BUILDING A REST API WITH SPRING BOOT

UNTITLED

- Overview
  - Source Code Download
  - For further interesting Content, you can follow me at
- Getting Started
  - Exploring Maven Dependencies
  - Define MongoDB properties
  - Creating Domain Model
  - Creating Repository
    - Creating Custom Queries using @Query
- Creating REST API
  - REST API URL Naming Conventions
    - Use Nouns instead of verbs
    - Use Lowercase Letters
    - Do not use file extensions
  - Add Expense
    - Response Status for POST Request
    - Testing POST Request
  - Get Expense
    - Response Status for GET Request
    - Testing GET Request
  - Get All Expenses
    - Testing GET Request
  - Update Expense
    - Response Status for PUT Requests
    - Testing PUT Request
  - Delete Expense
    - Response Status for DELETE Requests
    - Testing DELETE Request
- Error Handling in REST API
  - Using @ExceptionHandler Annotation
  - Using @ControllerAdvice
    - Using ResponseStatusException class
- Testing REST API

Documenting the REST API
Why should we document our REST APIs?
What is Swagger?
Adding Springfox Dependencies to project
Configure Swagger and Springfox
Conclusion

Overview

In this ebook, you are going to learn how to build a REST API with Spring Boot, we are going to build an Expense Tracker Application where users can add and track their expenses.

Here are the technologies we are going to use to Build the REST API:

- Spring Boot
- MongoDB

To be able to follow along this ebook, you need to have a basic understanding of Spring Boot.

Source Code Download

You can download the whole source code at Github

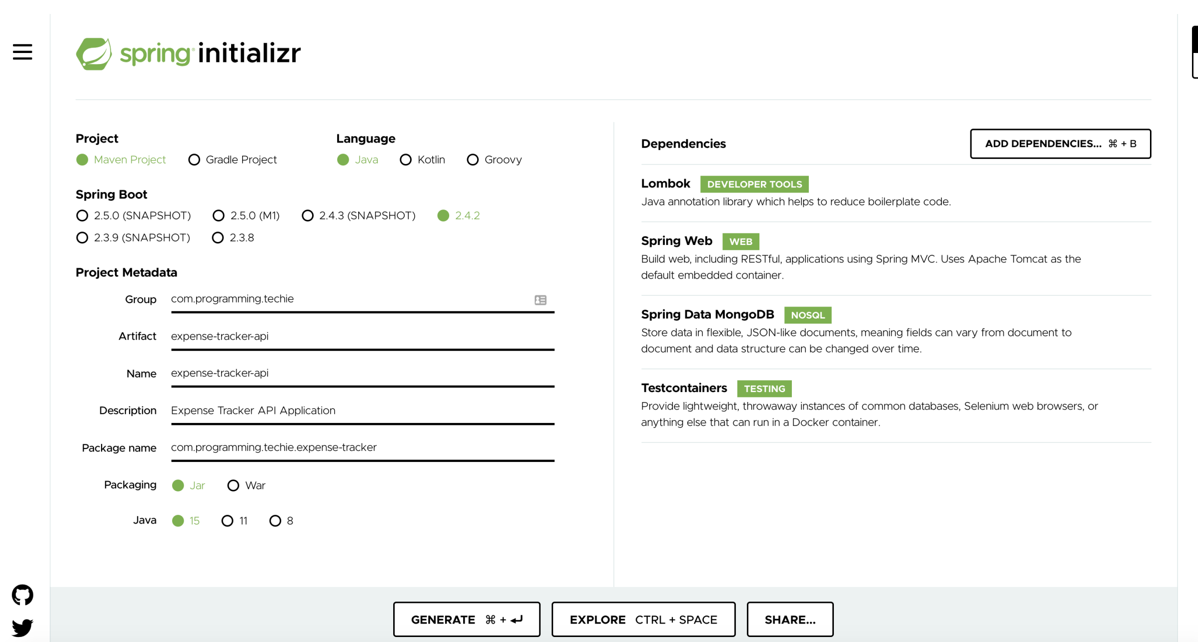
For further interesting Content, you can follow me at

- Youtube
- Twitter

Getting Started

To get started, open [Spring Initializr]() website, and select the following options.

As of writing this ebook, the latest version of Spring Boot is 2.4.2.



In the Spring Initializr home page, I selected the following options:

- **Project** : Maven Project
- **Language**: Java
- **Spring Boot**: Version 2.4.2
- **Dependencies**:
- **Spring Web**: To be able to build RESTful API using Spring MVC
- **Spring Data MongoDB**: To interact with MongoDB from the Spring Boot Application

- **Lombok**: Java Annotation Library which helps to reduce boiler plate code.
- **Testcontainers**: Provides lightweight instances of the Mongo Database which we can run inside a Docker Container.

After providing the Project Metadata, you can download the project to your machine, by clicking on the **Generate** button.

Exploring Maven Dependencies

Once you unzip and open the project in your favorite IDE, open the `pom.xml` file to have a look at the Maven dependencies which we are going to use in our project.

```
<?xml version="1.0" encoding="UTF-8"?>
<project                xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.1</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.programming.techie</groupId>
<artifactId>spring-boot-mongodb-tutorial</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>spring-boot-mongodb-tutorial</name>
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>15</java.version>
    <testcontainers.version>1.15.1</testcontainers.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
                            <artifactId>spring-boot-starter-data-
mongodb</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
```

```xml
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>junit-jupiter</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testcontainers</groupId>
            <artifactId>mongodb</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.testcontainers</groupId>
                <artifactId>testcontainers-bom</artifactId>
                <version>${testcontainers.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
```

```
                </excludes>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

If you want to learn more about Maven, have a look at the comprehensive Blog Post I wrote Maven Complete Tutorial

Define MongoDB properties

Now it's time to configure the MongoDB properties inside the `application.properties` file

You can define the MongoDB properties by either using the MongoURI or by defining the host, username, password and database details.

Approach 1:

```
spring.data.mongodb.uri=mongodb://localhost:27017/expense-tracker
```

Approach 2:

```
spring.data.mongodb.host=localhost

spring.data.mongodb.username=<your-username>

spring.data.mongodb.password=<your-password>

spring.data.mongodb.database=expense-tracker
```

Creating Domain Model

The model class we are going to create for the Expense Manager application is the `Expense.java` class.

