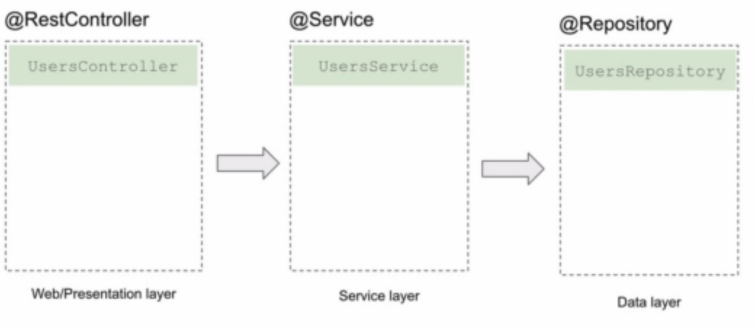
Unit test in Spring-Boot application, HTTP-clients

Unit-testing in Spring-Boot application:

Till now we have discussed unit testing in regular Java applications. now we will see the unit testing in the Spring-boot application.

As we know that we developed Spring-boot applications using layered architecture:



Here using Controller we expose the Rest API to the clients.

The Service class contains methods that perform the main business logic like sorting, and filtering. To do this the service class will communicate with the Data Access layer/Repository to get the data from the Database.

Now let's assume that we need to do the unit test with the following method of the controller in isolation from the service class.

```
@GetMapping("/customers")
public List<Customer> getAllCustomersHandler(){
--
--
}
```

To Isolation the above method of the controller class from the Service class, we need to Mock the Service class, and if we mock the Service class then the Repository will also not be invoked to test our controller method.

Here to test the getAllCustomersHandler() method we should also not start the Web server, because in unit testing we will not send the real HTTP request. This means that annotations like @Restcontroller or @GetMapping will also not be involved.

Our production application code will have these annotations, but our unit tests will not use them. in fact, unit-testing code in the Spring-boot application should not load the Spring container also.

So we will be able to unit test our getAllCustomersHandler() method in isolation from all other layers and from all other dependencies.

But sometimes we do need to test if our code is properly integrated with the Spring framework and this is when we will start writing **integration tests**.

Integration Testing of Controller/Web-layer layer:

Now if we need to test the getAllCustomersHandler() method properly integrated with Spring f/w and it works well in the controller layer, we do not want to isolate the getAllCustomersHandler() method completely.

We do want some of the spring f/w features to be available to us, but only to the controller layer.

For example, we want to test and make sure that the getAllCustomersHandler() method can be triggered by an HTTP GET request. and can read the HTTP request parameters or data. so in this case, we want annotations like @RestController, @GetMapping, @RequestMapping, @RequestBody to be enabled and to work.

So we do want actual integration with spring f/w in the controller layer. but because we are not testing the other 2 layers, we do not need these layers to be enabled or loaded into the spring application context. for this, spring f/w does allow us to test each layer of our application architecture separately, and there is a way to test the controller layer separately from the service and data access layer. and in this case, Spring f/w will create the spring beans that are related to the controller layer only, other layer-related beans will not be loaded into the Spring container.

The unit test that we will create in this case will be called the integration test because we are no longer testing the getAllCustomersHandler() method in isolation completely, it is tested with the integration with Spring f/w.

Integration Test with All Layers:

If we need to test how our method under test works with all layers integrated, we will write a test method that is called an integration test or some developers call it an acceptance test.

For this type of test, we will need all three layers integrated and we will not use any mock objects.
This type of test method will test how our getAllCustomersHandler() method works from end-to-end, meaning that in the controller layer, HTTP requests will be handled and bean validation will be performed. and in the service layer, a real production version of our code will be executed and in the data access layer, the actual communication with the database will be performed. and we need to start the spring container for all the layers in our application.

And to do that Spring f/w provides us with a special annotation that is called **@SpringBootTest.**

@SpringBootTest annotation will create the spring container that is very similar to the one we use in the production application, although by default it will not start a web server. but even though a real web server is not started, we can still test our code integrated with all layers. and if needed we can configure **@SpringBootTest** annotation to load a real web environment with an embedded server running on a specific port or even make it run on a random port.

