Introduction to Rest TEMPLATE

What is a Rest Template?

We can have different services in a single application and use Rest Templates to provide **intercommunication** between these services. The Rest Template is the central class within the Spring framework for executing synchronous HTTP requests on the client side.

Use of Rest Template

- Creating a URL object and opening the connection
- Configuring the HTTP request
- Executing the HTTP request
- Interpretation of the HTTP response
- Converting the HTTP response into a Java object
- Exception handling

Example:

Let's say our application has two service module and their respective models:

- **Hotel Module**: Provides services for creating, adding, deleting and updating hotels and their ratings.
- **Rating Module**: Provides create, add, delete and update services for key, value pair of <HotelId, Hotel Rating>.

Task: Let's say we have a service that returns a hotel by id, and we want an updated rating from the Rating Module.
Solution: With RestTemplate, we create a service that calls the Rating Module's service for updated rating.

```
//HotelService.java
public Hotel getHotelById(String id) {
    if (ObjectUtils.isEmpty(hotelMap.get(id))) {
 throw new HotelNotFoundException("Hotel not found for id: " + id);
}
Hotel hotel hotelMap.get(id);
//rest service to fetch the rating by id
return hotel;
}
```

When using RestTemplate, all these things happen in the background, and the developer doesn't have to bother with it. Rest Template makes it convenient for services to communicate with each other.

Rest Template Builder

RestTemplate Builder helps us to create a RestTemplateBuilder Object. The rest template object is created with the help of RestTemplateBuilder (a class), which contains methods of the rest template. It also provides convenient methods to register converters, error handlers, and UriTemplateHandlers.

Example:

This template is created in the Hotel module, and this service is used to call rating services in the Rating Module.

```
//RatingServiceCommunicator.java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
@Service
public class RatingServiceCommunicator {

    private final RestTemplate restTemplate;

    @Autowired
    public RatingServiceCommunicator (RestTemplateBuilder
restTemplateBuilder) {
    this.rest Template=restTemplateBuilder.build();
}
```

In the code above; the RatingServiceCommunicator class uses RestTemplate to communicate with the rating service in the Rating module. The RestTemplateBuilder is autowired into the class constructor, allowing Spring to provide an instance of RestTemplateBuilder.

The restTemplateBuilder.build() method is called to create a RestTemplate object. This method creates a new instance of RestTemplate with any customisations applied using the RestTemplateBuilder instance.

Once the RestTemplate object is created, it can be used within the RatingServiceCommunicator class to make HTTP requests to the rating service.

Valuable Methods for Rest Template

Methods primarily used in Rest Template are:

- `getForEntity()`
- `getForObject()`
- `postForObject()`
- `exchange()` - Common method for all GET, PUT, POST, and Delete operations.

For additional information on the methods of RestTemplate, please refer to the Javadoc.

Method 1: getForEntity()

The getForEntity() method retrieves resources from the HTTP Request. We have to pass the URL and response type, and it returns the response as ResponseEntity, using which we can get the response status code, response body, and much more helpful information. We can also add other variables to request if required.

Example:

We will use the rest template object created above for the getForEntity method.
Two parameters in the request:

- URL of the HTTP Request
- Return Type - which will store the response in the type mentioned.

```
//RatingServiceCommunicator.java
public Long getRating (String id){
String url="http://localhost:8081/rating/id/";
ResponseEntity<Long> response = restTemplate.getForEntity(url+id,
Long.class);
return response.getBody();

}
```

This response gets stored in ResponseEntity with a mentioned type (e.g., Long). ResponseEntity encapsulates the HTTP response status code, the HTTP headers, and the body that has already been converted into a Java object. It will give us methods like getStatusCode(), getHeaders(), and getBody().

Method 2: getForObject()

This method is similar to the getForEntity() method, as it requires the URL and the return Type in parameters, the only difference is, it will directly return only the value of the response body, so it will not contain methods like getHeaders() and getResponseBody().