```
package com.programming.techie.expensetracker.model;
```

```
import lombok.*;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.index.Indexed;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.mongodb.core.mapping.Field;

import java.math.BigDecimal;

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Document("expense")
public class Expense {
    @Id
    private String id;
    @Field("name")
    @Indexed(unique = true)
    private String expenseName;
    @Field("category")
    private ExpenseCategory expenseCategory;
    @Field("amount")
    private BigDecimal expenseAmount;
}
```

```
ExpenseCategory.java

package com.programming.techie.expensetracker.model;

public enum ExpenseCategory {
    ENTERTAINMENT, GROCERIES, RESTAURANT, UTILITIES, MISC
}
```

This is just a normal POJO class, which contains some interesting annotations, if you are not already aware of Lombok, it is an annotation library which helps us reduce the boiler plate code.

In the above class,

- You can see that by adding the `@Getter`, `@Setter`, `@AllArgsConstructor` and `@NoArgsConstructor` we can generate the required Getter, Setter Methods and the Constructors at compile time.

- To define a Model Class as a MongoDB Document, we are going to use the `@Document("expense")` where expense is the name of the Document.
- `@Id` represents a unique identifier for our Document.
- We can represent different fields inside the Document using the `@Field` annotation.
- By Default, Spring Data creates the field inside the document using the fieldName of the model class (Eg: expenseName), but we can override this by providing the required value to the annotation eg: `@Field("name")`
- To be able to easily retrieve the documents, we can also create an index using the `@Indexed` annotaion.
- We can also specify the unique=true property to make sure that this field is unique.

Creating Repository

Now it's time to creating the Repository interface, which is responsible to interact with MongoDB. **Spring Data MongoDB** provides an interface called `MongoRepository` which provides an API to perform read and write operations to MongoDB.

```
package com.programming.techie.expensetracker.repository;

import com.programming.techie.expensetracker.model.Expense;
import
org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import java.util.Optional;

public interface ExpenseRepository extends
MongoRepository<Expense, String> {
    @Query("{'name': ?0}")
    Optional<Expense> findByName(String name);
}
```

Creating Custom Queries using @Query

We can also perform custom queries using the `@Query` annotation and by passing in the required query we need to run to this annotation.

In the below example:

```
@Query("{'name': ?0}")
Optional<Expense> findByName(String name);
```

Spring Data will inject the value of the name field into the query, in the place of the **?0** placeholder.

So we created the **domain** and **repository** layers in our application, now it's time to go ahead and create the **API** layer.

Creating REST API

Now we are at the main part of our ebook, we are going to create the RestController which is going to receive the HTTP Requests and is going to delegate the requests to the Service Layer.

We are going to create a REST API, which exposes the following functionality:

- Add Expense
- Update Expense
- Delete Expense
- Get All Expenses
- Get Expense by Name

First I am going to create two classes `ExpenseController.java` which acts as the RestController to accept the incoming request and `ExpenseService.java` which contains the business logic of the **ExpenseManager** application.

```
ExpenseController.java

package com.programming.techie.expensetracker.web;

import
com.programming.techie.expensetracker.service.ExpenseService;
import lombok.RequiredArgsConstructor;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/expense")
@RequiredArgsConstructor
public class ExpenseController {

    private final ExpenseService expenseService;
```

```
}
```

The `@RestController` is the central part of the API tier, it will accept the RESTful HTTP Requests and delegate the request to the Service Layer.

The `@RestController` is basically a combination of the `@Controller` and `@ResponseBody` annotations.

The usual convention to write the URL of the REST API is the start with the prefix - **/api**, this is not mandatory, but a standard convention followed across the industry.

REST API URL Naming Conventions

There are some pre-defined conventions which we have to follow to define the URL of the REST API.

Use Nouns instead of verbs

API's are always designed around Resources, so in our example it's going to be the `Expense` class.

Any kind of operations we are going to perform against those resources should be defined through the HTTP Verbs like (**GET**, **POST**, **PUT**, **DELETE**)

For example:

If you want to define the URL for the Add Expense Endpoint, instead of naming the URL as `/api/expense/add` we are going to name it as `/api/expense/` and the action is taken care by the corresponding HTTP Verb we will use, in this case it's going to be **POST**.

Use Lowercase Letters

Lowercase Letters should be preferred for the URI's as much as possible. According to RFC-3986 URIs are case sensitive except for the scheme and

Host componenets.

Do not use file extensions

If you want to access a file, never use the extenstion as part of the URI, this is because URI should be independent of the implementation.

For example instead of defining the URL for a pdf file like below:

```
/api/expense/report.pdf
```

Define it like below:

```
/api/expense/report
```

In the above case, we will communicate our intent that we need a PDF file through the `Content-Type` header, by using the **Media Type** parameter.

