Spring Start Here

Chapter-10
Implementing REST services

Upcode Software
Engineer Team

-2024-

CONTENT

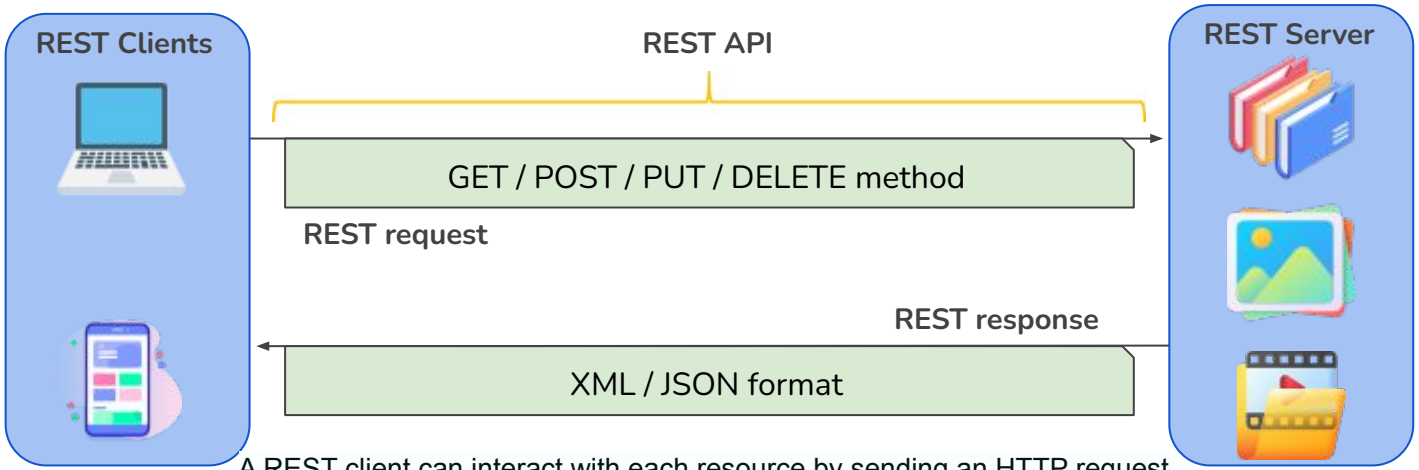
1. **Introduction**
   - 1.1 REST
   - 1.2 RESTful
   - 1.3 HTTP Methods of REST Web Services
   - 1.4 Structure of REST
2. **Implementing REST services**
3. **Using REST services to exchange data between apps**
4. **Implementing a REST endpoint**
5. **Managing the HTTP response**
   - 5.1 Sending objects as a response body
   - 5.2 Setting the response status and headers
   - 5.3 Managing exceptions at the endpoint level
6. **Using a request body to get data from the client**
7. **Conclusion**
8. **Resources**
9. **References**

1. Introduction

Before going into this presentation, let's talk briefly about **REST, RESTful**, their differences, structure of **REST** and then we plan to dive deeper into implementing **REST** services.

1.1 What is REST ?

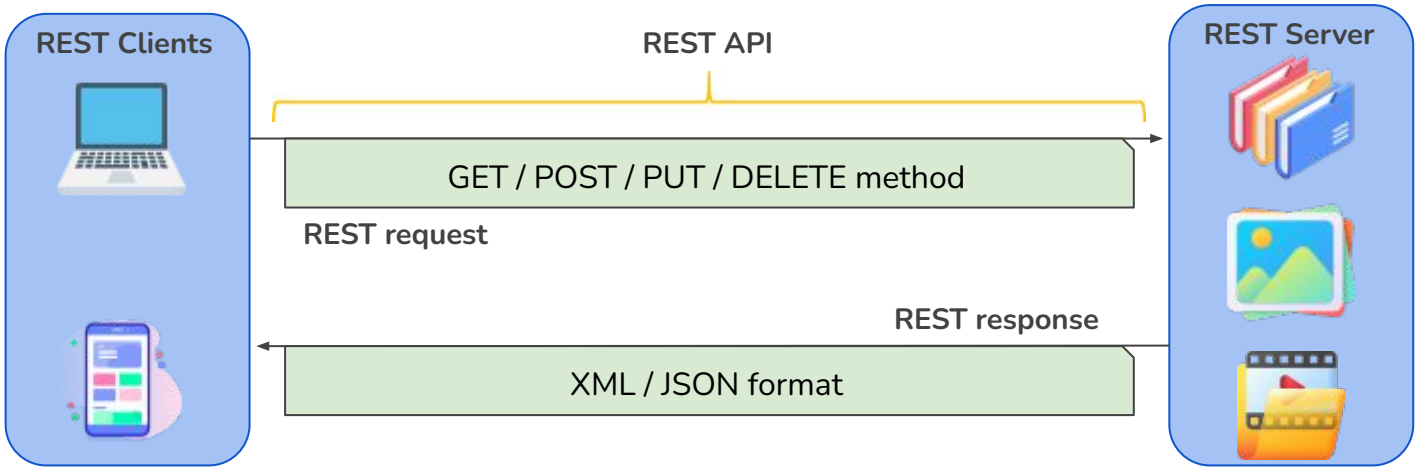
REST is short for Representational State Transfer, an architectural style for building web services that interact via an HTTP protocol. REST offers access to functionality the server exposes through endpoints a client can call.



A REST client can interact with each resource by sending an HTTP request.

1.2 What is RESTful ?

RESTful refers to a web service that follows the principles and constraints of REST architecture. **RESTful** services utilize the standard web protocols and methods (mainly HTTP) to interact with resources in a stateless, scalable, and easy-to-understand manner.



1.3 HTTP Methods of REST Web Services

REST uses URIs(Uniform Resource Identifiers) to identify resources and use standard HTTP methods (GET, POST, PUT, DELETE, PATCH and OPTIONS) to handle data.

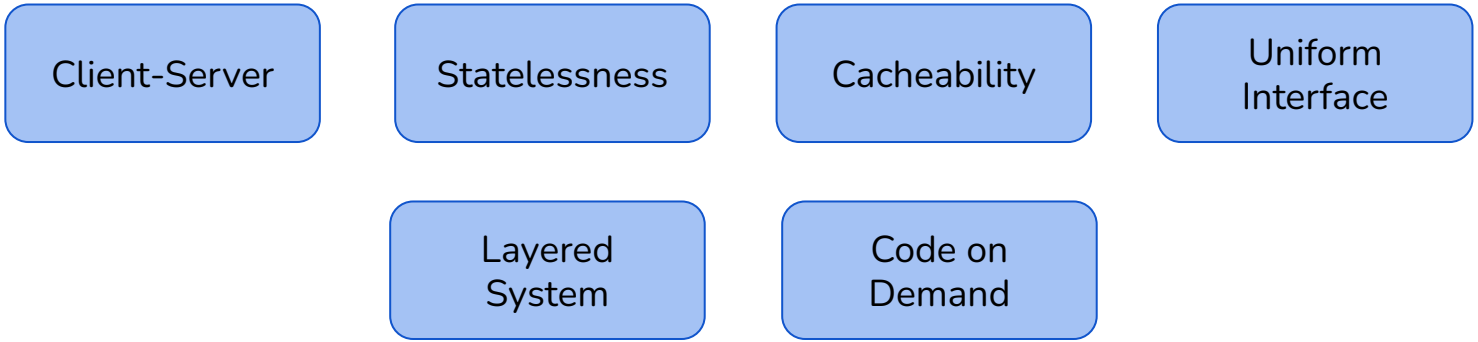
POST	Create a new resource
GET	Retrieve the data
PUT	Update an existing resource or create a resource if it doesn't exist.
DELETE	Retrieve the data
PATCH	Partially update a resource. If the data doesn't exist, server return 404.
OPTIONS	To get information about the possible communication options
HEAD	It is almost identical to GET, but without the response body options

```
POST /api/books
Content-Type: application/json

{
    "title": "New Book",
    "author": "John Doe",
    "publishedDate": "2024-05-21"
}
```

1.4 Structure of REST (1/4)

REST promotes scalability, efficiency, simplicity, and modifiability through a set of constraints. These constraints help in creating systems that are easy to understand and maintain.