Adding Testing support to the Spring Boot application:

Spring boot provides very good support for the unit testing and integration testing in our application.

Just by adding the **Web dependency** in the Spring-boot project, we also get the **Spring boot starter test** dependency by default.

This **Spring boot starter test** dependency internally adds all the Junit-Jupiter, Mockito, and other testing libraries.

If we add spring security to our project then the **spring-security-test** dependency also will be included. this dependency will help us to test if some of the API endpoints users are required to be authenticated.

for the downloaded application, by default one test class is created inside the **src/test/java** folder as follows:

```
package com.masai;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class <MainClassName>Tests {

  @Test
  void contextLoads() {
  }

}
```

Let’s create a Spring Boot application: (Without Using Spring Security)

- Create a simple Spring Boot application **EmployeeTestingApp** by adding the following dependencies:
  - Spring Web
  - Devtools
  - Spring Data JPA
  - MySQL driver
  - Spring Boot starter validation

```
application.properties:
-----------------------

#db specific properties
spring.datasource.url=jdbc:mysql://localhost:3306/masaidb
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=root

#ORM s/w specific properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

com.masai.model package:

```
package com.masai.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

@Entity
public class Employee {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Integer employeeId;

  @Size(min = 3, max = 10, message = "Name Length should be min 3 and max 10")
  @NotNull(message = "name is mandatory")
  private String name;

  @NotNull(message = "Address is mandatory")
  private String address;

  @NotNull(message = "Salary is mandatory")
  @Min(value = 10000, message = "Salary should be minimum 10000")
  @Max(value = 100000, message = "Salary should be maximum 100000")
  private Integer salary;

  public Employee() {
    // TODO Auto-generated constructor stub
  }

  public Employee(Integer employeeId, String name, String address, Integer salary) {
    super();
    this.employeeId = employeeId;
    this.name = name;
    this.address = address;
    this.salary = salary;
  }

  public Integer getEmployeeId() {
    return employeeId;
  }

  public void setEmployeeId(Integer employeeId) {
    this.employeeId = employeeId;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getAddress() {
    return address;
  }

  public void setAddress(String address) {
    this.address = address;
  }

  public Integer getSalary() {
    return salary;
  }

  public void setSalary(Integer salary) {
    this.salary = salary;
  }

  @Override
  public String toString() {
    return "Employee [employeeId=" + employeeId + ", name=" + name + ", address=" + address + ", salary=" + salary
        + "]";
  }
```

```
}
```

com.masai.repository package:

```
package com.masai.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.masai.model.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Integer>{

}
```

com.masai.exception package:

```
EmployeeException.java:
----------------------

package com.masai.exception;

public class EmployeeException extends RuntimeException{

  public EmployeeException(String message) {
    super(message);
  }

}


MyErrorDetails.java:
--------------------

package com.masai.exception;

import java.time.LocalDateTime;

public class MyErrorDetails {

  private LocalDateTime timestamp;
  private String message;
  private String details;

  //getters and setters

}


GlobalExceptionHandler.java:
----------------------------

package com.masai.exception;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;

@ControllerAdvice
public class GlobalExceptionHandler {

  @ExceptionHandler(EmployeeException.class)
```

```
  public ResponseEntity<MyErrorDetails> employeeExceptionHandler(EmployeeException ie, WebRequest req) {
    System.out.println("inside myHandler method...");

    MyErrorDetails err = new MyErrorDetails();
    err.setTimestamp(LocalDateTime.now());
    err.setMessage(ie.getMessage());
    err.setDetails(req.getDescription(false));

    ResponseEntity<MyErrorDetails> re = new ResponseEntity<>(err, HttpStatus.BAD_REQUEST);

    return re;
  }

  // to handle generic any type of Exception
  @ExceptionHandler(Exception.class)
  public ResponseEntity<MyErrorDetails> exceptionHandler(Exception e, WebRequest req) {

