.controller” package and moving the controller classes.
Select both controller classes ➔ right click ➔ “Refactor” ➔ select “Move”

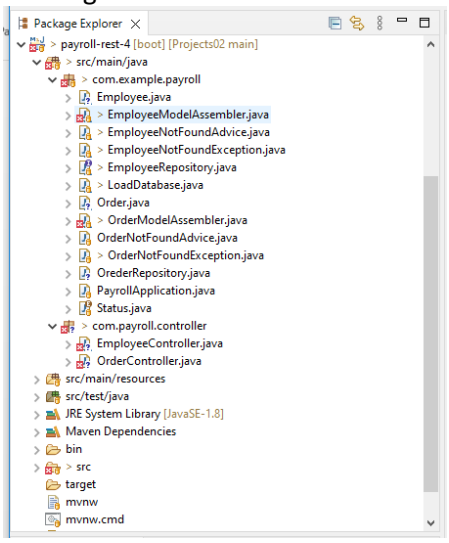


Select the package just created ➔ click “OK”



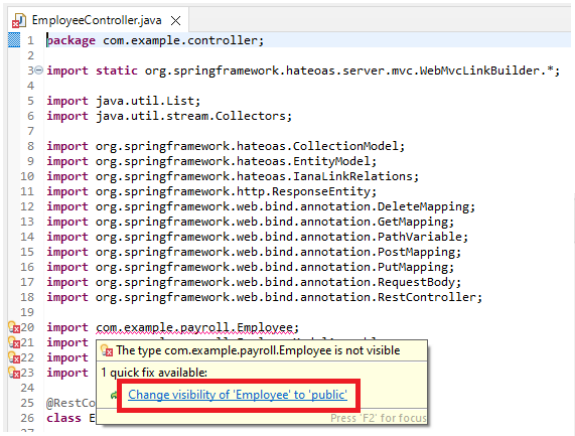
The references to the moved classes will be automatically updated by the refactoring Wizard.

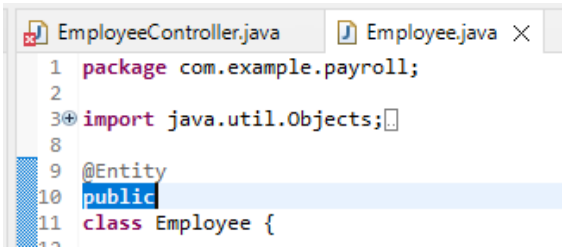
Although references to the moved classes were updated, the move produced some unexpected errors.



As it turns out most of the classes and interfaces created using the "Building REST services with Spring" code were created without access modifiers. When a class is created without an access modifier (public, protected, or private) then by default the member of a class is public, accessible only within its own package. The class cannot be accessed outside of its package. Not providing access modifiers when defining a class is another poor software engineering practice.

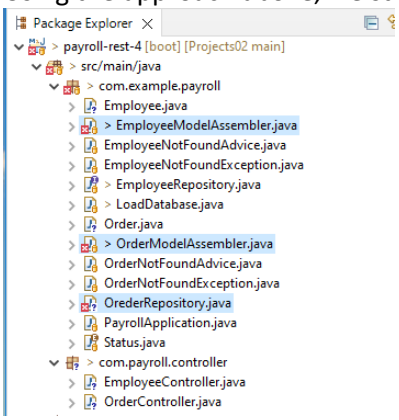
We can use the IDE Wizard to update the access modifier for a given error.



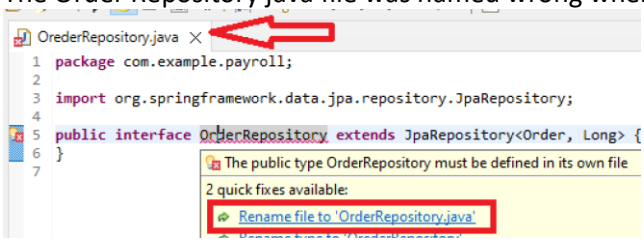


You can also open each class and add access modifier of "public". You will also want to make any class constructor public that does not have an access modifier. Taking this approach will limit issues when creating other packages and moving classes. We also need to update the two controller classes' access modifiers.

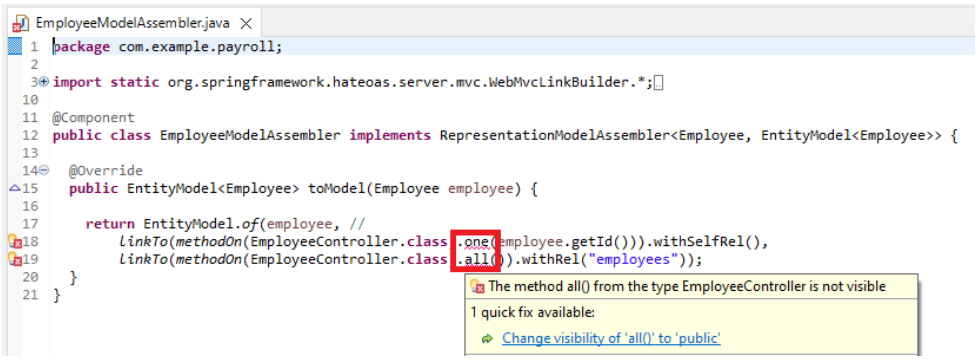
Using the approach above, we still have issues in three classes.