Key Constraints of REST

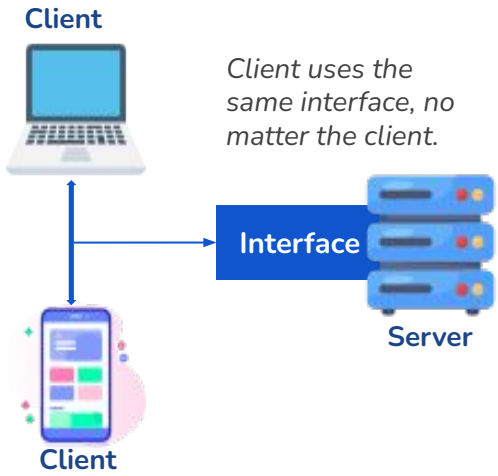


1.2 Structure of REST (2/4)

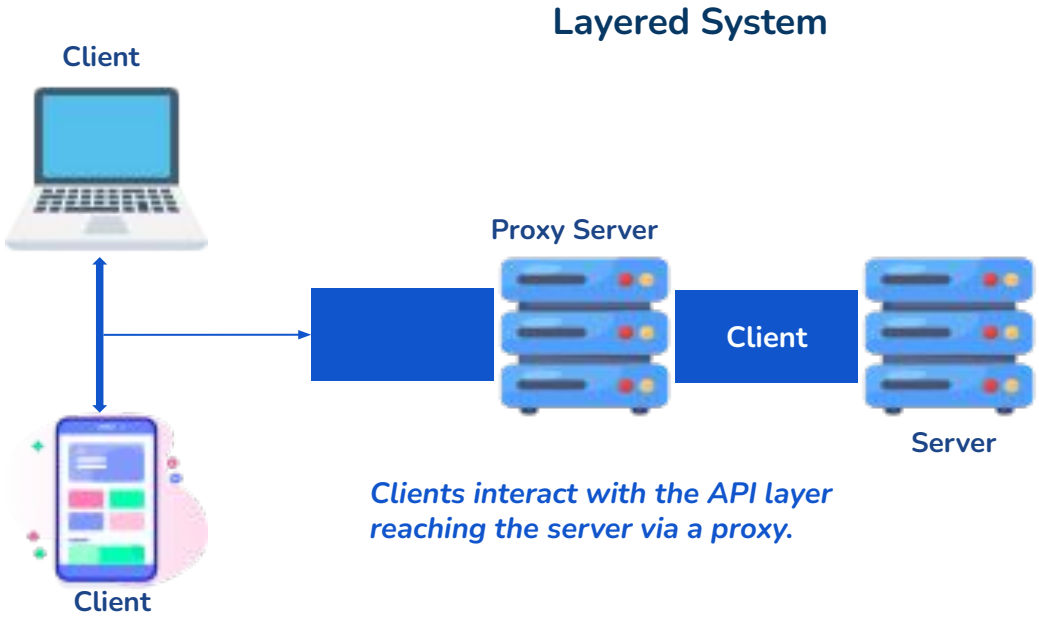
Uniform Interface

The **Uniform Interface** is one of the core principles of the **REST** architecture. It simplifies and decouples the architecture, enabling each part of the system to evolve independently. The uniform interface consists of 4 constraints that define the interaction between clients and servers.

Identification of resources	Resources as URIs http://auto.com/cars/123
Manipulation of resources through representations	JSON , XML ...
Self-descriptive language	HTTP GET / POST / PUT / DELETE media types, cacheability...
Hypermedia as the engine of application state	Hypermedia APIs HAL, JSON-LD, Siren...



1.4 Structure of REST (3/4)



- The **RESTful** system has a layered structure in which each layer works independently and interacts only with the layers directly connected to it.
- When calling a server, a client doesn't know whether there are any intermediaries along the way.

1.4 Structure of REST (4/4)

Client-Server: This architectural style separates the client from the server, enabling independent evolution and scalability of both components.

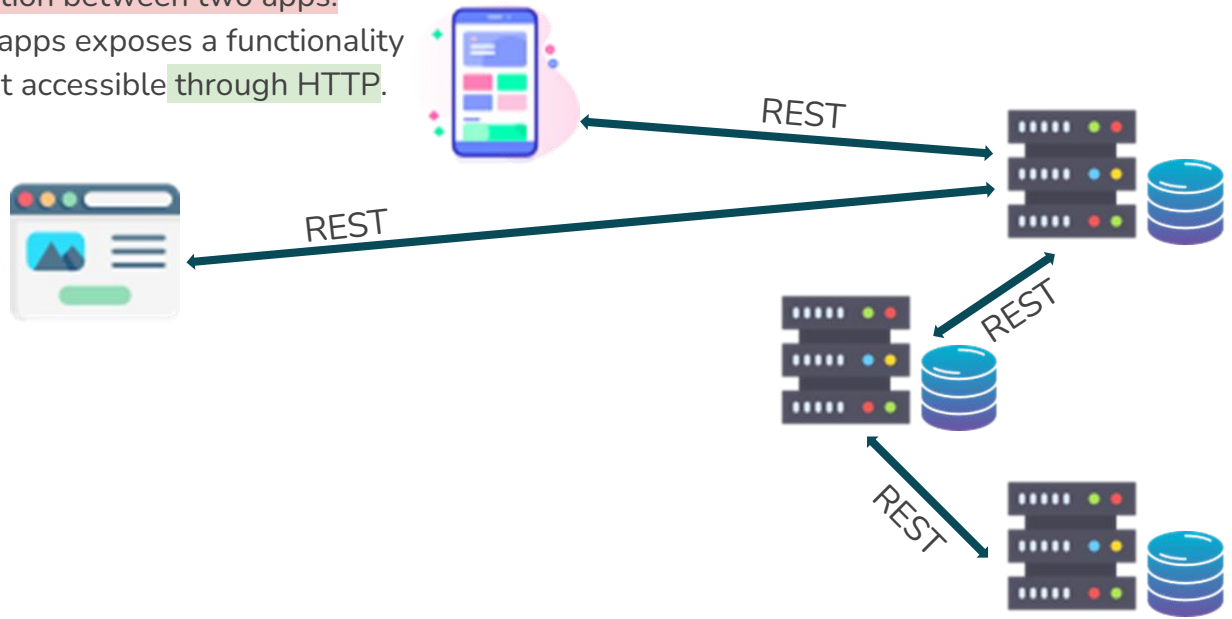
Statelessness: Each request from a client to the server must contain all the information necessary to understand the request, meaning the server should not rely on any previous interactions.

Cacheability: Responses from the server can be labeled as cacheable or non-cacheable, which allows clients to reuse responses, reducing latency and improving efficiency.