    MyErrorDetails err = new MyErrorDetails();
    err.setTimestamp(LocalDateTime.now());
    err.setMessage(e.getMessage());
    err.setDetails(req.getDescription(false));

    return new ResponseEntity<>(err, HttpStatus.BAD_REQUEST);

  }

  @ExceptionHandler(MethodArgumentNotValidException.class)
  public ResponseEntity<MyErrorDetails> myMANVExceptionHandler(MethodArgumentNotValidException me) {

    MyErrorDetails err = new MyErrorDetails();
    err.setTimestamp(LocalDateTime.now());
    err.setMessage("Validation Error");
    err.setDetails(me.getBindingResult().getFieldError().getDefaultMessage());

    return new ResponseEntity<>(err, HttpStatus.BAD_REQUEST);

  }

}
```

com.masai.service package:

```
EmployeeService.java:
---------------------
package com.masai.service;

import java.util.List;

import com.masai.exception.EmployeeException;
import com.masai.model.Employee;

public interface EmployeeService {

  public Employee registerEmployee(Employee employee);

  public Employee getEmployeeById(Integer empId)throws EmployeeException;

  public List<Employee> getAllEmployeeDetails()throws EmployeeException;

}


EmployeeServiceImpl.java:
-------------------------

package com.masai.service;

import java.util.List;
import java.util.Optional;
```

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

import com.masai.exception.EmployeeException;
import com.masai.model.Employee;
import com.masai.repository.EmployeeRepository;

@Service
public class EmployeeServiceImpl implements EmployeeService {

  @Autowired
  private EmployeeRepository employeeRepository;


  @Override
  public Employee registerEmployee(Employee employee) {

    return employeeRepository.save(employee);


  }

  @Override
  public Employee getEmployeeById(Integer empId) throws EmployeeException {


    Optional<Employee> opt= employeeRepository.findById(empId);

    if(opt.isPresent())
      return opt.get();
    else
      throw new EmployeeException("Employee does not exist with Id: "+empId);



  }

  @Override
  public List<Employee> getAllEmployeeDetails() throws EmployeeException {

    List<Employee> employees= employeeRepository.findAll();

    if(employees.isEmpty())
      throw new EmployeeException("No Employee found..");
    else
      return employees;


  }
}
```

com.masai.controller package:

```
EmployeeController.java
-----------------------

package com.masai.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.masai.model.Employee;
```

```
import com.masai.service.EmployeeService;

import jakarta.validation.Valid;
import jakarta.websocket.server.PathParam;

@RestController
public class EmployeeController {

  @Autowired
  private EmployeeService employeeService;

  @PostMapping("/employees")
  public ResponseEntity<Employee> registerEmployeeHandler(@Valid @RequestBody Employee employee) {

    Employee registeredEmployee = employeeService.registerEmployee(employee);

    return new ResponseEntity<>(registeredEmployee, HttpStatus.CREATED);

  }

  @GetMapping("/employees/{id}")
  public ResponseEntity<Employee> getEmployeeByIdHandler(@PathVariable("id") Integer empId) {

    Employee employee = employeeService.getEmployeeById(empId);

    return new ResponseEntity<>(employee, HttpStatus.OK);

  }

  @GetMapping("/employees")
  public ResponseEntity<List<Employee>> getAllEmployeeHandler() {

    List<Employee> employees = employeeService.getAllEmployeeDetails();

    return new ResponseEntity<>(employees, HttpStatus.OK);
  }

}
```

Let’s write the Integration test of the above EmployeeController class (Integration with Spring framework):

@WebMvcTest annotation:

To specify the spring f/w that we want to create application context only for those beans that are related to controller layer, we need to annotate this class with a special annotation **@WebMvcTest.**

because we annotate our test class with **@WebMvcTest** annotation, it will tell spring f/w that we are interested in testing only controller-layer only. so it will make spring f/w scan our spring boot application only for a limited number of classes, classes that are related to controller-layer only.
for example, it will scan and create controllers beans, but it will not scan the service layer and data access layer beans.

We can limit this annotation to work with only one controller class that we need.

