Spring Fundamentals

WEB API and REST Controllers

SoftUni Team

Technical Trainers

Software University

https://softuni.bg

Table of Contents

Software University

Questions

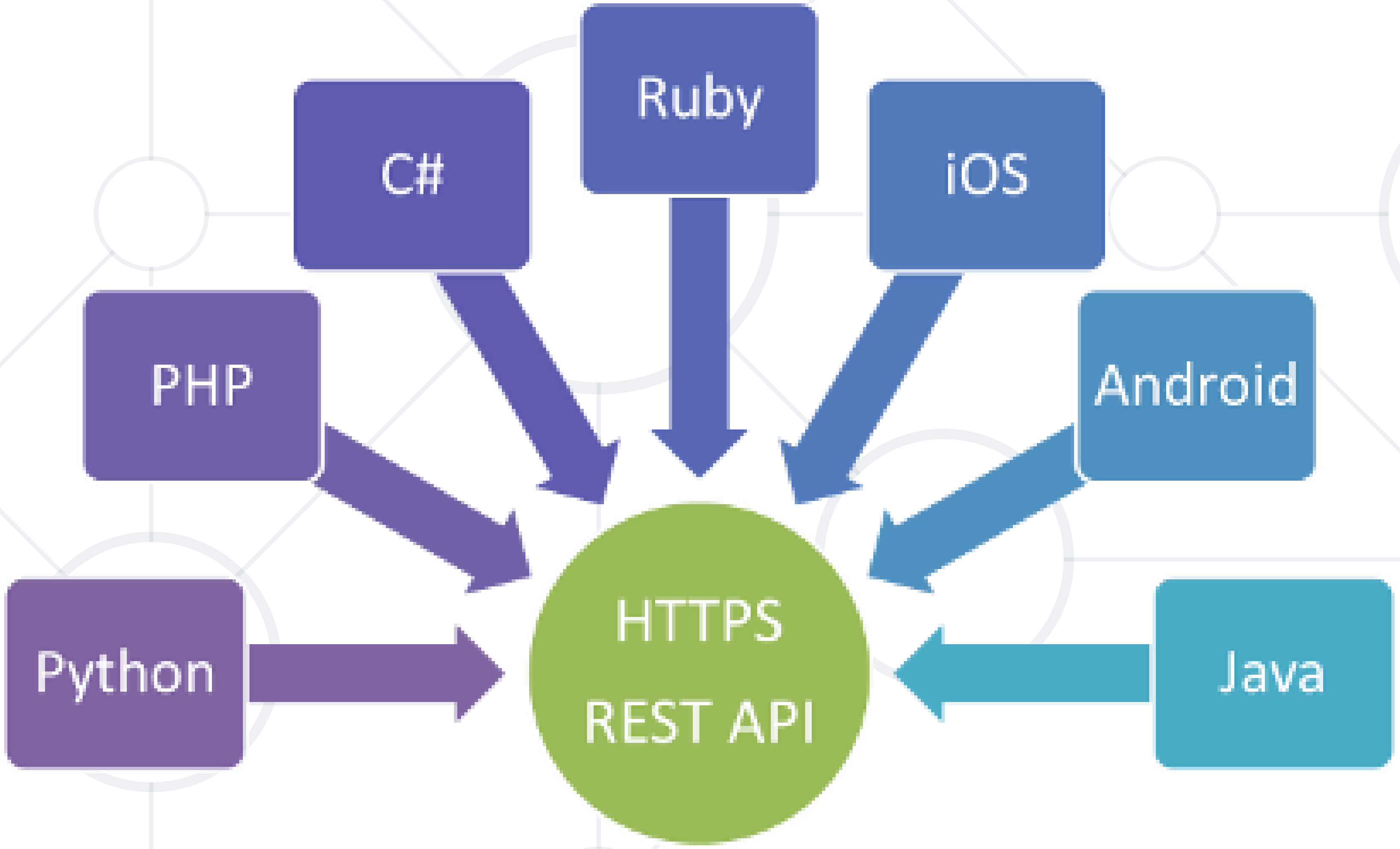
sli.do

#java-web

REST API

RESTful Design

RESTful Design



RESTful API

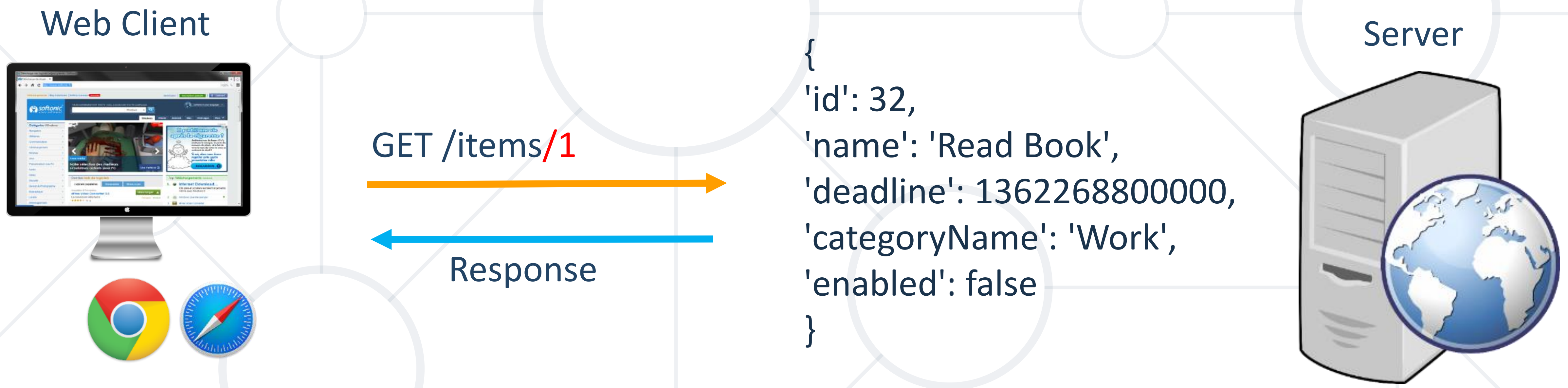
- True RESTful API, is a **web service** must adhere to the following six **REST architectural constraints**
  - Use of a **uniform interface** (UI)
  - **Client-server based**
  - **Stateless** operations
  - RESTful **resource caching**
  - **Layered system**
  - **Code on demand**

SOAP and RPC

- **Simple Object Access Protocol** (SOAP)
  - Standardized protocol that **sends messages** using other protocols such as **HTTP** and **SMTP**
  - The SOAP specifications are official web standards, maintained and developed by the World Wide Web Consortium (W3C)
- **Remote Procedure Call** (RPC)
  - A way to describe a mechanism that lets you **call a procedure in another process** and **exchange data by message passing**

HTTP GET (1)