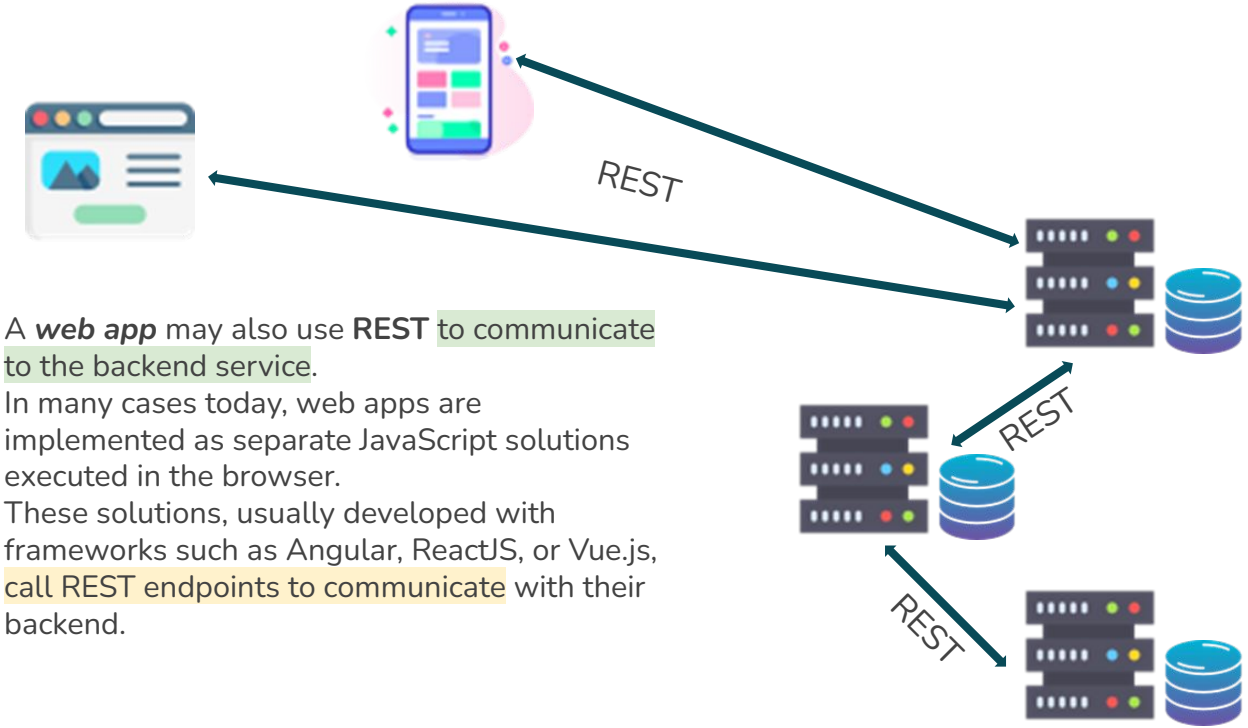
Code on Demand: Servers can temporarily extend or customize the functionality of a client by transferring logic to it in the form of executable code, such as JavaScript.

2. Implementing REST services (1/6)

● A **REST** *endpoint* is a way to implement communication between two apps.
One of the apps exposes a functionality by making it accessible through HTTP.

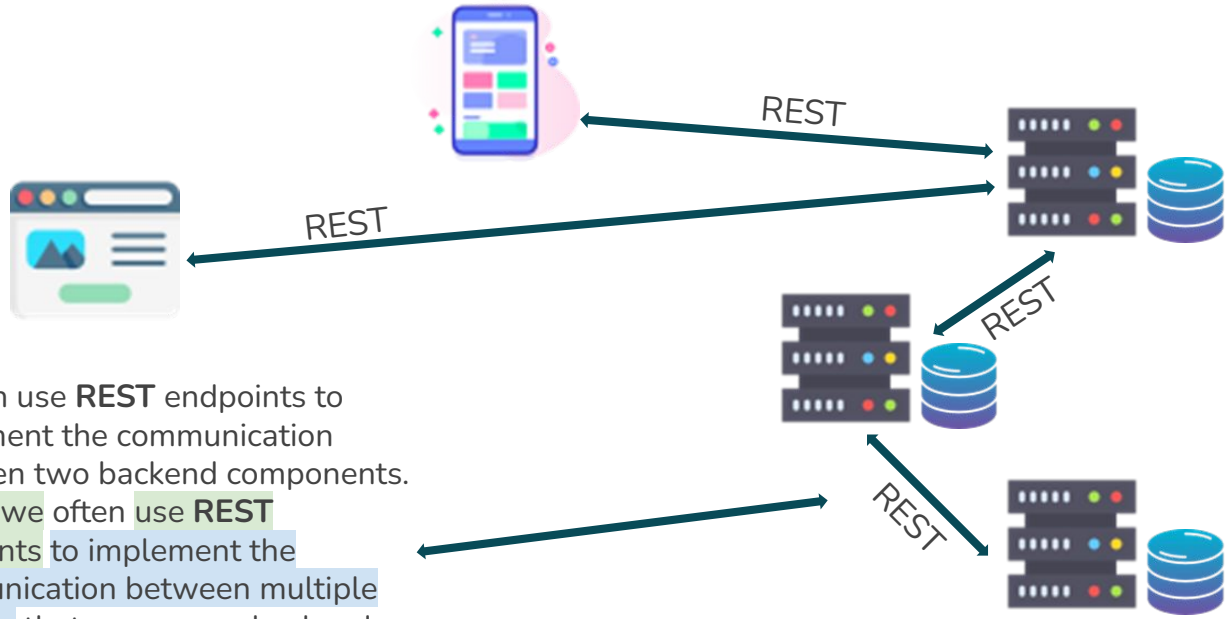


2. Implementing REST services (2/6)



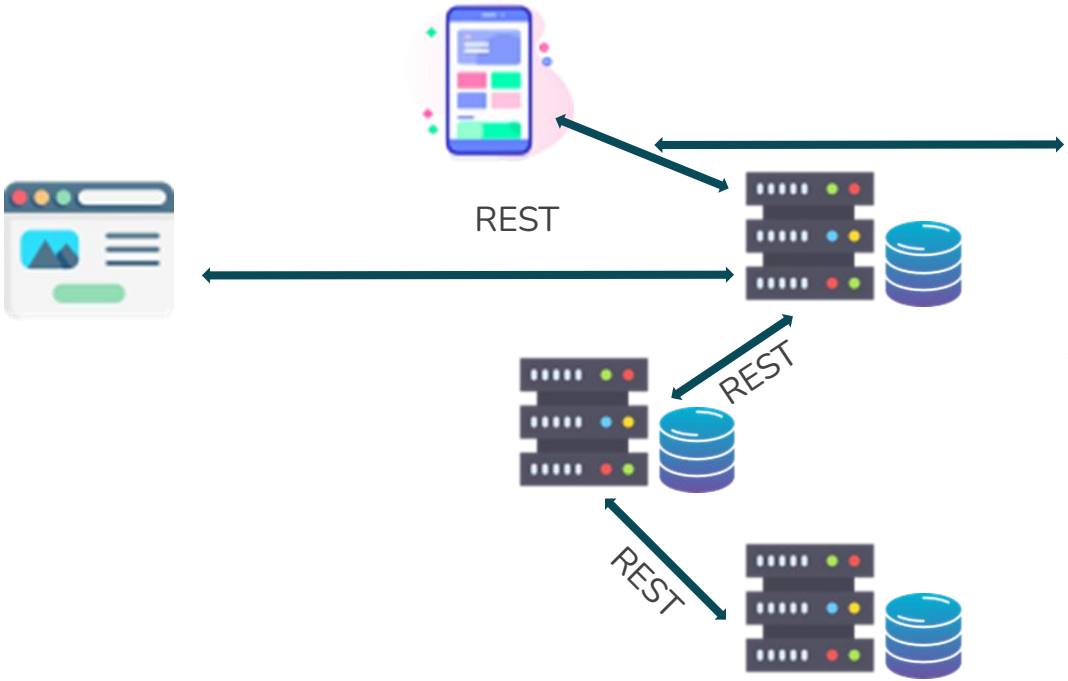
- A ***web app*** may also use **REST** to communicate to the backend service.
- In many cases today, web apps are implemented as separate JavaScript solutions executed in the browser.
- These solutions, usually developed with frameworks such as Angular, ReactJS, or Vue.js, call REST endpoints to communicate with their backend.

2. Implementing REST services (3/6)



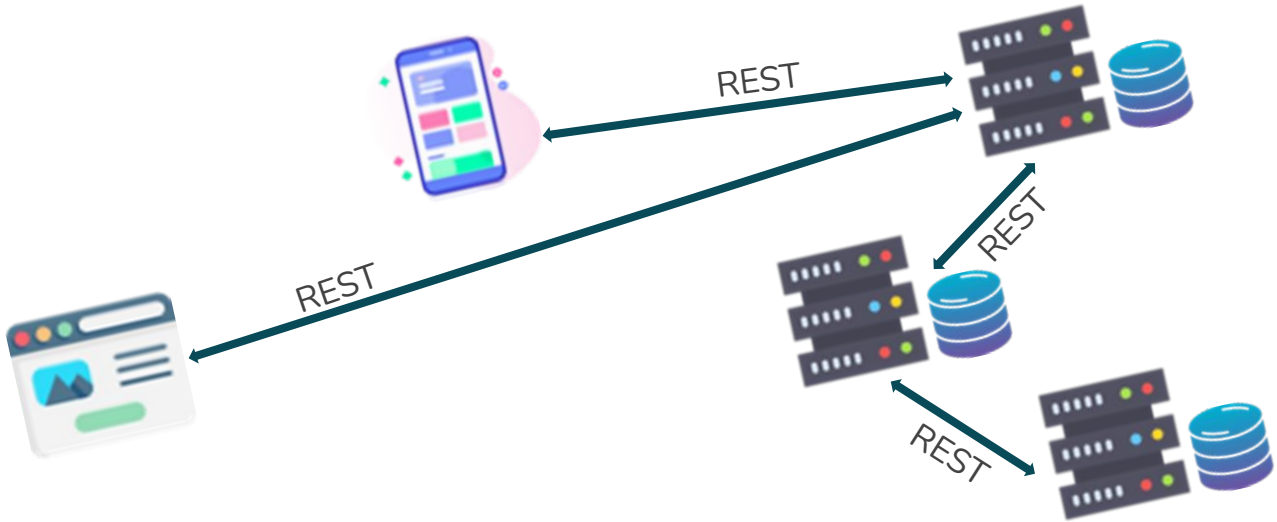
We can use **REST** endpoints to implement the communication between two backend components. Today, we often use **REST** endpoints to implement the communication between multiple services that compose a backend solution.