Example:

```
  @WebMvcTest(controllers = EmployeeController.class) // without this, it will load all the Controllers
  public class EmployeeControllerTest{
  --
  }
```

Testing Spring Boot application in which Spring Security is included:

If we test the Spring boot application which uses the spring security, there are security filters will be enabled by default.

If we are not interested in those security filters at the time of testing our controller methods so we can exclude those security filters. to do that we need to use following annotation:

```
@WebMvcTest(controllers = EmployeeController.class) // without this, it will load all the Controllers
@AutoConfigureMockMvc(addfilters = false)
public class EmployeeControllerTest{
  --
}
```

MockMvc:

MockMvc is a Spring Boot test tool class that lets us test controllers without needing to start an HTTP server. In these tests the application context is loaded and we can test the web layer as if it receiving the requests from the HTTP server without the hustle of actually starting it.

RequestBuilder and MockMvcRequestBuilders:

The `MockMvc::perform` method is used to send mock HTTP servlet requests to the `TestDispatcherServlet`. It accepts a `RequestBuilder` as a parameter.

The `MockMvcRequestBuilders` class has static factory methods used to create a `MockMvcRequestBuilder`. (`MockMvcRequestBuilder` is an implementation of `RequestBuilder`.) This argument is passed to the `MockMvc::perform` method.

@MockBean annotation:

We can use the *@MockBean* to add mock objects to the Spring IOC container. The mock will replace any existing bean of the same type in the IOC container.

If no bean of the same type is defined, a new one will be added. This annotation is useful in integration tests where a particular bean, like an external service, needs to be mocked.

When we use the annotation on a field, the mock will be injected into the field, as well as being registered in the Spring IOC container.

Example:

- Create a package **com.masai.controller** inside the **src/test/java** folder.
- Create a class **EmployeeControllerTest.java** inside that package:

Example:

```
EmployeeControllerTest.java:
---------------------------
package com.masai.controller;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.mockito.ArgumentMatchers.any;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.MvcResult;
import org.springframework.test.web.servlet.RequestBuilder;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.masai.model.Employee;
import com.masai.service.EmployeeService;

@WebMvcTest(controllers = EmployeeController.class)
//@WebMvcTest(controllers = EmployeeController.class, excludeAutoConfiguration = SecurityAutoConfiguration.class)
public class EmployeeControllerTest {

  @Autowired
  private MockMvc mockMvc;

  @MockBean
  private EmployeeService employeeService;

  private Employee requestemployee;
  private Employee responseemployee;

  @BeforeEach
  public void init() {

    requestemployee = new Employee();
    requestemployee.setName("Ram");
    requestemployee.setAddress("delhi");
    requestemployee.setSalary(80000);

    responseemployee = new Employee();
    responseemployee.setEmployeeId(10);
    responseemployee.setName("Ram");
    responseemployee.setAddress("delhi");
    responseemployee.setSalary(80000);

  }

  @Test
  @DisplayName("Employee can be created")
  public void testRegisterEmployeeHandler_whenValidDetailsProvided_returnRegisteredEmployee() throws Exception {

    // Arrange
    Mockito.when(employeeService.registerEmployee(any(Employee.class))).thenReturn(responseemployee);

    RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/employees")
        .contentType(MediaType.APPLICATION_JSON_VALUE)
        .content(new ObjectMapper().writeValueAsString(requestemployee));

    // Act
    MvcResult mvcResult = mockMvc.perform(requestBuilder).andReturn();

    String responseBodyAsString = mvcResult.getResponse().getContentAsString();

    Employee createdEmployee = new ObjectMapper().readValue(responseBodyAsString, Employee.class);
```

```
    // Assert
    assertEquals(responseemployee.getName(), createdEmployee.getName(),
        "returned created Employee Name is incorrect");

    assertNotNull(createdEmployee.getEmployeeId(), "created Customer id should not be empty");

  }

  @Test
  @DisplayName("name size should be min 3 and max 10 charecters")
  void testCreateCustomer_whenFirstNameIsOnlyOneCharecter_returns400StatusCode() throws Exception {

    // Arrange

    Mockito.when(employeeService.registerEmployee(any(Employee.class))).thenReturn(responseemployee);
    requestemployee.setName("R"); // lets provide invalid Name

    RequestBuilder requestBuilder = MockMvcRequestBuilders.post("/employees")
        .contentType(MediaType.APPLICATION_JSON_VALUE).accept(MediaType.APPLICATION_JSON_VALUE)
        .content(new ObjectMapper().writeValueAsString(requestemployee));

    // Act
    MvcResult mvcResult = mockMvc.perform(requestBuilder).andReturn();

    // Assert
    assertEquals(HttpStatus.BAD_REQUEST.value(), mvcResult.getResponse().getStatus(),
        "Http Status code is not set to 400");

  }

}
```