```
ExpenseService.java

package com.programming.techie.expensetracker.service;

import
com.programming.techie.expensetracker.repository.ExpenseRepositor
y;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@RequiredArgsConstructor
@Transactional
public class ExpenseService {

    private final ExpenseRepository expenseRepository;
}
```

Add Expense

To add an expense we are going to receive the Request from the client and delegate it to the Service Layer, the important thing to remember is to

decouple your Domain Layer (`Expense.java`) with your API/Presentation Layer.

For this reason, we are going to use a Data Transfer Object (DTO) to receive the input from the client and then map this object to our Domain Model.

For this reason we are going to create a `ExpenseDto.java` class

```
package com.programming.techie.expensetracker.dto;

import
com.programming.techie.expensetracker.model.ExpenseCategory;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.math.BigDecimal;

@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class ExpenseDto {
    private String id;
 private String expenseName;
 private ExpenseCategory expenseCategory;
 private BigDecimal expenseAmount;
}
```

Now let's add logic to add a new Expense first inside `ExpenseService.java`

```
package com.programming.techie.expensetracker.service;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.model.Expense;
import
com.programming.techie.expensetracker.repository.ExpenseRepositor
y;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
@RequiredArgsConstructor
@Transactional
public class ExpenseService {
```

```
    private final ExpenseRepository expenseRepository;

 public String addExpense(ExpenseDto expenseDto) {
        Expense expense = mapFromDto(expenseDto);
 return expenseRepository.insert(expense).getId();
  }

    private Expense mapFromDto(ExpenseDto expense) {
        return Expense.builder()
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
                .expenseAmount(expense.getExpenseAmount())
                .build();
  }
}
```

- We created the `addExpense` method which is going to map the `ExpenseDto` object to an `Expense` object.
- To map the `ExpenseDto` to `Expense` we are going to map it manually using the `mapFromDto()` method.
- Once we have the `Expense` object, we can save it to the database using the `expenseRepository.insert()` method.

Now let's see how to implement `ExpenseController.java`

```
package com.programming.techie.expensetracker.web;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import
com.programming.techie.expensetracker.service.ExpenseService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import
org.springframework.web.servlet.support.ServletUriComponentsBuild
er;

import java.net.URI;

@RestController
```

```
@RequestMapping("/api/expense")
@RequiredArgsConstructor
public class ExpenseController {

    private final ExpenseService expenseService;

   @PostMapping
   public ResponseEntity<Void> addExpense(@RequestBody ExpenseDto
expenseDto) {
        String expenseId = expenseService.addExpense(expenseDto);
   URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(expenseId)
                .toUri();
 return ResponseEntity.created(location)
                .build();
   }

}
```

As I mentioned before, we are going to implement the Add Expense Endpoint with a `@PostMapping`.

We are receiving the `ExpenseDto` as a `@RequestBody` and we are passing on this object to `ExpenseService` which returns the `expenseId` of the created `Expense`.

Response Status for POST Request

You can observe that we are returning a `ResponseEntity` with status as **CREATED**, because according to REST conventions, when you send a POST request, you are creating a Resource, so the appropriate response status should be **CREATED** (201).

Also as part of the POST request, we should return the URL of the resource we just created as part of the Locations Header, for this reason we are dynamically constructing the URL and sending it as part of the response.

Testing POST Request

Let’s start our Spring Boot Application and make the REST API call to add an expense.

Get Expense

Next step is to implement the GET Expense endpoint to read a single expense, we are going to use `@GetMapping` annotation.

```
ExpenseService.java

package com.programming.techie.expensetracker.service;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import
com.programming.techie.expensetracker.exception.ExpenseNotFoundEx
ception;
import com.programming.techie.expensetracker.model.Expense;
import
com.programming.techie.expensetracker.repository.ExpenseRepositor
y;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
```

```
@Service
@RequiredArgsConstructor
@Transactional
public class ExpenseService {

    private final ExpenseRepository expenseRepository;

 public String addExpense(ExpenseDto expenseDto) {
        Expense expense = mapFromDto(expenseDto);
 return expenseRepository.insert(expense).getId();
  }

    public ExpenseDto getExpense(String name) {
        Expense expense = expenseRepository.findByName(name)
                                .orElseThrow(() -> new
ExpenseNotFoundException(String.format("Cannot Find Expense by
Name - %s", name)));
 return mapToDto(expense);
  }

    private ExpenseDto mapToDto(Expense expense) {
        return ExpenseDto.builder()
                .id(expense.getId())
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
                .expenseAmount(expense.getExpenseAmount())
                .build();
  }

    private Expense mapFromDto(ExpenseDto expense) {
        return Expense.builder()
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
                .expenseAmount(expense.getExpenseAmount())
                .build();
  }
}
```

We are receiving the expense name we want to read as an input parameter and we are using the `expenseRepository.findByName()` to read the `Expense`

If we don't find an expense with a given name, we are throwing an `ExpenseNotFoundException`

```
package com.programming.techie.expensetracker.exception;

public class ExpenseNotFoundException extends RuntimeException {
```

```
    public ExpenseNotFoundException(String message) {
        super(message);
  }
}
```

ExpenseController.java

```
package com.programming.techie.expensetracker.web;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import
com.programming.techie.expensetracker.service.ExpenseService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import
org.springframework.web.servlet.support.ServletUriComponentsBuild
er;

import java.net.URI;

@RestController
@RequestMapping("/api/xpense")
@RequiredArgsConstructor
public class ExpenseController {

    private final ExpenseService expenseService;

  @PostMapping
   public ResponseEntity<Void> addExpense(@RequestBody ExpenseDto
expenseDto) {
        String expenseId = expenseService.addExpense(expenseDto);
  URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(expenseId)
                .toUri();
 return ResponseEntity.created(location)
                .build();
  }

    @GetMapping("/{name}")
    @ResponseStatus(HttpStatus.OK)
    public ExpenseDto getExpenseByName(@PathVariable String name)
{
        return expenseService.getExpense(name);
  }
```