2. Implementing REST services (4/6)



An example of an application that may use a **REST** endpoint is a mobile app communicating with its backend solution. The communication between a mobile app and its backend service may be implemented with **REST** endpoints.

2. Implementing REST services (5/6)

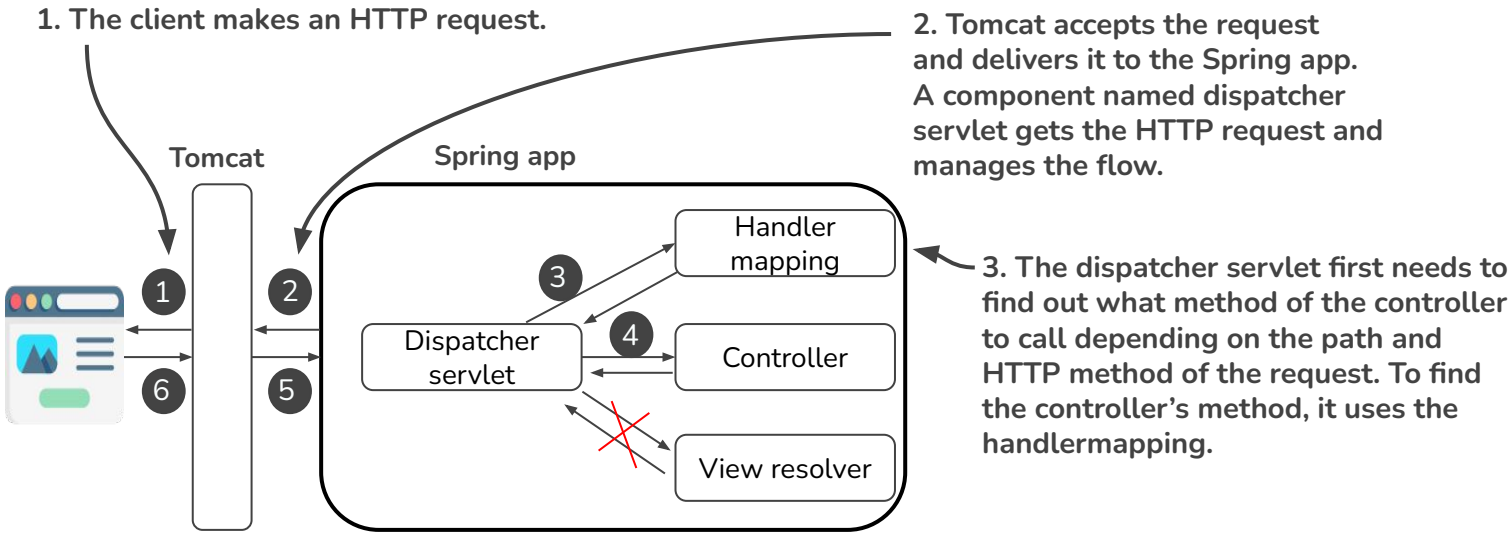


REST services are a communication method between two apps. Today, we can find **REST services** in many places. A web client app or mobile app may call its backend solution through **REST** endpoints, but even backend services might communicate using **REST** web service calls.

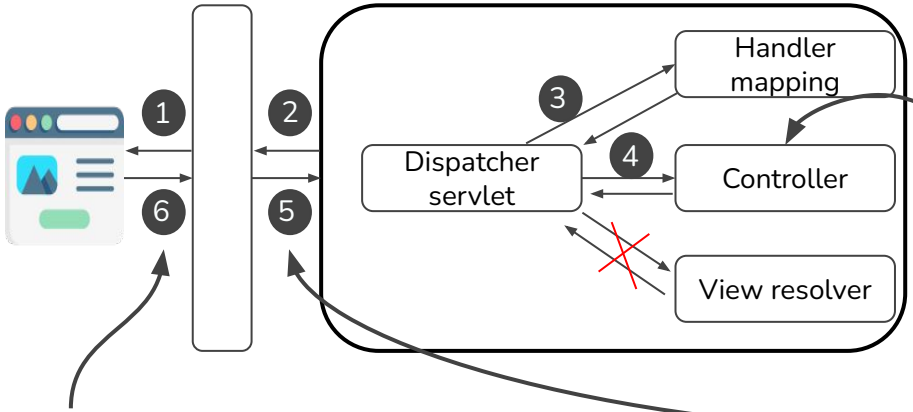
2. Implementing REST services (6/6)

We'll work on several examples to
elaborate on the critical aspects any Spring developer needs to know
when implementing communication between two apps with **REST services.**

3. Using REST services to exchange data (1/3)



3. Using REST services to exchange data (2/3)



4. Once it knows what controller method to call, the dispatcher servlet calls the method.
After execution, the controller method returns the value to be sent to the client in the HTTP response body.

5. Through Tomcat, the HTTP response is returned to the client.

6. The client gets the HTTP response data.

When implementing REST endpoints, the Spring MVC flow changes. The app no longer needs a view resolver because the client needs the data returned by the controller's action directly. Once the controller's action completes, the dispatcher servlet returns the HTTP response without rendering any view.

3. Using REST services to exchange data (3/3)

Before starting with our first example, I'd like to make you aware of some communication issues the REST endpoint might bring:

- If the controller's action takes a long time to complete, the HTTP call to the endpoint might time out and break the communication.
- Sending a large quantity of data in one call (through the HTTP request) might cause the call to time out and break the communication. Sending more than a few megabytes through a REST call usually isn't the right choice.
- Too many concurrent calls on an endpoint exposed by a backend component might put too much pressure on the app and cause it to fail.
- The network supports the HTTP calls, and the network is never 100% reliable. There's always a chance a REST endpoint call might fail because of the network

4. Implementing a REST endpoint (1/10)

In this section, we'll learn to implement REST endpoints with Spring.
The good news is that Spring uses the same Spring MVC mechanism behind REST endpoints, so we already know a big part of how they work from previous presentations of 7 and 8.
Let's start with an example on the next page (**"exercise_1"**).

4. Implementing a REST endpoint (2/10)

Implementing a REST endpoint action in a controller class

```
@Controller
public class HelloController {
  @GetMapping("/hello")
  @ResponseBody
  public String hello() {
    return "Hello!";
  }
}
```

We use the @Controller annotation to mark the class as a Spring MVC controller.

We use the @GetMapping annotation to associate the GET HTTP method and a path with the controller's action.

We use the @ResponseBody annotation to inform the dispatcher servlet that this method doesn't return a view name but the HTTP response directly.