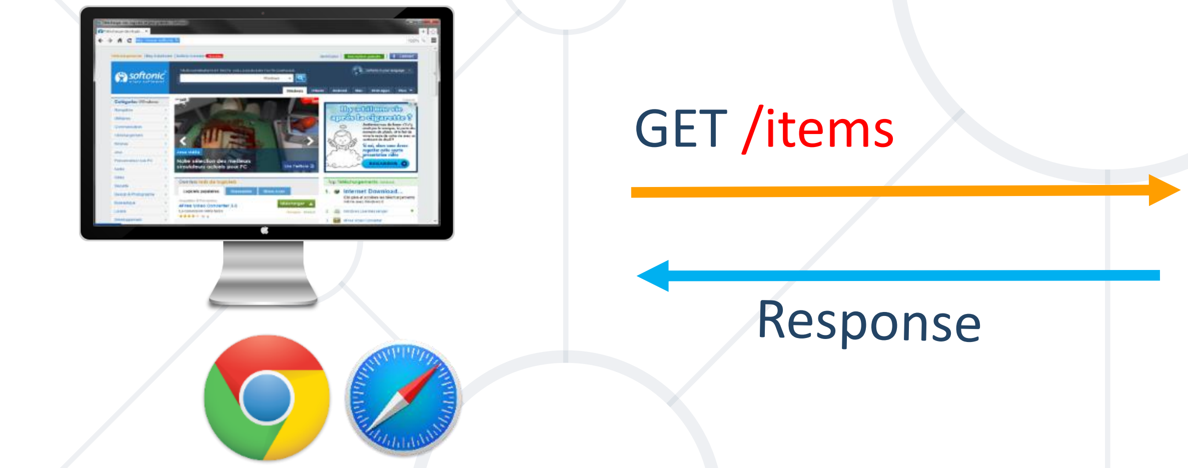
- Used to retrieve single data entities



```
{
'id': 32,
'name': 'Read Book',
'deadline': 1362268800000,
'categoryName': 'Work',
'enabled': false
}
```

HTTP GET (2)

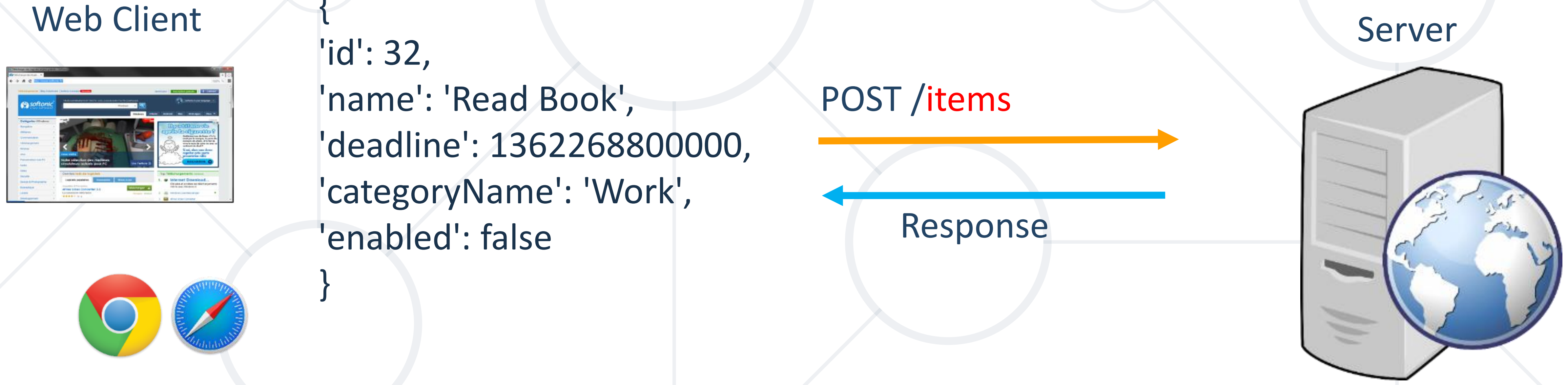
- Used to retrieve data arrays



```
[
{
'id': 32,
'name': 'Read Book',
'deadline': 1362268800000,
'categoryName': 'Work',
'enabled': false
},
…
]
```

HTTP POST

- Used to save data



```
{
'id': 32,
'name': 'Read Book',
'deadline': 1362268800000,
'categoryName': 'Work',
'enabled': false
}
```

HTTP PUT

- Used to update data.