```
}
```

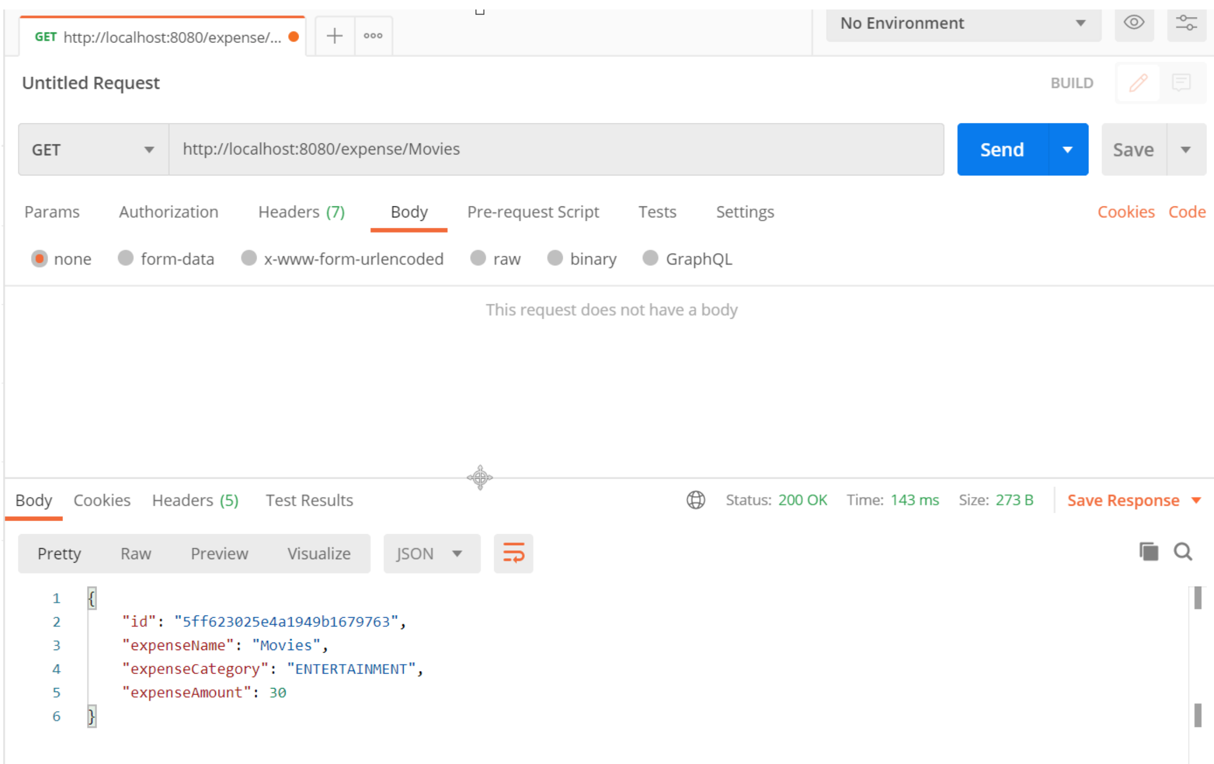
We are receiving the `expenseName` as a URL PathVariable, and we are returning the `ExpenseDto` back to the client.

Response Status for GET Request

The appropriate response the GET Request is **OK** (200)

Testing GET Request

Let's start our Spring Boot Application and make the REST API call to Get One Expense.



Get All Expenses

Let's go ahead and implement the Endpoint logic to GET ALL Expenses.

`ExpenseService.java`

```
package com.programming.techie.expensetracker.service;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import
com.programming.techie.expensetracker.exception.ExpenseNotFoundEx
ception;
import com.programming.techie.expensetracker.model.Expense;
import
com.programming.techie.expensetracker.repository.ExpenseRepositor
y;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
@Transactional
public class ExpenseService {

    private final ExpenseRepository expenseRepository;

 public String addExpense(ExpenseDto expenseDto) {
        Expense expense = mapFromDto(expenseDto);
 return expenseRepository.insert(expense).getId();
  }

    public ExpenseDto getExpense(String name) {
        Expense expense = expenseRepository.findByName(name)
                                        .orElseThrow(()   ->   new
ExpenseNotFoundException(String.format("Cannot  Find  Expense  by
Name - %s", name)));
 return mapToDto(expense);
  }

    public List<ExpenseDto> getAllExpenses() {
        return expenseRepository.findAll()
                .stream()

.map(this::mapToDto).collect(Collectors.toList());
  }

    private ExpenseDto mapToDto(Expense expense) {
        return ExpenseDto.builder()
                .id(expense.getId())
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
```

```
                .expenseAmount(expense.getExpenseAmount())
                .build();
  }

    private Expense mapFromDto(ExpenseDto expense) {
        return Expense.builder()
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
                .expenseAmount(expense.getExpenseAmount())
                .build();
  }
}
```

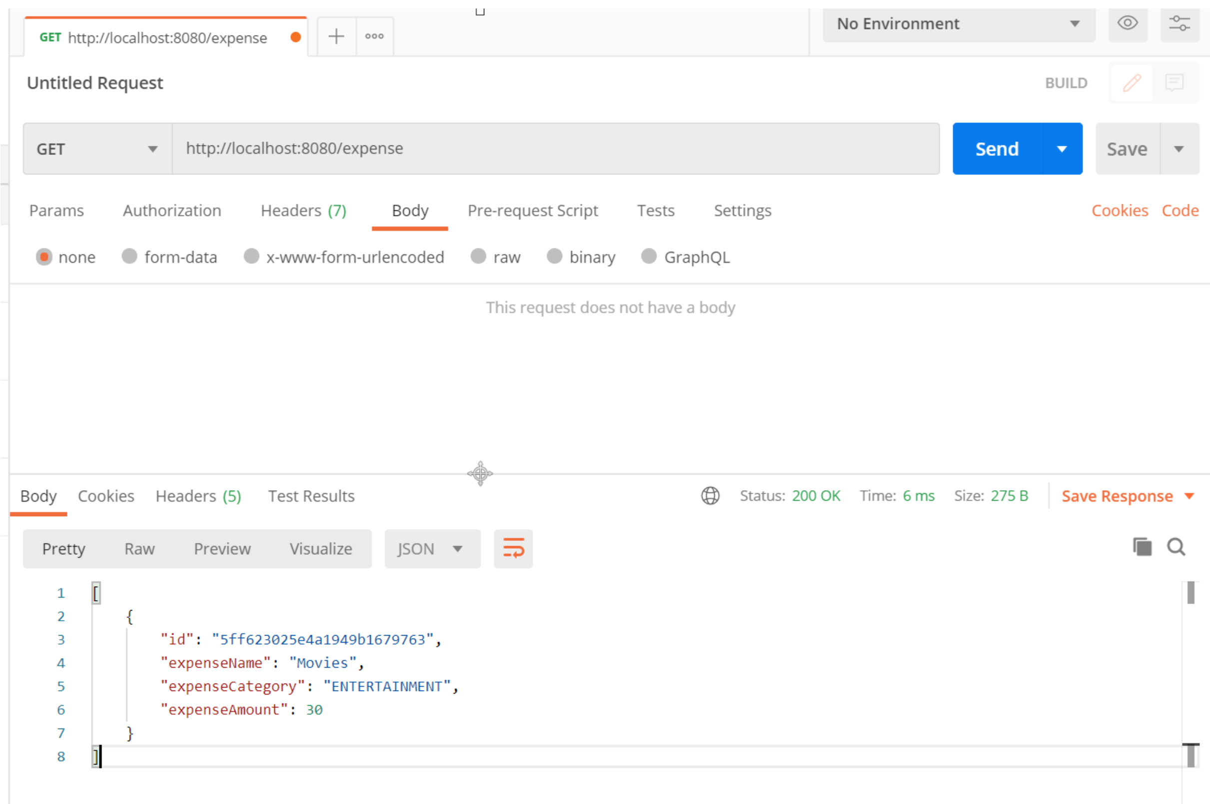
In this case we are just reading all the `Expense` objects inside the database, mapping them to `ExpenseDto` and returning them to the API layer.