4. Implementing a REST endpoint (3/10)

```java
@Controller
public class HelloController {
  @GetMapping("/hello")
  @ResponseBody
  public String hello() {
    return "Hello!";
  }
}
```

Code, shows us a Controller Class that implements a simple action. As we learned from presentation 7, we annotate the Controller Class with the ***@Controller*** *stereotype annotation. This way, an instance of the class becomes a bean in the Spring context, and Spring MVC knows this is a controller that maps its methods to specific HTTP paths.*
Also, we used the ***@GetMapping*** *annotation to specify the action path and HTTP method.*
The only new thing we find in this listing is the use of the ***@ResponseBody*** *annotation.*
The ***@ResponseBody*** *annotation tells the dispatcher servlet that the controller's action doesn't return a view name but the data sent directly in the HTTP response.*

4. Implementing a REST endpoint (4/10)

```
@Controller
public class HelloController {

  @GetMapping("/hello")
  @ResponseBody
  public String hello() {
    return "Hello!";
  }

  @GetMapping("/ciao")
  @ResponseBody
  public String ciao() {
    return "Ciao!";
  }
}
```

Look what happens if we add more methods to the controller, shown in the following listing. Repeating the **@ResponseBody** annotation on every method becomes annoying.

A best practice is avoiding code duplication. We want to somehow prevent repeating the **@ResponseBody** annotation for each method. To help us with this aspect, Spring offers the **@RestController** annotation, a combination of **@Controller** and **@ResponseBody**.

4. Implementing a REST endpoint (5/10)

Using the @RestController annotation to avoid code duplication

```
@RestController
public class HelloController {

  @GetMapping("/hello")
  public String hello() {
    return "Hello!";
  }

  @GetMapping("/ciao")
  public String ciao() {
    return "Ciao!";
  }
}
```

Instead of repeating the **@ResponseBody** annotation for each method, we replace **@Controller** with **@RestController**.

4. Implementing a REST endpoint (6/10)

So, we implemented couple of endpoints. But how do we validate they work correctly ?

- **Postman** - Offers a nice GUI and is comfortable to use
- **cURL** - A command-line tool useful in cases where you don't have a GUI (e.g., when you connect to a virtual machine via SSH or when you write a batch script)

Lately, in presentation 15, we'll learn next approach for validating that an endpoint behaves as expected by writing an integration test.

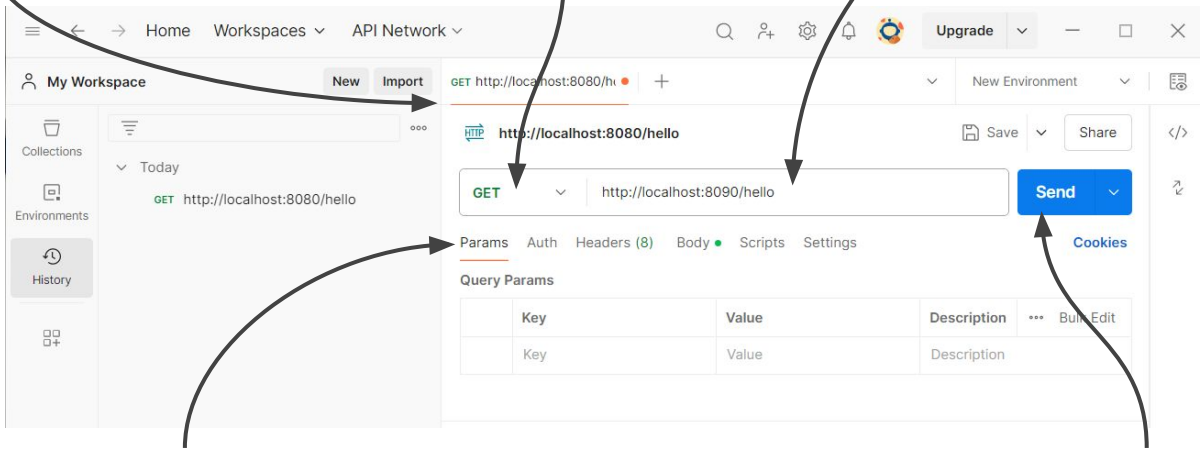
Let's discuss Postman first. We need to install the tool on our system as presented on their official website: **https://www.postman.com/**

4. Implementing a REST endpoint (7/10)

We create a new tab to define an HTTP request.

We select the HTTP method of the HTTP request we want to send.

In the address bar, we write the URI for the HTTP request.

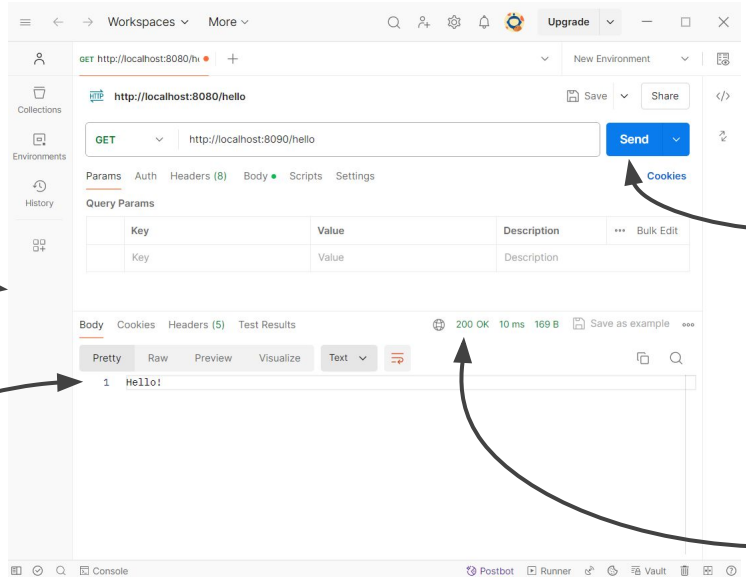


We can use these tabs to define HTTP request parameters, headers, or the body of the request.

We click the Send button to send the HTTP request.

4. Implementing a REST endpoint (8/10)

If we send headers in the HTTP response, Postman displays them in this tab.



Once we click the Send button, Postman sends the HTTP request. When the HTTP request is completed, Postman displays details of the HTTP response it received.

Here, we find the HTTP response body. In our case, the string “Hello!”

Here, we find the HTTP response status code, the execution time, and the amount of transferred data in bytes.

4. Implementing a REST endpoint (9/10)

We'll find *articles* and *books* often use **command-line** tool like in the case of **Postman**, we need first to make sure we install it.

- We install **cURL** according to our operating system as described on the tool's official web page: **https://curl.se/**
- Once we installed and configured, we can use the cURL command to send HTTP requests.

  curl -X GET http://localhost:8080/hello

  Result: Hello!

4. Implementing a REST endpoint (10/10)

Result below:

- If we want to get more details of the HTTP request, we can add the **-v** option to the command, as presented below:

curl -v http://localhost:8080/hello

```
Trying ::1:8080...
* Connected to localhost (::1) port 8080 (#0)
> GET /hello HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.73.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 6
< Date: Mon, 20 May 2024 23:11:02 GMT
<
{ [6 bytes data]
100 6 100 6 0 0 857 0 --:--:-- --:--:-- --:--:--
1000
Hello!
* Connection #0 to host localhost left intact
```

The HTTP response status

The HTTP response body

5. Managing the HTTP response (1/1)

In this section, we discuss managing the HTTP response in the controller's action.
The HTTP response holds data as the following:

```
HTTP/1.1 200 OK
```

Status response - A short representation of the request's result

```
Date: Mon, 20 May 2024 12:28:58 GMT
Server: Apache/2.2.14(Win64)
Last-Modified: Wed, 24 Fri 2024 19:54:11 GMT
Content-Length: 88
Content-Type: text/html
Connection: Closed
```

Header response - Short pieces of data in the response (usually not more than a few words long)

```
<html>
<body>
<h1>Hello Spring Start Here!</h1>
</body>
</html>
```

Body response - A larger amount of data the backend needs to send in the response

5.1 Sending objects as a response body (1/4)

Model of the data the server returns in the HTTP response body

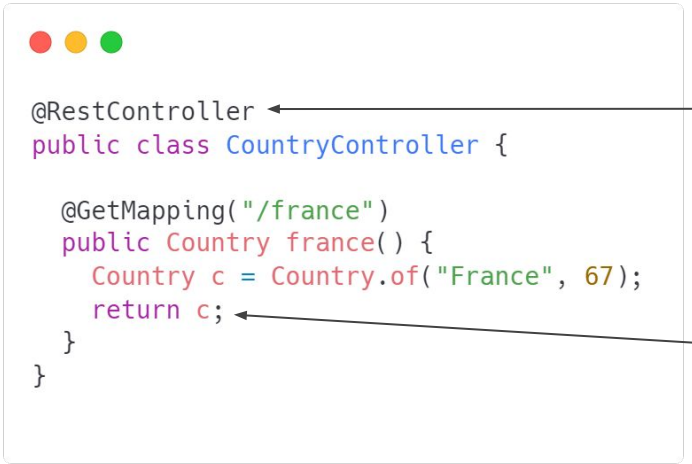
```
public class Country {

  private String name;
  private int population;

  public static Country of(
    String name,
    int population) {
      Country country = new Country();
      country.setName(name);
      country.setPopulation(population);
      return country;
  }
  // Omitted getters and setters
}
```

To make a Country instance simpler, we define a static factory method that receives the name and the population. This method returns a Country instance with the provided values set.