Web Client

```
{
'id': 32,
'name': 'Read News',
'deadline': 1362268800000,
'categoryName': 'Work',
'enabled': false
}
```

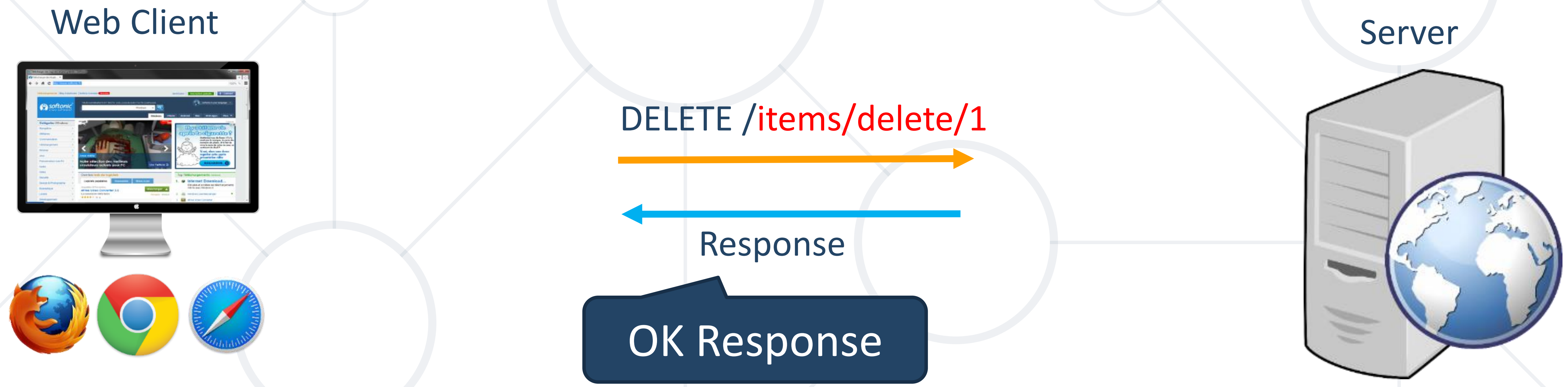
PUT /items/1

Response

Server

HTTP DELETE

- Used to delete data.



REST with Spring

Creating REST API with Spring

Response Body On MVC Controller

- Returning plain-text in MVC controller:

```
@GetMapping('/info/{id}')
@ResponseBody
public Student getInfo(@PathVariable Long id){

  ...
  return new Student().setName(“Joro”);
}
```

Response Status

- Setting the correct Response Code

```
@GetMapping('{id}/info')
@ResponseStatus(HttpStatus.OK)
public String getInfo(@PathVariable Long id){

  GameInfoView gameInfo = this.gameService.getInfoById(id);

  return new Gson().toJson(gameInfo);
}
```

REST Controllers

- **@RestController** is essentially **@Controller** + **@ResponseBody**

```
@RestController
public class OrderController {

  @GetMapping('{id}/info')
  public ResponseEntity<Game> getGame(@PathVariable Long id){
    ...
  }
}
```

Response Entity

- Controlling the entire response object

```
@GetMapping('{id}/title')
public ResponseEntity<Game> getTitle(...){
  ...
  return new ResponseEntity<>(gameService.getGame(id), HttpStatus.OK);
}
```

- The **ResponseEntity<>** object allows you **to change the response body**, response headers and response code

Spring Data REST

- Maven Dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-rest</artifactId>
</dependency>
```

- Spring Data REST **scans your project** and **provides REST API** for your application **using HAL** as media type

Configuring Repositories

- You can configure repository settings using the **@RepositoryRestResource** annotation:

```
@RepositoryRestResource(path = 'gameIssues')
public interface IssueRepository extends
                                JpaRepository<Issue, Long> {
    Issue getById(@Param('id') Long id);
    List<Issue> getAllByOrderByDateDesc();
}
```

Rest Template

Rest Template

- Accessing **a third-party REST service** inside a Spring application revolves around the use of the Spring **RestTemplate class**
- Class is **designed to call REST services**
- Its **main methods** are closely tied to **REST's underpinnings**, which are the **HTTP protocol's methods**: **HEAD**, **GET**, **POST**, **PUT**, **DELETE**
- **Recommended** to use the non-blocking, **reactive WebClient**.
- RestTemplate will be **deprecated in a future version**

HTTP GET Method Example (1)

- `getForObject(url, classType)`
  - Retrieves a **representation by doing a GET on the URL**.
  - The response (if any) is unmarshalled to given class type and returned
- `getForEntity(url, responseType)`
  - Retrieve a **representation as ResponseEntity** by doing a GET on the URL

HTTP GET Method Example (2)

- `exchange(requestEntity, responseType)`
  - **Executes** the specified **request** and **returns** the response as **ResponseEntity**
- `execute(url, httpMethod, requestCallback, responseExtractor)`
  - **Executes the httpMethod** to the given URI template and **preparing the request** with the **RequestCallback**

HTTP POST (1)

- `postForObject(url, request, classType)`
  - **POSTs** the given object **to the URL** and **returns the representation** found in the response **as given class type**
- `postForEntity(url, request, responseType)`
  - **POSTs** the given object **to the URL** and **returns the response as ResponseEntity**

HTTP POST (2)

- `postForLocation(url, request, responseType)`
  - **POSTs** the given object **to the URL** and **returns** the value of the **Location header**
- `exchange(url, requestEntity, responseType)`
- `execute(url, httpMethod, requestCallback, responseExtractor)`

HTTP PUT and HTTP DELETE

- `put(url, request)`
  - PUTs the given request object to URL
- `delete(url)`
  - Deletes the resource at the specified URL

DOM Manipulations

Creating DOM Elements

- Create with document.createElement

```
let p = document.createElement('p');
```

- Append text to the <p> element

```
let text = document.createTextNode('Random Text');
```

```
p.appendChild(text);
```

- Text added to textContent will be escaped.
- Text added to innerHTML will be parsed and turned into actual HTML elements beware of XSS attacks!

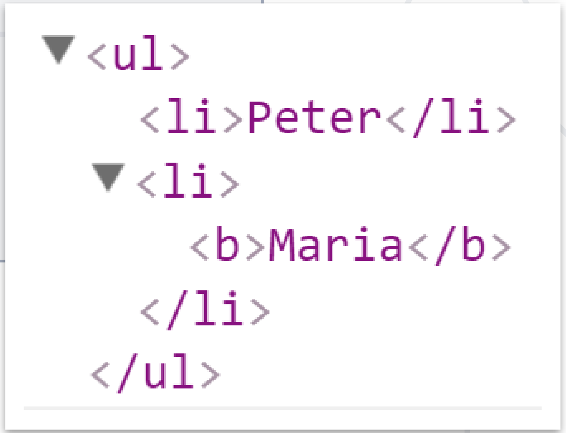
Creating DOM Elements

```
let list = document.createElement('ul');

let liPeter = document.createElement('li');
liPeter.textContent = 'Peter';
list.appendChild(liPeter);

let liMaria = document.createElement('li');
liMaria.innerHTML = '<b>Maria</b>';
list.appendChild(liMaria);