The Order Repository java file was named wrong when created. Use the IDE Wizard to rename the file.



The two methods in the EmployeeController class require public access. Add manually or with the IDE Wizard.



The four methods in the OrderController class require public access. Add manually or with the IDE Wizard.

The project is now free of errors and refactoring can continue.

Continue Refactoring Packages and Classes

Refactoring for Model Classes

Create “com.payroll.model” package and move the four model related classes (Employee, EmployeeModelAssembler, OrderModelAssembler and Order).
This is straight forward where no errors are produced.

Refactoring for Repository Classes

Create “com.payroll.repository” package and move the two repository classes (EmployeeRepository and OrderRepository). Also move the LoadDatabase classes to this package.
This is straight forward where no errors are produced.

Refactoring for Exception Classes

Create “com.payroll.exception” package and move the four exception related classes (EmployeeNotFoundAdvice, EmployeeNotFoundException, OrderNotFoundAdvice, and OrderNotFoundException).
This is straight forward where no errors are produced.

Refactoring for Application Classes

The remaining classes (PayrollApplication and Status) can be package as application related. Here we can rename the remaining package from the project copied.

Refactor renaming package “com.example.payroll” to “com.payroll”.

It is important to note that the PayrollApplication class package is the root package and not a subpackage under “com.payroll”. This allows the annotation @SpringBootApplication to work in its capacity as @ComponentScan, meaning it will scan the package that this configuration class is in, and also any sub-packages (i.e. com.payroll.*). This way when the application is started, it will find the LoadDatabase class, and controller classes and work as expected.

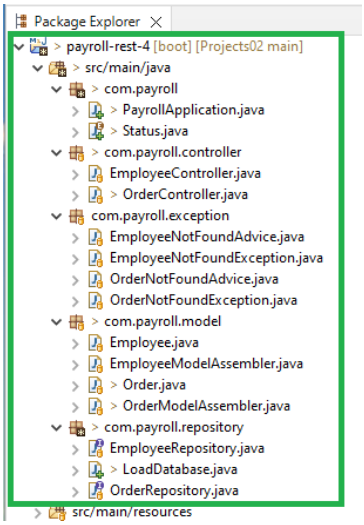
As stated in spring boot documentation:

> The `@SpringBootApplication` annotation is equivalent to using `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` with their default attributes.
>
> Where:
>
> - `@EnableAutoConfiguration`: enable Spring Boot’s auto-configuration mechanism
> - `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)
> - `@Configuration`: allow to register extra beans in the context or import additional configuration classes
>
> -- 18. Using the @SpringBootApplication Annotation

This is straight forward where no errors are produced.

Final Refactoring Structure

After creating new packages, moving classes, and resolving errors the final project structure looks as follows:



Testing Refactored Project Discussion

The Postman request created in the previous project documents are used to perform regression testing on the all endpoints that exist in the program. Regression testing will shake out any issue created with refactoring the project. Regression testing is an important part of software engineering, in this case we can use test cases created in previous project to test the refactoring in this project. No new test cases are required.

The test cases and sequence from previous projects are as follows:

GET: all employee records
GET: find by employee record id
GET: find by employee record id not found
POST: create an employee
GET: all employee records
PUT: update an existing employee record
GET: all employee records
DELETE: delete an employee record
GET: all employee records
POST: create employee by first and last name
GET: all order records
DELETE: cancel order by id
PUT: complete order by id
POST: create order

Test Refactoring Project: Regression Testing

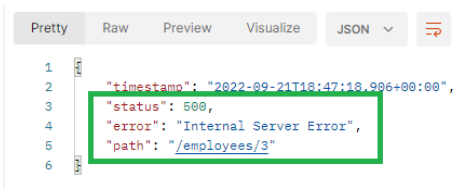
Start the application inside the IDE as Spring Boot App.
Run the sequence of Postman request and validate the response.
Validation of the responses can be compared to the results in the previous "Spring Boot 300 REST Services Part 4" document.

1. GET: all employee records
2. GET: find by employee record id
3. GET: find by employee record id not found
4. POST: create an employee
5. GET: all employee records
6. PUT: update an existing employee record
7. GET: all employee records
8. DELETE: delete an employee record
9. GET: all employee records
10. POST: create employee by first and last name
11. GET: all order records
12. DELETE: cancel order by id
13. PUT: complete order by id
14. POST: create order

Error Deleting a Created Record

With the regression testing of the application using the test cases sequence identified above, an error occurred with the test "DELETE: delete an employee record". This error would have occurred with the previous version (before refactoring) of the application. It was not discovered in the testing of the last application.