When we use an object (such as Country) to model the data transferred between two apps, we name this object a data transfer object (**DTO**). We can say that Country is our **DTO**, whose instances are returned by the **REST endpoint** we implement in the **HTTP response body.**

5.1 Sending objects as a response body (2/4)

Returning an object instance from the controller's action



```
@RestController
public class CountryController {

  @GetMapping("/france")
  public Country france() {
    Country c = Country.of("France", 67);
    return c;
  }
}
```

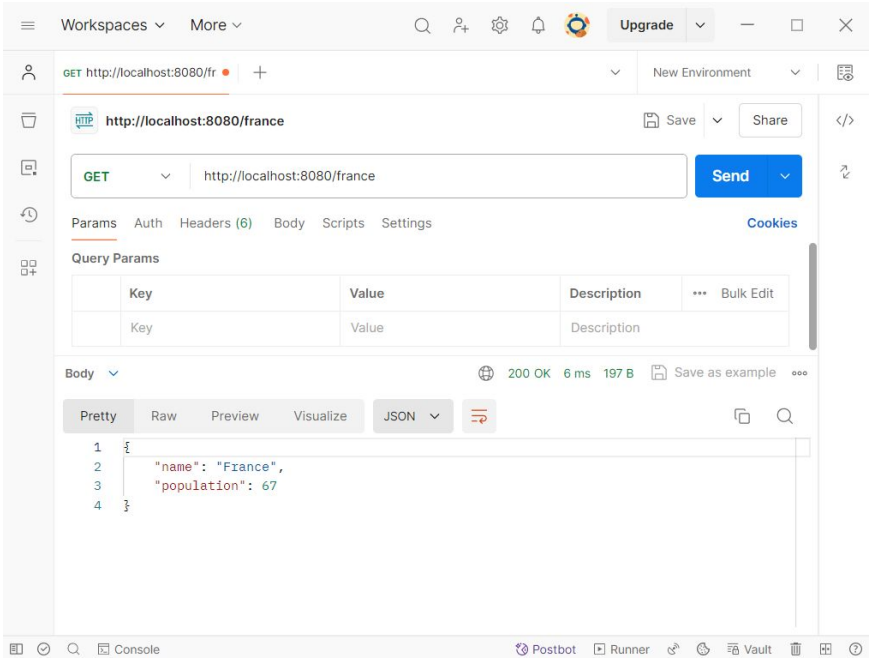
Marking the class as a REST controller to add a bean in the Spring context and also inform the dispatcher servlet not to look for a view when this method returns

Mapping the controller's action to the HTTP GET method and /france path

Returning an instance of type Country

5.1 Sending objects as a response body (3/4)

When calling the /france endpoint, the response body looks as presented



Once you press the ***Send button***, Postman sends the request. When the request completes, Postman displays the response details, including the *response body*.

5.1 Sending objects as a response body (4/4)

- The next listing shows that we added a method that returns a List of Country objects.
- **Returning a collection in the response body**

```
@RestController
public class CountryController {

  // Omitted code

  @GetMapping("/all")
  public List<Country> countries() {
    Country c1 = Country.of("France",
67);
    Country c2 = Country.of("Spain", 47);
    return List.of(c1,c2);
  }

}
```

Returns a collection in the HTTP response body

When we call this endpoint, the response body looks as presented:

```
[
 {
   "name": "France",
   "population": 67
 },
 (
   "name": "Spain",
   "population": 47
 )
]
```

In **JSON**, the list is defined with brackets.

Each object is between curly braces, and the objects are separated with commas.

Besides **JSON**, Spring offers the possibility of using other ways to format the response body (like **XML** or **YAML**) if we'd like.

5.2 Setting the response status and headers (1/4)

The response status is also an essential flag in the HTTP response we use to signal the request's result. By default, **Spring** sets some common HTTP statuses:

CODE	MESSAGE	DESCRIPTION
200	OK	If no exception was thrown on the server side while processing the request
404	Not Found	If the requested resource doesn't exist
400	Bad Request	If a part of the request could not be matched with the way the server expected the data
500	Error on server	If an exception was thrown on the server side for any reason while processing the request.

5.2 Setting the response status and headers (2/4)

Adding custom headers and setting a response status

```
@RestController
public class CountryController {

  @GetMapping("/france")
  public ResponseEntity<Country> france() {
    Country c = Country.of("France", 67);
    return ResponseEntity
            .status(HttpStatus.ACCEPTED)
            .header("continent", "Europe")
            .header("capital", "Paris")
            .header("favorite_food", "cheese and wine")
            .body(c);
  }
}
```

Changes the HTTP response status to 202 Accepted

Adds three custom headers to the response

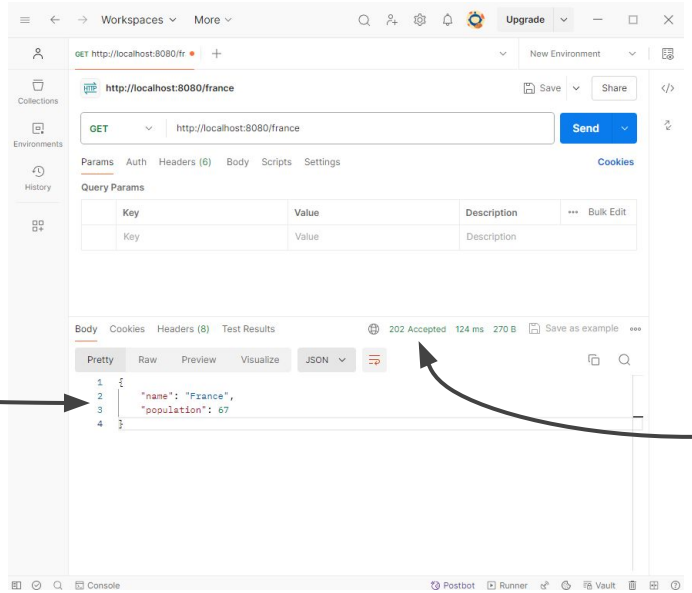
Sets the response body

Some requirements ask us to configure a custom status.
The easiest and most common way to customize the HTTP response is using the **ResponseEntity** class. This class provided by Spring allows us to specify the response body, status, and headers on the **HTTP response.**

5.2 Setting the response status and headers (3/4)

Once we send the **HTTP** *request* by pressing the *Send button* and get the **HTTP** *response*, we observe the **HTTP** *response status* is **202 Accepted**. We can still see the response body as a **JSON** formatted string.

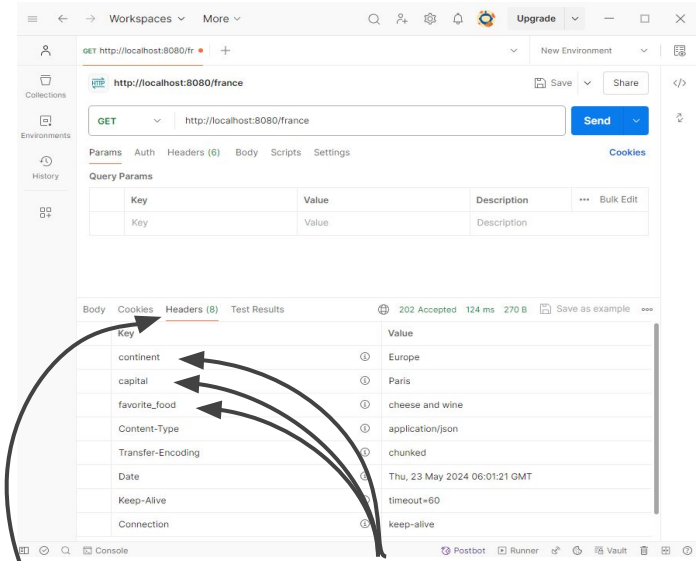
The response body still contains the JSONformatted Country object.