```
ExpenseController.java

package com.programming.techie.expensetracker.web;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.model.Expense;
import
com.programming.techie.expensetracker.service.ExpenseService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import
org.springframework.web.servlet.support.ServletUriComponentsBuild
er;

import java.net.URI;
import java.util.List;

@RestController
@RequestMapping("/api/xpense")
@RequiredArgsConstructor
public class ExpenseController {

    private final ExpenseService expenseService;

  @PostMapping
   public ResponseEntity<Void> addExpense(@RequestBody ExpenseDto
expenseDto) {
        String expenseId = expenseService.addExpense(expenseDto);
  URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
```

```
                .path("/{id}")
                .buildAndExpand(expenseId)
                .toUri();
 return ResponseEntity.created(location)
                .build();
  }

 @GetMapping
 @ResponseStatus(HttpStatus.OK)
 public List<ExpenseDto> getAllExpenses() {
        return expenseService.getAllExpenses();
 }

    @GetMapping("/{name}")
    @ResponseStatus(HttpStatus.OK)
    public ExpenseDto getExpenseByName(@PathVariable String name)
{
        return expenseService.getExpense(name);
  }

}
```

Testing GET Request

Let’s start our Spring Boot Application and make the REST API call to Get All Expenses.



Update Expense

To update an Expense, we are going to use a PUT request, using the `@PutMapping` annotation.

```
ExpenseService.java

package com.programming.techie.expensetracker.service;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import
com.programming.techie.expensetracker.exception.ExpenseNotFoundEx
ception;
import com.programming.techie.expensetracker.model.Expense;
import
com.programming.techie.expensetracker.repository.ExpenseRepositor
y;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
```

```
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
@Transactional
public class ExpenseService {

    private final ExpenseRepository expenseRepository;

 public String addExpense(ExpenseDto expenseDto) {
        Expense expense = mapFromDto(expenseDto);
 return expenseRepository.insert(expense).getId();
  }

    public void updateExpense(ExpenseDto expenseDto) {
                                    Expense    savedExpense    =
expenseRepository.findById(expenseDto.getId())
                                        .orElseThrow(()   ->   new
RuntimeException(String.format("Cannot  Find  Expense  by  ID  %s",
expenseDto.getId())));
  savedExpense.setExpenseName(expenseDto.getExpenseName());

savedExpense.setExpenseCategory(expenseDto.getExpenseCategory());
  savedExpense.setExpenseAmount(expenseDto.getExpenseAmount());

  expenseRepository.save(savedExpense);
  }

    public ExpenseDto getExpense(String name) {
        Expense expense = expenseRepository.findByName(name)
                                        .orElseThrow(()   ->   new
ExpenseNotFoundException(String.format("Cannot  Find  Expense  by
Name - %s", name)));
 return mapToDto(expense);
  }

    public List<ExpenseDto> getAllExpenses() {
        return expenseRepository.findAll()
                .stream()

.map(this::mapToDto).collect(Collectors.toList());
  }

    private ExpenseDto mapToDto(Expense expense) {
        return ExpenseDto.builder()
                .id(expense.getId())
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
                .expenseAmount(expense.getExpenseAmount())
```

```
                 .build();
  }

    private Expense mapFromDto(ExpenseDto expense) {
        return Expense.builder()
                .expenseName(expense.getExpenseName())
                .expenseCategory(expense.getExpenseCategory())
                .expenseAmount(expense.getExpenseAmount())
                .build();
  }
}
```

We are receiving the `Expense` we need to

```
ExpenseController.java

package com.programming.techie.expensetracker.web;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.model.Expense;
import
com.programming.techie.expensetracker.service.ExpenseService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import
org.springframework.web.servlet.support.ServletUriComponentsBuild
er;

import java.net.URI;
import java.util.List;

@RestController
@RequestMapping("/api/xpense")
@RequiredArgsConstructor
public class ExpenseController {

    private final ExpenseService expenseService;

  @PostMapping
   public ResponseEntity<Void> addExpense(@RequestBody ExpenseDto
expenseDto) {
        String expenseId = expenseService.addExpense(expenseDto);
  URI location = ServletUriComponentsBuilder
                 .fromCurrentRequest()
                 .path("/{id}")
                 .buildAndExpand(expenseId)
```

```
                .toUri();
return ResponseEntity.created(location)
                .build();
  }

    @PutMapping
 @ResponseStatus(HttpStatus.OK)
    public void updateExpense(@RequestBody Expense expense) {
        expenseService.updateExpense(expense);
  }

    @GetMapping
 @ResponseStatus(HttpStatus.OK)
    public List<ExpenseDto> getAllExpenses() {
        return expenseService.getAllExpenses();
  }