After creating an employee in test #4, test #8 attempts to delete the employee by its id. The response from the delete request give a status of 500, "Internal Server Error". This is due to employee record number 3 does not exist.



Looking at the response from test #4 we see that the employee has record id of 5. This is unexpected since this should be the third employee record created.

Why did the delete error happen?

If we examine the console log when the application was started at the section where preloading of the database results is shown we see employee with id of 1 then 2 is created. The orders are created starting at id 3. Again this is not what was expected. Since the employee table and the order table are separate database tables they should have generated unique record ids starting with 1.

```
Preloaded Employee{id=1, firstName='Bilbo', lastName='Baggins', role='burglar'}
Preloaded Employee{id=2, firstName='Frodo', lastName='Baggins', role='thief'}
Preloaded Order{id=3, description='MacBook Pro', status=COMPLETED}
Preloaded Order{id=4, description='iPhone', status=IN_PROGRESS}
```

Now look at the entity model code. Both classes use the following line to identify and generate a value for the record id.

```
private @Id @GeneratedValue Long id;
```

The @GenerateValue as used is using a default generation scheme. The default is the same if written as:

```
@GeneratedValue(strategy = GenerationType.AUTO)
```

As defined GenerationType.AUTO allows the database framework to decide which generator type to be used. By observation we see for the memory database GenerationType.AUTO simply increments one value for any table for each row created. Meaning for the initial four records created, two for employee and two for orders, the GenerationType.AUTO used 1, 2, 3, & 4 regardless of which entity table was creating a record.

Correcting the delete error

A better and more logical approach is to generate the primary key for each entity table starting at 1 respectively. So on our programs record load we want employee record 1 & 2 created and order record 1 & 2 created. This scheme would make testing better at backward compatibility and easier to understand and maintain. The way to get the desired effect is to replace the @GenerationType with @GeneratedValue(strategy = GenerationType.IDENTITY). GenerationType.IDENTITY in this case starts the record count in each entity table at one, incrementing by one for each record created.

To make the code more readable we can define as follows:

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

The Order class:

```
13 @Entity
14 @Table(name = "CUSTOMER_ORDER")
15 public class Order {
16
17     private @Id
18     @GeneratedValue(strategy = GenerationType.IDENTITY)
19     Long id;
20
```

Also correct the import error by adding:

```
import javax.persistence.GenerationType;
```

The Employee class:

```
11 @Entity
12 @Table(name = "EMPLOYEE")
13 public class Employee {
14
15     private @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     Long id;
18
```

For consistency and control add the @Table line giving the table its name.

```
@Table(name = "EMPLOYEE")
```

Test Refactoring Project Part 2: Regression Testing

Now that the delete error is corrected, run the set of test cases as was planned.

Start the application inside the IDE as Spring Boot App.
Examine console output related to database load.

```
Preloaded Employee{id=1, firstName='Bilbo', lastName='Baggins', role='burglar'}
Preloaded Employee{id=2, firstName='Frodo', lastName='Baggins', role='thief'}
Preloaded Order{id=1, description='MacBook Pro', status=COMPLETED}
Preloaded Order{id=2, description='iPhone', status=IN_PROGRESS}
```

Now we see the sequencing of record id starting at one for each entity table.

Run the sequence of Postman request and validate the response.
Validation of the responses can be compared to the results in the previous “Spring Boot 300 REST Services Part 4” document.

1. GET: all employee records
2. GET: find by employee record id
3. GET: find by employee record id not found
4. POST: create an employee
   Now we see the new employee created has the expected record id of 3.

5. GET: all employee records
6. PUT: update an existing employee record
7. GET: all employee records
8. DELETE: delete an employee record
   The delete now works as expected deleting record 3.
9. GET: all employee records
10. POST: create employee by first and last name
11. GET: all order records
12. DELETE: cancel order by id
    Now that the record id sequence is corrected, this request needs to change to:
    `http://localhost:8080/orders/2/cancel`
13. PUT: complete order by id
    The same is needed for this request:
    `http://localhost:8080/orders/2/cancel`
14. GET: find order by record id
    The same is needed for this request:
    `http://localhost:8080/orders/2/cancel`
15. GET: find order by record id Not Found
16. POST: create order
17. GET: all order records

Project Conclusion

The set of documents created to step through various stages of the Spring Boot demo on the "Building REST services with Spring". The final document refactored the project making it resemble the MVC architecture and making easier to maintain. The final project can be used as a model for building other simple Spring Boot application using a memory database. To make the project realistic, the memory database should be converted to a physical database with persistent data. The memory database can still exists and be used in JUnit testing.