Aside from the response body, you can observe the HTTP response status changed to 202 Accepted.

5.2 Setting the response status and headers (4/4)

To see the customer *headers* in **Postman**, you have to navigate to the *Headers tab* of the ***HTTP*** *response*.



In the headers tab, you find the custom headers you added on the HTTP response.

5.3 Managing exceptions at the endpoint level (1/7)

Suppose we create an endpoint the client calls to make a payment. If the user doesn't have enough money in their account, the app might represent this situation by throwing an exception.

```
public class NotEnoughMoneyException extends RuntimeException {
}
```

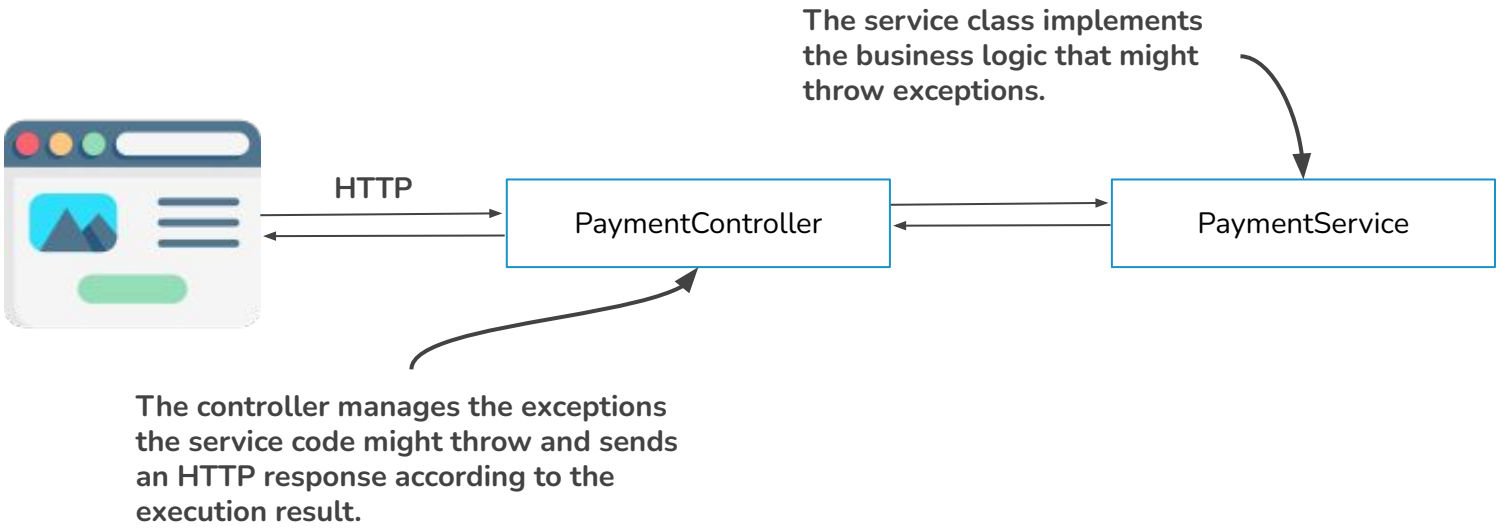
```
@Service
public class PaymentService {

  public PaymentDetails processPayment() {
    throw new NotEnoughMoneyException();
  }
}
```

```
public class PaymentDetails {

  private double amount;
// Omitted getters and setters
}
```

```
public class ErrorDetails {

  private String message;
// Omitted getters and setters
}
```

5.3 Managing exceptions at the endpoint level (2/7)

The **PaymentService** class implements the business logic that might *throw exceptions*. The **PaymentController** class manages the *exception* and sends the client an **HTTP** *response* according to the execution result.



5.3 Managing exceptions at the endpoint level (3/7)

Managing the **HTTP** *response* for exceptions in the Controller's action

```
@RestController
public class PaymentController {

  private final PaymentService paymentService;

  public PaymentController(PaymentService paymentService) {
    this.paymentService = paymentService;
  }

  @PostMapping("/payment")
  public ResponseEntity<?> makePayment() {

    try {
      PaymentDetails paymentDetails =
        paymentService.processPayment();

      return ResponseEntity
              .status(HttpStatus.ACCEPTED)
              .body(paymentDetails);
    } catch (NotEnoughMoneyException e) {
      ErrorDetails errorDetails = new ErrorDetails();
      errorDetails.setMessage("Not enough money to make the payment.");
      return ResponseEntity
              .badRequest()
              .body(errorDetails);
    }
  }
}
```

We try calling the **processPayment()** method of the service.

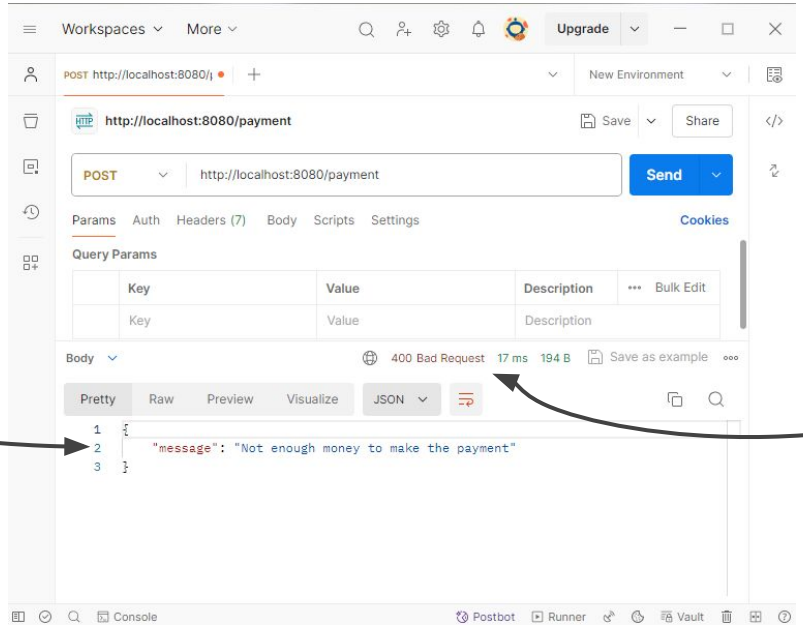
If calling the service method succeeds, we return an **HTTP** response with status *Accepted* and the **PaymentDetails** instance as a response body.

If an exception of type **NotEnoughMoneyException** is thrown, we return an **HTTP** *response* with status **Bad Request** and an ***ErrorDetails*** instance as a body.

5.3 Managing exceptions at the endpoint level (4/7)

We know that we made the service method to always throw the NotEnoughMoneyException, so we expect to see the response status message is **“400 Bad Request”**.

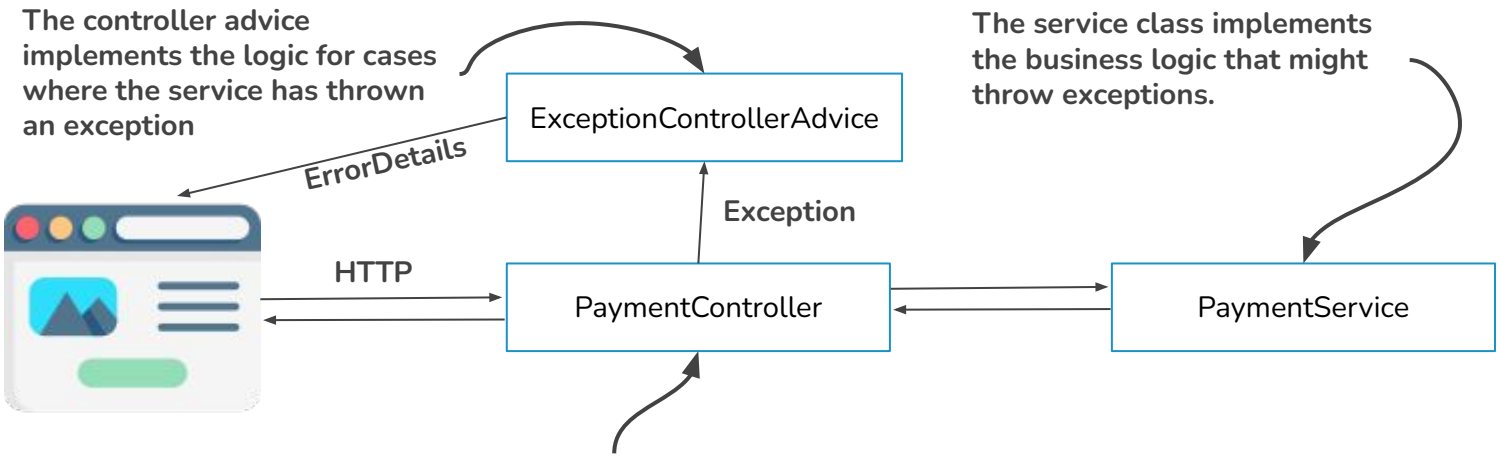
The **exception** description appears in the HTTP *response body.*