document.body.appendChild(list);
```



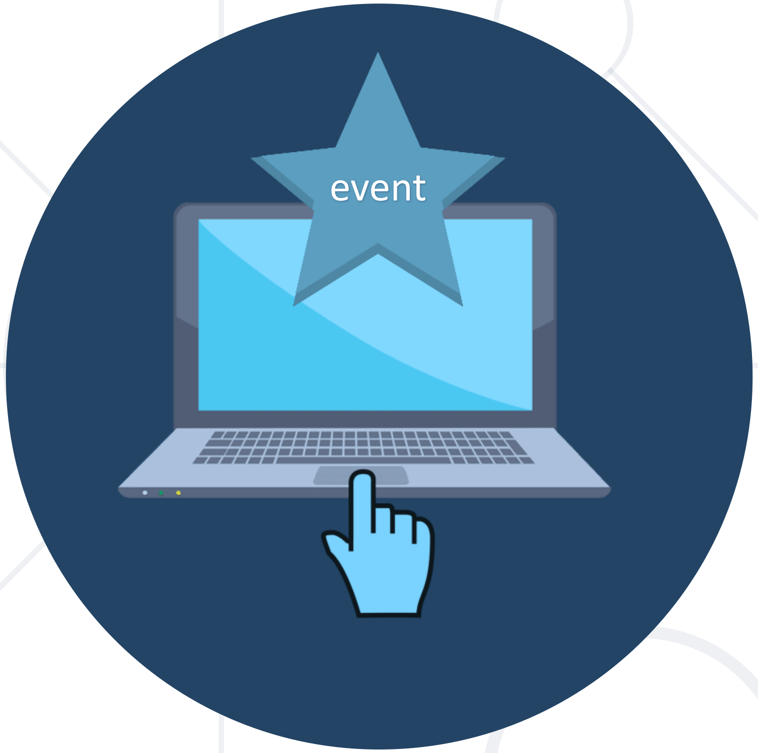
```
▼<ul>
   <li>Peter</li>
  ▼<li>
     <b>Maria</b>
   </li>
 </ul>
```

Deleting DOM Elements

- To remove an HTML element, you must know his parent

```
<div id='div1'>
     <p id='p1'>This is a paragraph.</p>
     <p id='p2'>This is another paragraph.</p>
</div>
```

```
let parent = document.getElementById('div1');
let child = document.getElementById('p1');
parent.removeChild(child);
```



Handling Events

Browser Events and DOM Events

Handling Events in JS

- Browsers send events to notify the JS code of interesting things that have taken place

```
<div id='text'>Some text</div>
```

```
let div = document.getElementById('text');
div.onmouseover = function(event) {
  event.target.style.border = '3px solid green';
}
div.onmouseout = function() {
  this.style.border = ''; // this === event.target
}
```

Event Types in DOM API

- **Mouse** events

```
click
mouseover
mouseout
mousedown
mouseup
```

- **Touch** events

```
touchstart
touchend
touchmove
touchcancel
```

- **DOM / UI** events

```
load
unload
resize
dragstart / drop
```

- **Keyboard** events

```
keydown
Keypress
keyup
```

- **Focus** events

```
focus (got focus)
blur (lost focus)
```

- **Form** events

```
input
change
submit
reset
```

Attach / Remove Events

- Attach an event to an element.

```
let textbox = document.createElement('input');
textbox.type = 'text';
textbox.value = 'I am a text box';
document.body.appendChild(textbox);

textbox.addEventListener('focus', focusHandler);
```

- Remove an event.

```
function focusHandler(event) {
  textbox.value = 'Event handler removed';
  textbox.removeEventListener('focus', focusHandler);
}
```

Multiple Events

- The **addEventListener()** method also allows you to add many events to the same element, without overwriting existing events:

```
element.addEventListener('click', function);
element.addEventListener('click', myFunction);
element.addEventListener('mouseover', mySecondFunction);
element.addEventListener('mouseout', myThirdFunction);
```

- Note that you don't use the 'on' prefix for the event; use 'click' instead of 'onclick'.

Fetch API

Fetch API

- Fetch provides a generic definition of Request and Response objects
- Fetch API allows you to make network requests similar to **XMLHttpRequest** (XHR).
- The response of a **fetch()** is a Stream object.

Fetch API (Demo) (1)

```java
@GetMapping('/')
public ModelAndView index(ModelAndView modelAndView) {
    modelAndView.setViewName('index');
    return modelAndView;
}

@GetMapping(value = '/fetch', produces = 'application/json')
@ResponseBody
public Object fetchData() {
    return new ArrayList<Product>() {{
        add(new Product(){{
            setName('Chewing Gum');
            setPrice(new BigDecimal(1.00));
            setBarcode('133242556222');
        }});
        ...
    }};
}
```

HomeController.java

```java
public class Product {
    private String name;
    private BigDecimal price;
    private String barcode;