Testing All Layers of Spring boot:

Let’s try to write the test case for the API which will engage all 3 layers(integration test). here we will not mock service layer or data-access layer.

- Our test method will perform http request which trigger code in controller class and from the controller class will invoke the code in the service class and our code in the service class will actually invoke code in the data access layer and data access layer code actually read or write the information from a database.
- This time we will not disable any spring security configuration, and to communicate with any protected API endpoints, our http request will need to include a valid JWT token.

@SpringBootTest annotation:

To make spring f/w create and run spring application context(Spring Container) that involves all three layers of our application we need to annotate our test class with **@SpringBootTest** annotation.

- This annotation will tell the Spring boot to look for the main application class (class having the main method) and it will use it to start the spring application context. so spring f/w will then scan the root package of our application and its sub packages looking for the classes with different types of annotations.
- It will create beans for all three layers of our application.
- We can also use this annotation to specify if we want spring f/w to run web environment on a specific port or on a random port number. So when we annotate our test class with @SpringBootTest annotation, we can also specify the type of web environment that we want to use for our test

Example:

```
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
//@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
//@SpringBootTest(webEnvironment = WebEnvironment.MOCK) // it uses the Mock server as previous example only for controller layer
public class EmployeeControllerTest {

}
```

Note: With WebEnvironment.Mock we can not use RestTemplate or TestRestTemplate class to call the API because it does not load the real HTTP server, Here we need to make use of @MockMvc like in previous example.

@TestPropertySource annotation:

For test environment we can create a separate configuration file called **application-test.properties** file inside the **src/main/resource** folder.

in this file we can mention the server port, data source configuration details, etc. for the testing environment.

Example:

```
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
@TestPropertySource(locations = "/application-test.properties")
public class EmployeeControllerTest {

  @Value("${server.port}")
  private Integer serverPort;

  @Test
  public void test() {
    System.out.println("server port is :"+serverPort);
  }
}
```

Let's add in-memory database H2 configuration inside the **application-test.properties** file:

```
server.port=8888

spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enable=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show_sql=true
```

Note: to work with h2 database we need to add the following dependencies inside the pom.xml file:

```
<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

In normal application if we are using H2 database then we can open the console of H2 database using the following URL:

```
http://localhost:8080/h2-console

with the username="sa" and password= (empty)
```

But in testing environment we can not open this console because the server will be stopped after running our test code.

EmployeeControllerTest.java:

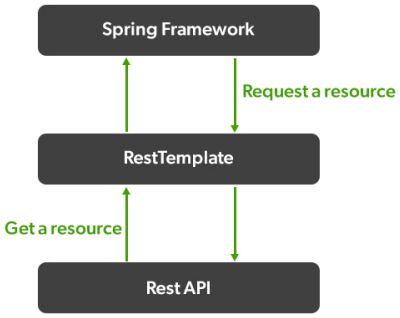
```
package com.masai.controller;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.test.context.TestPropertySource;
import com.masai.model.Employee;

@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
@TestPropertySource(locations = "/application-test.properties")
public class EmployeeControllerTest {

  @Autowired
  private TestRestTemplate testRestTemplate;


  @Test
  @DisplayName("Employee can be created")
  public void testRegisterEmployeeHandler_whenValidDetailsProvided_returnRegisteredEmployee() throws Exception {

    //Arrange
    Employee employee = new Employee();
    employee.setName("Vikash");
    employee.setAddress("delhi");
    employee.setSalary(90000);