    @GetMapping("/{name}")
    @ResponseStatus(HttpStatus.OK)
    public ExpenseDto getExpenseByName(@PathVariable String name)
{
        return expenseService.getExpense(name);
  }

}
```

Response Status for PUT Requests

We are also returning the status as OK(200) for PUT Requests.

Testing PUT Request

Let's start our Spring Boot Application and make the REST API call to Update Expense.

PUT http://localhost:8080/expense

No Environment

Untitled Request

BUILD

PUT http://localhost:8080/expense

Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
    "id": "5ff623025e4a1949b1679763",
    "expenseName": "Movies",
    "expenseCategory": "ENTERTAINMENT",
    "expenseAmount": 30
}
```

Body Cookies Headers (4) Test Results Status: 200 OK Time: 120 ms Size: 123 B Save Response

Pretty Raw Preview Visualize Text

```
1
```

GET http://localhost:8080/expense

No Environment

Untitled Request

BUILD

GET http://localhost:8080/expense

Send Save

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (5) Test Results Status: 200 OK Time: 6 ms Size: 275 B Save Response

Pretty Raw Preview Visualize JSON

```
[
    {
        "id": "5ff623025e4a1949b1679763",
        "expenseName": "Movies",
        "expenseCategory": "ENTERTAINMENT",
        "expenseAmount": 30
    }
]
```

Delete Expense

To implement the DELETE Endpoint we can use the `@DeleteMapping` annotation.

ExpenseService.java

```
    package com.programming.techie.expensetracker.service;

    import com.programming.techie.expensetracker.dto.ExpenseDto;
                                                            import
com.programming.techie.expensetracker.exception.ExpenseNotFoundEx
ception;
    import com.programming.techie.expensetracker.model.Expense;
                                                            import
com.programming.techie.expensetracker.repository.ExpenseRepositor
y;
    import lombok.RequiredArgsConstructor;
    import org.springframework.stereotype.Service;
                                                            import
org.springframework.transaction.annotation.Transactional;

    import java.util.List;
    import java.util.stream.Collectors;

    @Service
    @RequiredArgsConstructor
    @Transactional
    public class ExpenseService {

        private final ExpenseRepository expenseRepository;

     public String addExpense(ExpenseDto expenseDto) {
            Expense expense = mapFromDto(expenseDto);
     return expenseRepository.insert(expense).getId();
      }

        public void updateExpense(ExpenseDto expenseDto) {
                                   Expense   savedExpense   =
expenseRepository.findById(expenseDto.getId())
                                     .orElseThrow(()  ->  new
RuntimeException(String.format("Cannot Find Expense by ID %s",
expenseDto.getId())));
      savedExpense.setExpenseName(expenseDto.getExpenseName());

savedExpense.setExpenseCategory(expenseDto.getExpenseCategory());
```

```
savedExpense.setExpenseAmount(expenseDto.getExpenseAmount());

      expenseRepository.save(savedExpense);
      }

        public ExpenseDto getExpense(String name) {
            Expense expense = expenseRepository.findByName(name)
                                          .orElseThrow(() -> new
ExpenseNotFoundException(String.format("Cannot Find Expense by
Name - %s", name)));
     return mapToDto(expense);
      }

        public List<ExpenseDto> getAllExpenses() {
            return expenseRepository.findAll()
                    .stream()

.map(this::mapToDto).collect(Collectors.toList());
      }

      public void deleteExpense(String id) {
         expenseRepository.deleteById(id);
      }

        private ExpenseDto mapToDto(Expense expense) {
            return ExpenseDto.builder()
                    .id(expense.getId())
                    .expenseName(expense.getExpenseName())

.expenseCategory(expense.getExpenseCategory())
                    .expenseAmount(expense.getExpenseAmount())
                    .build();
      }

        private Expense mapFromDto(ExpenseDto expense) {
            return Expense.builder()
                    .expenseName(expense.getExpenseName())

.expenseCategory(expense.getExpenseCategory())
                    .expenseAmount(expense.getExpenseAmount())
                    .build();
      }
    }
```

ExpenseController.java

```
package com.programming.techie.expensetracker.web;
```

```
import com.programming.techie.expensetracker.dto.ExpenseDto;
import com.programming.techie.expensetracker.model.Expense;
import
com.programming.techie.expensetracker.service.ExpenseService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import
org.springframework.web.servlet.support.ServletUriComponentsBuild
er;

import java.net.URI;
import java.util.List;

@RestController
@RequestMapping("/api/xpense")
@RequiredArgsConstructor
public class ExpenseController {

    private final ExpenseService expenseService;

  @PostMapping
   public ResponseEntity<Void> addExpense(@RequestBody ExpenseDto
expenseDto) {
        String expenseId = expenseService.addExpense(expenseDto);
  URI location = ServletUriComponentsBuilder
                .fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(expenseId)
                .toUri();
 return ResponseEntity.created(location)
                .build();
  }

    @PutMapping
 @ResponseStatus(HttpStatus.OK)
    public void updateExpense(@RequestBody Expense expense) {
        expenseService.updateExpense(expense);
  }

    @GetMapping
 @ResponseStatus(HttpStatus.OK)
    public List<ExpenseDto> getAllExpenses() {
        return expenseService.getAllExpenses();
  }

    @GetMapping("/{name}")
    @ResponseStatus(HttpStatus.OK)
```

```
    public ExpenseDto getExpenseByName(@PathVariable String name)
{
        return expenseService.getExpense(name);
  }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void deleteExpense(@PathVariable String id) {
        expenseService.deleteExpense(id);
  }

}
```

Response Status for DELETE Requests

As we are going to Delete a Resource, the appropriate Response Status should be NO_CONTENT(204).

Testing DELETE Request

Let’s start our Spring Boot Application and make the REST API call to Delete Expense.