The **status** of the HTTP response is **400 Bad Request.**

5.3 Managing exceptions at the endpoint level (5/7)

Instead of managing the exception cases, the controller now only takes care of the happy flow. We added a controller advice named **ExceptionControllerAdvice** to take care of the logic that will be implemented if the controller's action throws an exception.



5.3 Managing exceptions at the endpoint level (6/7)

Controller's action that no longer treats the exception case

```
@RestController
public class PaymentController {

  private final PaymentService paymentService;

  public PaymentController(PaymentService paymentService) {
    this.paymentService = paymentService;
  }

  @PostMapping("/payment")
  public ResponseEntity<PaymentDetails> makePayment() {
      PaymentDetails paymentDetails = paymentService.processPayment();
      return ResponseEntity
              .status(HttpStatus.ACCEPTED)
              .body(paymentDetails);
  }
}
```

5.3 Managing exceptions at the endpoint level (7/7)

Separating the exception logic with a REST controller advice

```
@RestControllerAdvice
public class ExceptionControllerAdvice {

  @ExceptionHandler(NotEnoughMoneyException.class)
  public ResponseEntity<ErrorDetails> exceptionNotEnoughMoneyHandler() {
    ErrorDetails errorDetails = new ErrorDetails();
    errorDetails.setMessage("Not enough money to make the payment.");
    return ResponseEntity
        .badRequest()
        .body(errorDetails);
  }
}
```

We use the **@RestControllerAdvice** annotation to mark the class as a **REST** controller advice.

We use the **@ExceptionHandler** method to associate an exception with the logic the method implements.

The following listing shows the **REST *controller advice*** class's definition and the *exception* handler method that implements the logic associated with the **NotEnoughMoneyException** exception.

5.3 Using a request body to get data from the client

Getting data from the client in the request body

```
@RestController
public class PaymentController {

  private static Logger logger =
    Logger.getLogger(PaymentController.class.getName());

  @PostMapping("/payment")
  public ResponseEntity<PaymentDetails> makePayment(
      @RequestBody PaymentDetails paymentDetails) {

      logger.info("Received payment " +
      paymentDetails.getAmount());

      return ResponseEntity
              .status(HttpStatus.ACCEPTED)
              .body(paymentDetails);
  }
}
```

We get the payment details from the **HTTP** *request body*.

We get the payment details from the **HTTP** *request body*.

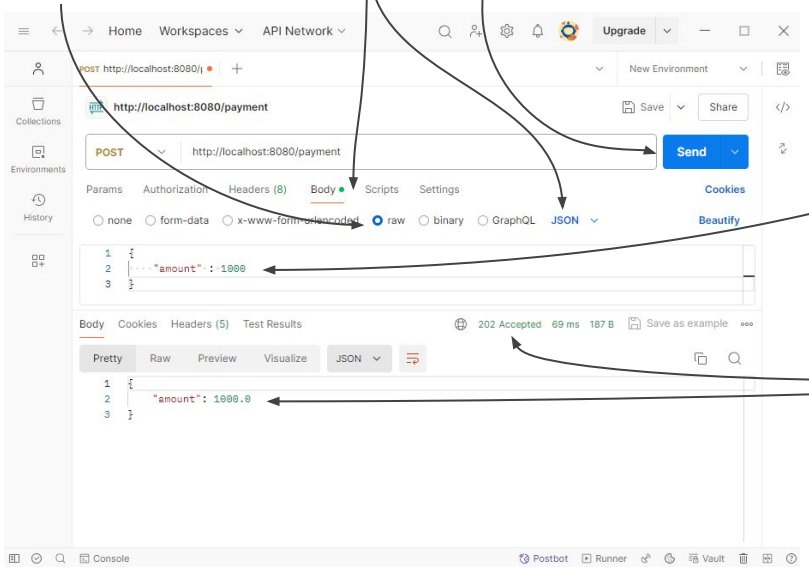
We get the payment details from the **HTTP** *request body*.

Code, shows us how to use **Postman** to call the ***/payment*** endpoint with a request body.

5.3 Using a request body to get data from the client

To set the *request body*, we select the *Body tab* on the **HTTP** request configuration. We then choose the *option* "***raw***" by clicking the ***radio button*** and then choose **JSON** as the formatting style.

In the text area we write the **JSON**-formatted *request body* representing the **PaymentDetails** object. Then we click the ***Send button*** to send the **HTTP** *request*.



Once the *request* completes, **Postman** shows the *response details*.

5.3 Using a request body to get data from the client

Using **Postman** to call the endpoint and specify the request body, We need to fill the JSONformatted request body in the request body text area and select the data encoding as **JSON**. Once the request completes, **Postman** displays the response details.

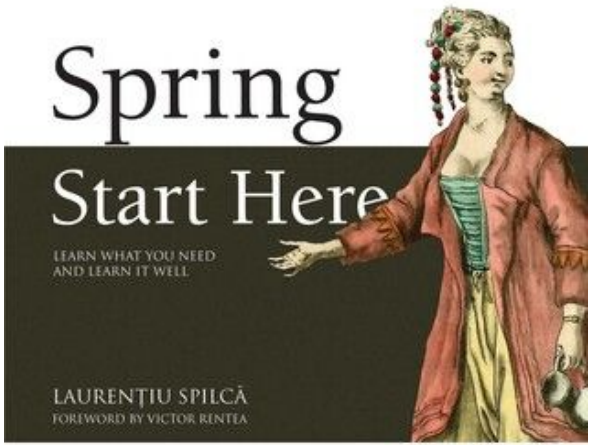
- If we prefer using **cURL**, we can use the ***command*** presented by the next line:

```
curl -v -X POST http://127.0.0.1:8080/payment -d '{"amount": 1000}' -H
➥ "Content-Type: application/json"
```

7. Conclusion

- Representational state transfer (**REST**) ***web services*** are a simple way to establish communication between two applications.
- In a Spring app, the Spring MVC mechanism supports the implementation of **REST *endpoints***. We either need to use the **@ResponseBody** annotation to specify that a method directly returns the response body or replace the **@Controller** annotation with **@RestController** to implement a **REST *endpoint***. If we don't use one of these, the dispatcher servlet will assume the controller's method returns a view name and try to look for that view instead.
- We can make the controller's action directly return the **HTTP** response body and rely on Spring default's behavior for the **HTTP** status.
- We can manage the **HTTP** status and headers by making your controller's action return a **ResponseEntity** instance.
- One way to manage exceptions is to treat them directly at the controller's action level. This approach couples the logic used to treat the exception to that specific controller action. Sometimes using this approach can lead to code duplication, which is best to avoid.
- We can manage the exceptions in the controller's action directly or separate the logic executed if the controller's action throws an exception using a **REST** controller advice class.
- An *endpoint* can get data from the client through the **HTTP** request in request parameters, path variables, or the **HTTP** *request body*.

Resources



MANNING

Reference

1. Spring Start Here
2. Baeldung.com
3. RESTfulAPI.net

Thank you!

Presented by

Urunov Khamdamboy

(hamdamboy.urunov@gmail.com)

Created By

Nizomiddinov Shakhobiddin

(shohobiddindeveloper@gmail.com)