    HttpHeaders headers= new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);

    HttpEntity<Employee> entity= new HttpEntity<Employee>(employee,headers);

    //Act
    ResponseEntity<Employee> re= testRestTemplate.postForEntity("/employees", entity, Employee.class);

    Employee createdEmployee= re.getBody();


    //Assert
    assertEquals(HttpStatus.CREATED, re.getStatusCode(), "Created Status code is not 201");
    assertNotNull(createdEmployee.getEmployeeId(), "Registed Employee should have an Id");


  }
}
```

Calling the REST API using Java:

RestTemplate class:

To interact with REST, the client needs to create a client instance and request object, execute the request, interpret the response, map the response to domain objects, and also handle the exceptions. It is common for the Spring framework to both create an API and consume internal or external application's APIs.

Spring provides a convenient way to consume REST APIs – through the **RestTemplate** class.



RestTemplate is a synchronous REST client provided by the Spring Framework.

Some of the methods of RestTemplate class are:

Operation	Method	Action Performed
GET	getForEntity() getForObject()	Sends an HTTP GET request, returning a ResponseEntity containing an object mapped from the response body. Sends an HTTP GET request, returning an object mapped from a response body
POST	postForEntity() postForObject()	POSTs data to a URL, returning a ResponseEntity containing an object mapped from the response body. POSTs data to a URL, returning an object mapped from the response body.
PUT	put()	PUTs resource data to the specified URL.
DELETE	delete()	Performs an HTTP DELETE request on a resource at a specified URL.
PATCH	patchForObject()	Sends an HTTP PATCH request, returning the resulting object mapped from the response body.
ANY	exchange()	Executes a specified HTTP method against a URL, returning a ResponseEntity containing an object.

Consuming REST API using a Maven based Java Application:

- Consuming the above Employee Management application in Spring boot test.

- Create a Maven project and add the following dependency:
  - Spring-webmvc
  - Jackson-databind

```
<!-- https://mvnrepository.com/artifact/org.springframework/spring-webmvc -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>6.0.9</version>
</dependency>



<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.15.2</version>
</dependency>
```

AppConfig.java:

```
package com.masai;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.masai")
public class AppConfig {

}
```

EmployeeBean.java:

```
package com.masai;

public class EmployeeBean {

  private Integer employeeId;
  private String name;
  private String address;
  private Integer salary;

//getters and setters

}
```

RestClientApp.java:

```
package com.masai;

import java.util.Set;

import org.springframework.http.HttpHeaders;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class RestClientApp {

  public void callingApi() {

    RestTemplate rt= new RestTemplate();
```

```
    //String result= rt.getForObject("http://localhost:8080/employees/10", String.class);

        EmployeeBean emp= new EmployeeBean();
        emp.setName("Rakesh");
        emp.setAddress("Mumbai");
        emp.setSalary(89000);

        ResponseEntity<EmployeeBean> re= rt.postForEntity("http://localhost:8080/employees", emp, EmployeeBean.class);
        //here automatically the matching fileds will be populated

        //to get the actual response data (response body)
        EmployeeBean registerdEmployee= re.getBody();

        System.out.println(registerdEmployee);

        System.out.println("the status code is "+re.getStatusCode());

        HttpHeaders hh= re.getHeaders();

        Set set= hh.entrySet();

        System.out.println("Response Headers are :");
        System.out.println(set);

  }

}
```

Demo.java:

```
package com.masai;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Demo {

  public static void main(String[] args) {

    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

    RestClientApp restClientApp= ctx.getBean("a", RestClientApp.class);

    restClientApp.callingApi();
  }

}
```

Passing Headers details with the HTTP request:

```
        EmployeeBean emp= new EmployeeBean();
        emp.setName("Rakesh");
        emp.setAddress("Mumbai");
        emp.setSalary(89000);

    RestTemplate restTemp=new RestTemplate();

    HttpHeaders headers = new HttpHeaders();

    headers.add("jwt", "sdjfbskjdfhkjaldsjfklasjd");
    headers.add("message", "Welcome to Spring Rest API");
```