PUT http://localhost:8080/expense

No Environment

Untitled Request

BUILD

PUT http://localhost:8080/expense

Send Save

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
{
    "id": "5ff623025e4a1949b1679763",
    "expenseName": "Movies",
    "expenseCategory": "ENTERTAINMENT",
    "expenseAmount": 30
}
```

Body Cookies Headers (4) Test Results Status: 200 OK Time: 120 ms Size: 123 B Save Response

Pretty Raw Preview Visualize Text

```
1
```

GET http://localhost:8080/expense

No Environment

Untitled Request

BUILD

GET http://localhost:8080/expense

Send Save

Params Authorization Headers (7) Body Pre-request Script Tests Settings Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL

This request does not have a body

Body Cookies Headers (5) Test Results Status: 200 OK Time: 19 ms Size: 166 B Save Response

Pretty Raw Preview Visualize JSON

```
[]
```

Error Handling in REST API

While Implementing GET All Expenses and GET Single Expense Endpoints, we are throwing an `ExpenseNotFoundException`

Here are the ways we can handle the Exceptions:

Using @ExceptionHandler Annotation

You can add a method which contains the `@ExceptionHandler` annotation, inside the Controller itself, for example inside `ExpenseController.java`

```
@RestController
@RequestMapping("/api/expense")
@RequiredArgsConstructor
public class ExpenseController {

    ..

    @ExceptionHandler({ExpenseNotFoundException.class})
      @ResponseStatus(value = HttpStatus.BAD_REQUEST, reason =
"Cannot Find Expense with the given data")
    public void handleException() {
        // Do Nothing
  }

}
```

The major drawback in this approach is this Exception Handler is only applicable for the `ExpenseController.java`. For any kind of non-trivial applications, there will be more than one controller, so in that case, its not so practical to repeat this information in all our controllers.

Using @ControllerAdvice

We can define a Global Exception Handler using the `@ControllerAdvice` annotation.

ExpenseNotFoundExceptionHandler.java

```
package com.programming.techie.expensetracker.exception;

import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import
org.springframework.web.bind.annotation.ControllerAdvice;
import
org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseE
ntityExceptionHandler;

@ControllerAdvice
public    class    ExpenseNotFoundExceptionHandler    extends
ResponseEntityExceptionHandler {
    @ExceptionHandler(value
            = {ExpenseNotFoundException.class})
    protected ResponseEntity<Object> handleConflict(
            RuntimeException ex, WebRequest request) {
        String bodyOfResponse = ex.getMessage();
 return handleExceptionInternal(ex, bodyOfResponse,
 new HttpHeaders(), HttpStatus.BAD_REQUEST, request);
  }
}
```

As you can see we are using the `@ExceptionHandler` inside a `ExpenseNotFoundExceptionHandler` class which extends `ResponseEntityExceptionHandler`

Whenever, there is an `ExpenseNotFoundException` thrown in our application, our clients will receive **HttpStatus.BAD_REQUEST** as response, with an appropriate error message we are setting during creating the Exception.

Using ResponseStatusException class

Since `Spring 5.0` instead of throwing a seperate exception like `ExpenseNotFoundException` we can directly throw the `ResponseStatusException`

Example:

```
Expense savedExpense =
expenseRepository.findById(expense.getId())
.orElseThrow(() -> new
ResponseStatusException(HttpStatus.BAD_REQUEST,
    String.format("Cannot Find Expense by ID %s",
expense.getId())));
```

The downside of using this class is we have to duplicate the code across muliple classes, and it's also difficult to enforce application wide exception handling.

The recommended way is to use the `@ControllerAdvice` approach.

But whatever approach you choose, make sure to follow that consistently across your project.

Testing REST API

We can make use of the Spring Testing Library to test our REST API.

The Spring Test Framework provides us with an annotation called `@WebMvcTest` which is a specialized annotation which will create the Spring Context for us with only beans which are related to the Spring MVC components like `@Controller`, `@RestController`, `@AutoconfigureWebMvc` etc.

Let's create the `ExpenseControllerTest`

```
package com.programming.techie.expensetracker.web;

import com.programming.techie.expensetracker.dto.ExpenseDto;
import
com.programming.techie.expensetracker.model.ExpenseCategory;
import
com.programming.techie.expensetracker.service.ExpenseService;
import org.junit.Test;
import org.junit.jupiter.api.DisplayName;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTes
t;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.HttpHeaders;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import
org.springframework.test.web.servlet.result.MockMvcResultMatchers
;

import java.math.BigDecimal;

import static org.junit.jupiter.api.Assertions.assertTrue;
import                                                       static
org.springframework.test.web.servlet.request.MockMvcRequestBuilde
rs.post;

@WebMvcTest(controllers = ExpenseController.class)
public class ExpenseControllerTest {
```

```
    @MockBean
  private ExpenseService expenseService;
  @Autowired
  private MockMvc mockMvc;

  @Test
 @DisplayName("Should Create Expense")
    public void shouldCreateExpense() throws Exception {
        ExpenseDto expenseDto = ExpenseDto.builder()
                .expenseCategory(ExpenseCategory.ENTERTAINMENT)
                .expenseName("Movies")
                .expenseAmount(BigDecimal.TEN)
                .build();


Mockito.when(expenseService.addExpense(expenseDto)).thenReturn("1
23");

  MvcResult mvcResult = mockMvc.perform(post("/api/expense"))

.andExpect(MockMvcResultMatchers.status().isCreated())

.andExpect(MockMvcResultMatchers.header().exists(HttpHeaders.LOCA
TION))
                .andReturn();


assertTrue(mvcResult.getResponse().getHeaderValue(HttpHeaders.LOC
ATION).toString().contains("123"));