Example:

If you only require the response value, use the getForObject() method.

```
RatingServiceCommunicator.java
public long getRating (String id)
String url="http://localhost:8081/rating/id/";
//ResponseEntity<Long> response= restTemplate.getForEntity(url+id,
Long.class);
Long ratingResponse = restTemplate.getForObject(url+id, Long.class);
return ratingResponse;
}
```

Method 3: postForObject()

This method creates a new resource using the HTTP POST method. It takes the URL, request value, and response type as input and returns an entity of the object created.

Example:

Task: Let's say we add a new hotel, and we also want to call a method in the “Rating Module” to add a rating for this new hotel.

Solution: The Hotel Module will use postForObject() in its service to call the service in the image below, which is present in the Rating Module. This service adds a <HotelId, RatingValue>.pair into the Rating Model.

```
RatingService.java
@PostMapping(path = "/add")
    public void addRating(@RequestBody Map<String, Double>
hotelRatingMap) {
        ratingService.addRating(hotelRatingMap);
    }
```

Simply, we will create an addRating method in hotel service, which will take <HotelId, RatingValue> map and pass it as a parameter postForObject, which will save Object.class as a response, as we are not returning anything.

```
RatingServiceCommunicator.java
public void addRating (Map<String, Long> ratingsMap) {

String url="http://localhost:8081/rating/add";
restTemplate.postForObject(url, ratingsMap, Object.class);

}
```

Create hotel Service will call our “addRating” service in RatingServiceCommunicator.java.

```
HotelService.java
public void createHotel (Hotel hotel) {

Map<String, Long> ratingsMap = new HashMap<>();

hotelList.add(hotel); hotelMap.put(hotel.getId(), hotel);
ratingsMap.put(hotel.getId(), hotel.getRating());
ratingServiceCommunicator.addRating(ratingsMap);
}
```

Method 4: exchange()

Exchange is the method through which we can call all HTTP requests, so we can use this method in place of calling specific HTTP methods like getForObject(), postForObject(), etc. The exchange method can be used for HTTP DELETE, GET, HEAD, OPTIONS, PATCH, POST, and PUT methods.

The exchange is a method to fetch, update, create new and delete older data from our application. The exchange method returns ResponseEntity, using which we can get the response status, body, and headers.

HTTP Entity

HttpEntity is a parameter we pass in the exchange method.

- It contains your request, wrapped in an HTTP entity. It will contain both headers and a request body.
- We can create this entity by passing both body and header in the input.

Parameters of exchange() method

- URL
- HTTP Request Method
- Request Entity
- Response Type
- Other URI Variables

- ResponseEntity is the return type of this method.

```
ResponseEntity<T>   exchange(String url, HttpMethod method, HttpEntity<?>
requestEntity, Class<T> responseType, Map<String,?> uriVariables)
```

```
RatingServiceCommunicator.java
public void addRating(Map<String, Long> ratingsMap) {
            String url ="http://localhost:8081/rating/add";
            //restTemplate.postForObject(url, ratingsMap, Object.class);
            HttpEntity<Map<String, Long>> requestEntity = new
HttpEntity<>(ratingsMap);

restTemplate.exchange(url,HttpMethod.POST,requestEntity,Object.class);
      }
```

exchange(): PUT

To use the exchange() to update data, we need to use the HTTP method as **HttpMethod.PUT**. For request entities, we can use HttpEntity.

Example:

Task: If we update our rating in hotel service, we also want the ratings to be updated in the Rating module.

Solution: A put request is used in the rating service of the Hotel Module to communicate with the Rating module.

```
RatingService.java
@PutMapping (path = "/update")
    public void updateRating(@RequestBody Map<String, Double> hotelRatingMap) {
        ratingService.updateRating(hotelRatingMap);
    }
```

This method will call the updateRating method in the Rating Module. For this, we will use the **HttpMethod.PUT** request.

```
RatingServiceCommunicator.java
public void updateRating(Map<String, Long> ratingsMap)
      {
            String url ="http://localhost:8081/rating/update";

            HttpEntity<Map<String, Long>> requestEntity = new
HttpEntity<>(ratingsMap);
      restTemplate.exchange(url,HttpMethod.PUT,requestEntity,Object.class);
      }
```