    // Getters & Setters
    ...
}
```

Product.java

Fetch API (Demo) (2)

- Now let's head to the view
  - There is no need for a separate .js file for one-time use

index.html

```
...
<div class='container-fluid'>
    <h1 class='text-center mt-5 display-1'>Data Fetch</h1>
    <div class='data-container mt-5'></div>
    <div class='button-holder mt-5'>
        <button id='fetch-button' class='btn btn-info'>Fetch Data</button>
        <button id='clear-button' class='btn btn-secondary'>Clear Data</button>
    </div>
</div>
<script>
    // jQuery Event handlers
    $('#fetch-button').click(() => {...}); // Fetch and render the data
    $('#clear-button').click(() => $('.data-container').empty()); // Clear the data
</script>
```

Fetch API (Demo) (3)

```
$('#fetch-button').click(() => {
    fetch('http://localhost:8000/fetch') // Fetch the data (GET request)
        .then((response) => response.json()) // Extract the JSON from the Response
        .then((json) => json.forEach((x, y) => { // Render the JSON data to the HTML
                if (y % 4 === 0) {
                    $('.data-container').append('<div class='row d-flex justify-content-
around mt-4'>');
                }

                let divColumn =
                    '<div class='col-md-3'>' +
                    '<h3 class='text-center font-weight-bold'>' + x.name + '</h3>' +
                    '<h4 class='text-center'>Price: $' + x.price + '</h4>' +
                    '<h4 class='text-center'>Barcode: $' + x.barcode + '</h4>' +
                    '</div>';

                $('.data-container .row:last-child').append(divColumn);
            }));
});
```

What is HATEOAS

Hypermedia As the Engine of Application State

- **HATEOAS** is a constraint of the REST application architecture
- Keeps the RESTful style architecture **unique from most other network application** architectures
- Uses **hypermedia** to describe what future actions are available to the client
- Allowable actions are derived in the API based on the current application state and returned to the client as a **collection of links**

Hypermedia As the Engine of Application State (2)

- Client uses these **links to drive further** interactions with the API
- Tells the client what **options** are **available** at a given point in time.
  - Doesn't tell them how each link should be used or exactly what information should be sent
- It is conceptually the same as a **web user browsing** through web pages by clicking the **relevant hyperlinks** to achieve a final goal

HATEOAS Example

HATEOAS Example (1)

- Simple response **without** using **HATEOAS**
  - We have a simple REST controller that returns entity in JSON format to the client

```
{ "id" :2, "name": "Peter", "age":12 }
```

HATEOAS Example (2)

- **Using HATEOAS**

```
{ "id":2,"name":"Pesho","age":12,"
  _links":{
    "self":{"href":"http://localhost:8080/students/2"},

    "delete":{"href":"http://localhost:8080/students/delete/2"},

   "update":{"href":"http://localhost:8080/students/update/2"},

   "orders":{"href":"http://localhost:8080/orders/allByStudentId/2"}

    } }
```

Benefits of Using HATEOAS

- **URL structure** of the API can be **changed without affecting** clients
  - If the URL structure is changed in the service, clients will automatically pick up the new URL structure via hypermedia
- Hypermedia APIs are **explorable**
- Guiding clients toward the next step in the workflow by **providing** only the **links** that are **relevant** based on the current application state

Negatives of Using HATEOAS

- Adds **extra complexity** to the API, which affects to:
  - **developer** needs to handle the **extra work** of adding links to each response
  - **more complex** to **build** and **test** than a vanilla CRUD REST API
  - **clients** also have to deal with the **extra complexity** of **hypermedia**

Summary

- What is the REST Controllers
- Rest Templates
- How to manipulate DOM
  - Creating and appending html elements
- Using JQuery and Fetch

Software University

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

Flutter International

INDEAVR Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

BOSCH

Postbank Решения за твоето утре

PHAR VISION

SmartIT

DXC TECHNOLOGY

createX

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

License