```
HttpEntity<EmployeeBean> httpEntity = new HttpEntity<>(emp, headers);

ResponseEntity<EmployeeBean> re= restTemp.exchange(uri, HttpMethod.POST, httpEnitity, EmployeeBean.class);

EmployeeBean createdEmployee= re.getBody();
```

Calling Protected API using RestTemplate:

Example1: Using normal maven application:

```
Customer.java:
--------------
package com.masai;

public class Customer {

  private Integer custId;
  private String name;
  private String address;
  private String email;
  private String password;
  private String role;


//getters and setters
//toString method

}
```

RestClientBean.java:

```
RestClientBean.java:
--------------------

package com.masai;

import java.util.Base64;
import java.util.List;

import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.util.MultiValueMap;
import org.springframework.web.client.RestTemplate;

@Component
public class RestClientBean {

  public void callingApi() {

    RestTemplate rt= new RestTemplate();

    String authStr = "ram@gmail.com:1234";
    String base64Creds = Base64.getEncoder().encodeToString(authStr.getBytes());
```

```
    HttpHeaders headers = new HttpHeaders();
    headers.add("Authorization", "Basic " + base64Creds);

    // shortcut approach
    // headers.setBasicAuth("ram@gmail.com", "1234");

    HttpEntity<String> request = new HttpEntity<>(headers);



    ResponseEntity<String> response = rt.exchange("http://localhost:8080/signIn", HttpMethod.GET, request,
        String.class);

    //getting the body
    String result = response.getBody();
    System.out.println(result);


    //getting the token
    String token = response.getHeaders().getFirst("Authorization");
    System.out.println(token);


    //creating another headers object to call another api
    HttpHeaders headers2 = new HttpHeaders();

    //attaching the jwt token to the another protected api call
    headers2.add("Authorization", "Bearer " + token);

    // shortcut approach
    headers2.setBearerAuth(token);

    HttpEntity<String> he = new HttpEntity<>(headers2);

    // ResponseEntity<List> re2= rt.exchange("http://localhost:8080/customers",HttpMethod.GET ,he,List.class);

    ResponseEntity<List<Customer>> re2 = rt
                      .exchange("http://localhost:8080/customers",
                          HttpMethod.GET, he,
                          new ParameterizedTypeReference<List<Customer>>() {});

    System.out.println(re2.getBody());

  }

}
```

Example 2: Using Spring Boot application

- Create a Simple Spring Boot application (No need to add Spring Security related dependencies)
- Create A LoginBean to hold the user credentials:

  LoginBean.java:

```
package com.masai.model;
public class LoginBean{

  private String username;
  private String password;

//getter and setters
```

```
}
```

CustomerClientController.java:

```
package com.masai.controller;
@RestController
public class CustomerClientController {

@PostMapping("/signIn")
  public List<Customer> signInCustomer(@RequestBody LoginBean bean) {

    RestTemplateBuilder builder = new RestTemplateBuilder();
    RestTemplate rt= builder.basicAuthentication(bean.getUsername(),bean.getPassword()).build();

    String url = "http://localhost:8080/signIn";

    ResponseEntity<String> re= rt.getForEntity(url, String.class);

    System.out.println("Loggin Message is :"+re.getBody());

    String token= re.getHeaders().getFirst("Authorization");

    System.out.println(token);

    String url2="http://localhost:8080/customers";


    HttpHeaders headers = new HttpHeaders();

    headers.add("Authorization", "Bearer "+token);


    HttpEntity<String> he = new HttpEntity<>(headers);


    //ResponseEntity<List> re2= rt2.exchange(url2,HttpMethod.GET ,he,List.class);

ResponseEntity<List<Customer>> re2= rt.exchange(url2,HttpMethod.GET ,he,new ParameterizedTypeReference<List<Customer>>() {});

    System.out.println(re2.getBody());

    return re2.getBody();

}
```

Call this API using :

```
Method: POST
URL: http://localhost:8888/signIn

Body: {
        "username": "ram@gmail.com",
        "password": "1234"
     }
```