UpdateHotel will call the above method to update the rating.

```
HotelService.java
public void updateHotel(Hotel updatedHotel) {
            //1. Get the previous data of the hotel
            //2. remove this old data from list
            //3. Add the updated data to the list.

            Hotel existingHotel= getHotelById(updatedHotel.getId());

            hotelList.remove(existingHotel);
            hotelList.add(updatedHotel);



            //4. update the previous data with new data.
            //5. Update the map with new data.

            hotelMap.put(updatedHotel.getId(), updatedHotel);

            Map<String,Long> updatedRating = new HashMap<>();
            updatedRating.put(updatedHotel.getId(), updatedHotel.getRating());
            ratingServiceCommunicator.updateRating(updatedRating);

      }
```

exchange(): DELETE

To use the exchange() to delete data, we need to use the HTTP method as **HttpMethod.DELETE** No request entity is required for this method, so it will pass null for the same.

```
RatingService.java
@DeleteMapping(path = "/id/{id}")
    public Double getRatingById(@PathVariable String id) {
        return ratingService.deleteRating(id);
    }
```

```
RatingServiceCommunicator.java
public void deleteRating(String id)
      {
            String url ="http://localhost:8081/rating/remove/id/";
            restTemplate.exchange(url+id,HttpMethod.DELETE,null,Object.class);
      }
```

```
HotelService.java
public void deleteHotelById(String id) {
            Hotel hotel = getHotelById(id);
            hotelList.remove(hotel);
            hotelMap.remove(id);
            ratingServiceCommunicator.deleteRating(id);
      }
```

By using the exchange() method, you can handle different types of HTTP requests (DELETE, GET, HEAD, OPTIONS, PATCH, POST, and PUT) in a flexible manner. It lets you specify the HTTP method, request entity (body and headers), response type, and other parameters.

In the provided examples, the response type is set to Object.class, which means the expected response can be of any type. However, you can modify the Object.class to match the specific type of response you expect. For example, if you expect the response to be a String, you can set the response type as String.class.

Overall, the exchange() method in RestTemplate provides versatility and customisation options for making HTTP requests in your application, enabling you to handle various scenarios and interact with different APIs effectively.

HTTP Error Handling

Error handling is the process comprised of anticipation, detection, and resolution of application errors. The tasks of the Error Handling process are to detect each error, report it to the user, and then make some recovery strategies and implement them to handle the error. We will use try-and-catch blocks to handle exceptions in our code.

When the hotel service communicates with the rating service, there are changes or errors, like the rating service getting modified or the URL passed in the hotel service being incorrect. So, we have to handle these kinds of errors.

By default, the RestTemplate will throw one of these exceptions in the case of an HTTP error:

- HttpClientErrorException: Client errors occur when the client side of a communication process encounters issues while interacting with a server. These errors are typically indicated by HTTP status codes in the **4xx** range. Here are some common client error status codes::
  - 400 Bad Request: This status code indicates that the server could not understand the client's request due to malformed syntax or invalid parameters. It often occurs when the client sends a request the server cannot process.
  - 401 Unauthorized: When a client receives a 401 status code, the requested resource requires authentication. The client must provide valid credentials (e.g., username and password) to access the resource.
  - 403 Forbidden: The server understands the client's request, but the client cannot access the requested resource. This status code differs from 401, implying the client lacks the necessary permissions to access the resource even after authentication.
  - 404 Not Found: This status code indicates that the server could not find the requested resource. It commonly occurs when a client tries to access a URL or endpoint that doesn't exist on the server.

- HttpServerErrorException: Server errors occur when there are issues on the server side of a communication process. These errors are typically indicated by HTTP status codes in the **5xx** range. Here are some common server error status codes:
-
  - 500 Internal Server Error: This is a generic error message indicating that an unexpected condition occurred on the server, preventing it from fulfilling the client's request. It signifies an error within the server's infrastructure or application code.
  - 504 Gateway Timeout: This status code indicates that the server, acting as a gateway or proxy, did not receive a timely response from an upstream server to complete the request. It often occurs when there is a delay or timeout between different servers.