   }
}
```

We are setting up the needed Spring Context for our REST API Test using the `@WebMvcTest` and `MockMvc` to create a mocked out Servlet Environment to fire mocked HTTP Requests.

To make the REST API call we are using the `mockMvc.perform()` method , and with the return type we are making assertion that the required HTTP STATUS is 201, followed by the assertions for the LOCATION Header.

If you want to learn more about Testing Spring Boot Applications, have a look at the Youtube Series - where I show you how to test a complete Spring Boot Application.

Documenting the REST API

In this chapter, we are going to learn how to document our REST APIs.

Why should we document our REST APIs?

In the real world, consumers of an API should have a good understanding of the REST APIs they are using. Having good documentation is vital in helping the users to use the API effectively.

Having good documentation for our REST APIs is necessary. On the other hand, maintaining the documentation manually is tiresome and error-prone.

##Generating REST API Documentation using Swagger and Springfox

Swagger and Springfox makes this process of generating REST API documentation quick and painless. Using these tools, we can automate the process of documentation.

What is Swagger?

So what is Swagger? It is an OPEN API specification that is created as a standard to describe your REST API.

As we are using Springboot to develop our REST API we can use a library called as [Springfox](#) to automatically create JSON Documentation.

Adding Springfox Dependencies to project

Inside our `pom.xml` file, add the following maven dependencies. This should download the required springfox dependencies to our project.

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
```

```
        <version>3.0.0</version>
    </dependency>
```

Configure Swagger and Springfox

Now it's time to configure Swagger and Springfox in our project, for that we will create a configuration class called `SwaggerConfiguration`.

This class is marked with annotation `@Configuration` and `@EnableSwagger2`

```
package com.programming.techie.expensetracker.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import
springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
  public Docket expenseTrackerApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(PathSelectors.any())
                .build()
                .apiInfo(getApiInfo());
  }

    private ApiInfo getApiInfo() {
        return new ApiInfoBuilder()
                .title("Expense Tracker API")
                .version("1.0")
                          .description("API  for  Expense  Tracker
Application")
                         .contact(new  Contact("Sai  Upadhyayula",
"http://programmingtechie.com", "xyz@email.com"))
```

```
                .license("Apache License Version 2.0")
                .build();
  }
}
```

Inside the `SwaggerConfiguration.java` class, we created a Bean with the name expenseTrackerApi, this can be anything you like.

Inside the bean, we are creating a new Docket which is a Springfox internal class and we are specifying the Documentation Type as Swagger2, everything which is returned inside the `expenseTrackerAPI()` method is the standard defaults for Springfox.

We have also some API information through the `apiInfo()` method.

Now let's import this configuration file to our `ExpenseTrackerRestApiApplication`

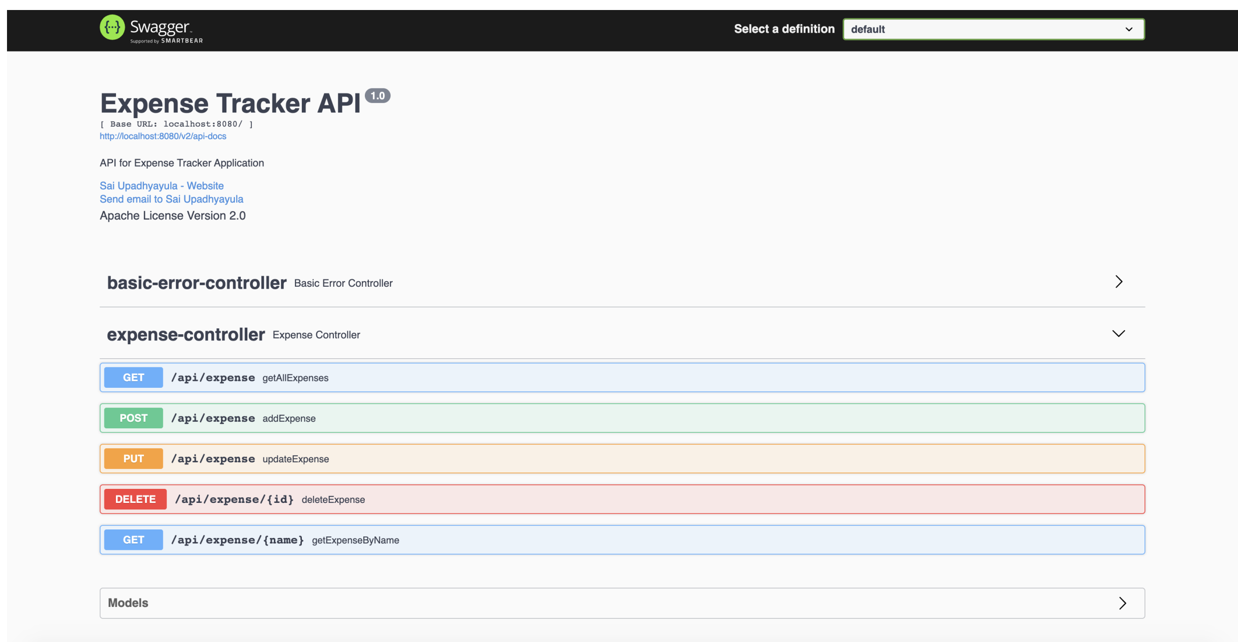
```
package com.programming.techie.expensetracker;

import
com.programming.techie.expensetracker.config.SwaggerConfiguratio
n;
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Import;

@SpringBootApplication
@Import(SwaggerConfiguration.class)
public class ExpenseTrackerRestApiApplication {

    public static void main(String[] args) {

SpringApplication.run(ExpenseTrackerRestApiApplication.class,
args);
  }

}
```

We have added `@Import(SwaggerConfiguration.class)` to our `ExpenseTrackerRestApiApplication`. This should enable Swagger and Springfox in our application. Now let's see how this all works.

###How Springfox works? Springfox scans our backend application and looks for all the Controllers and related components when starting up the application and it automatically generates the documentation for our REST API.

Using springfox-swagger-ui, it constructs a webpage where we can see the documentation for our REST API.

Once we start the application, we can see the documentation at **http://localhost:8080/swagger-ui/**



You can also make requests to the API using the Swagger UI.

Conclusion

I hope this ebook was helpful in improving your understanding of how to build a REST API.

You can find the source code of the Expense Tracker API Project at Github.