- UnknownHttpStatusCodeException: The UnknownHttpStatusCodeException is an exception by RestTemplate when it encounters an unrecognized or unknown HTTP status code during communication with a server. This exception provides a way to

handle scenarios where the server returns custom or non-standard status codes. By catching the UnknownHttpStatusCodeException, developers can implement specific error-handling logic and access details about the unknown status code, such as the numeric value and associated message. This enables better control and informed handling of unknown status codes in the application.

All of these exceptions are extensions of ***RestClientResponseException.***

```
RatingServiceCommunicator.java
public void deleteRating(String id)
{
       String url ="http://localhost:8081/rating/remove/id/";
     try {
restTemplate.exchange(url+id,HttpMethod.DELETE,null, Object.class);
              }
              catch(HttpClientErrorException e)
              {
throw new
HttpRatingServiceNotFound(HttpStatusCode.valueOf(HttpStatus.NOT_FOUND.value()));
              }
       }
```

We have created our custom exception, which will be thrown if we catch a Client Exception in the delete Rating Service.

```
HttpRatingServiceNotFound.java
@ResponseStatus(HttpStatus.NOT_FOUND)
public class HotelNotFoundException extends RuntimeException{

       private static final long serialVersionUID = 1L;

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

}
```

In the provided code example, error handling is implemented in the deleteRating() method of the RatingServiceCommunicator class. It uses try-catch blocks to handle exceptions that may occur during the HTTP request to the Rating service. Specifically, it catches HttpClientErrorException to handle client-side errors.

This error handling aims to detect client-side errors during communication with the Rating service, report them, and provide a meaningful response. By catching and throwing specific exceptions, the code ensures that appropriate error messages and status codes are returned to the caller.

Overall, this error-handling approach enhances the robustness of the code by handling potential errors during communication with external services, allowing for better user experience and error resolution strategies.

Message Convertors

In Spring, message converters are crucial in converting Java objects to JSON, XML, and other formats during HTTP requests and responses. The HttpMessageConverter interface is used by Spring to perform these conversions. When an HTTP request is made to a Spring application, the message converters come into play to determine the format of the request payload. They convert the request body from its raw form (JSON or XML) into the corresponding Java object.

Similarly, when a Spring application generates an HTTP response, the message converters handle the conversion of the Java object into the desired format specified by the client's request. This allows the response body to be sent in the appropriate format (e.g., JSON or XML) based on the requested content type. Spring provides a variety of built-in message converters that cover common scenarios. These converters can handle data types, such as byte arrays, strings, resources (files), and XML sources. They automatically handle the serialisation and deserialisation of objects based on the content type negotiation and request/response headers.

Types of Message Convertors

By default, the following HttpMessageConverters are loaded by Spring. There are several types of message converters available in Spring by default. Here are some commonly used ones:

- **ByteArrayHttpMessageConverter:** This converter handles byte arrays, allowing the conversion of entire Java objects to JSON or other formats and vice versa.

- **StringHttpMessageConverter:** As the name suggests, this converter deals with string conversion. It is responsible for reading and writing strings. When returning a string or integer from a request, this converter deserialises it into the correct form.

- **ResourceHttpMessageConverter:** This converter handles resources and supports byte-range requests. It can read and write resources, making it useful for scenarios involving files or binary data.

- **SourceHttpMessageConverter:** The SourceHttpMessageConverter is responsible for handling Source objects. It can read and write Source objects, typically used in XML-based communication.

- **FormHttpMessageConverter:** This converter specialises in handling HTML forms. It can read and write data from "normal" HTML forms, allowing for seamless interaction with form-based requests.

These message converters are automatically loaded by Spring, and they provide convenient ways to convert Java objects to the desired format during HTTP communication. Spring's message converters greatly simplify handling different data formats and ensure seamless integration with the specified content types.

For additional information on the Message Convertors of RestTemplate, please refer to the MessageConvertorDoc.

Instructor Codes

- Hotel Application

References

1. RestTemplate
2. getForEntity()
3. getForObject()
4. postForObject()
5. exchange()
6. Message Converters