REST API and REST With Spring

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Questions

Software University

sli.do

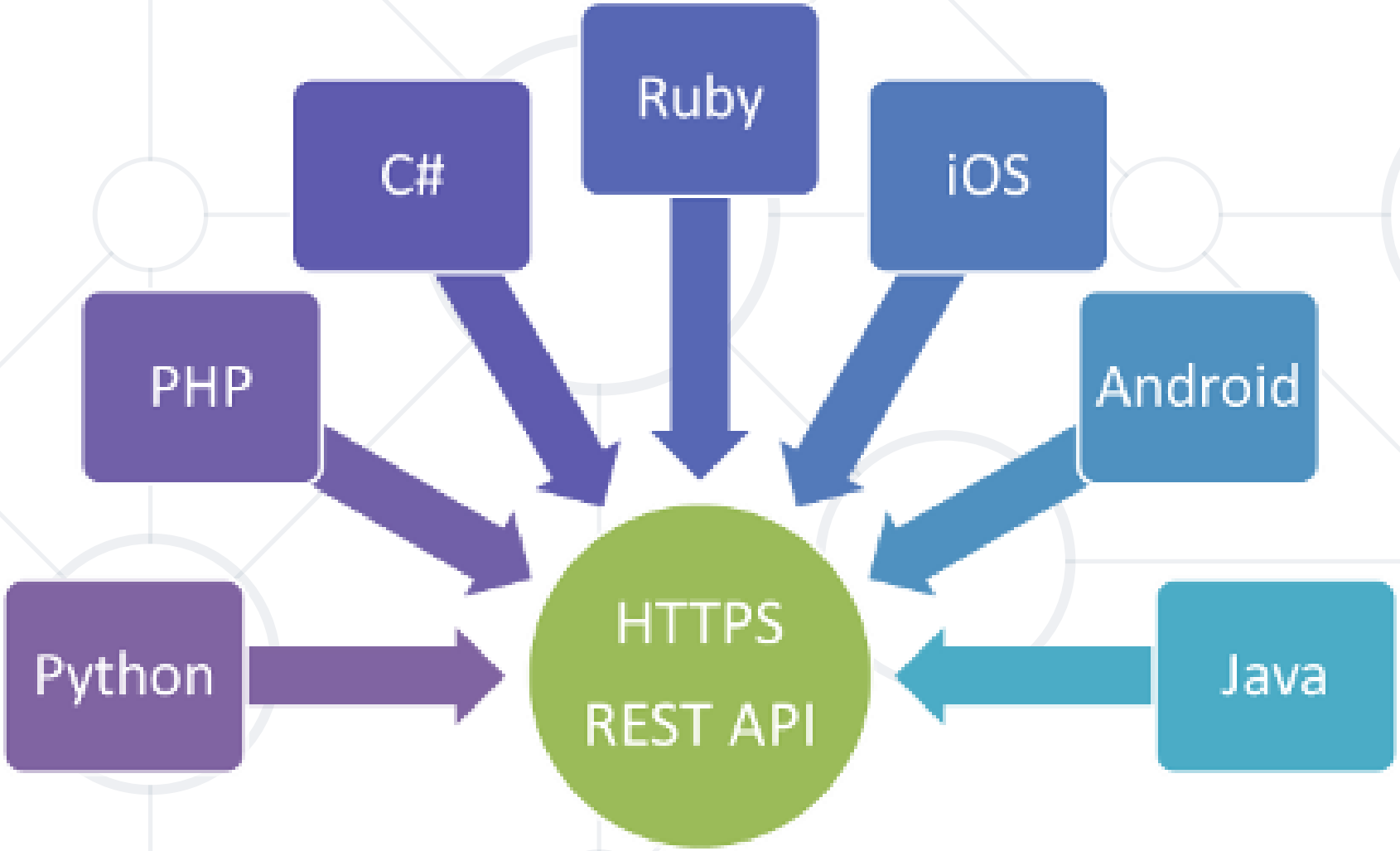
#java-web

Table of Contents

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

- Used to retrieve single data entities

Web Client

GET /items/1

Response

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

Server

HTTP GET

- Used to retrieve data arrays

GET /items

Response

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

Server

HTTP POST

- Used to save data

Web Client

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

POST /items

Response

Server

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

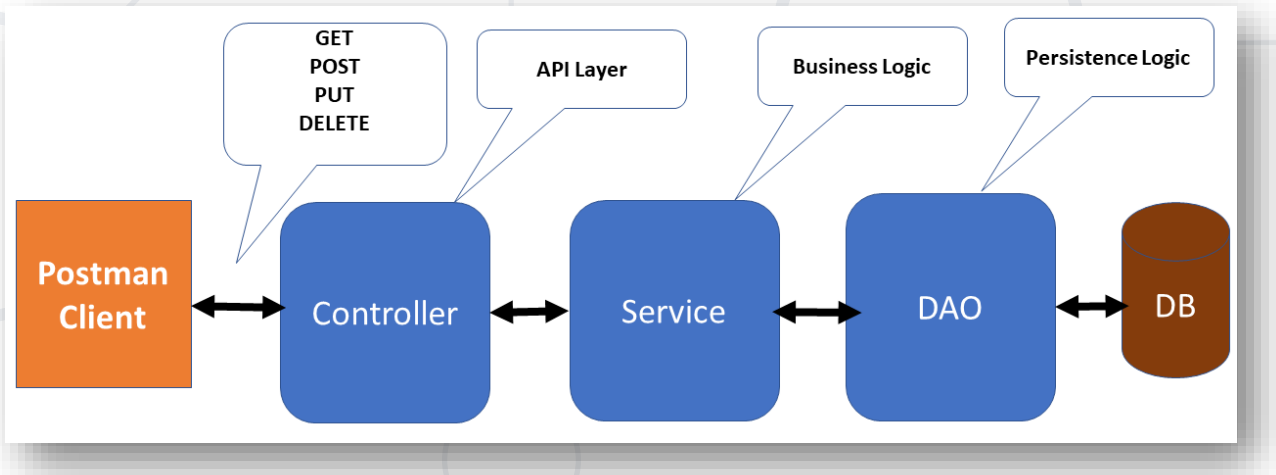
- You can configure repository settings using the `@RepositoryRestResource` annotation:

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

Rest Client

Rest Client

- The **RestClient** is a synchronous **HTTP client** that offers a modern, fluent API
- It offers an **abstraction** over HTTP libraries that allows for convenient conversion from a **Java object** to an **HTTP request**
- It creates objects from an HTTP response



Creating a RestClient

Software University

- The **RestClient** is created using one of the **static create** methods
- You can also use **builder()** to get a builder with further options:
  - specifying which HTTP library to use
  - which message converters to use
  - setting a default URI, default path variables, default request headers, or uriBuilderFactory, or registering interceptors and initializers
- Once created (or built), the **RestClient** can be used safely by **multiple** threads

Creating a RestClient

```
RestClient defaultClient = RestClient.create();

RestClient customClient = RestClient.builder()
  .requestFactory(new HttpComponentsClientHttpRequestFactory())
  .messageConverters(converters -> converters.add(new
MyCustomMessageConverter()))
  .baseUrl("https://example.com")
  .defaultUriVariables(Map.of("variable", "foo"))
  .defaultHeader("My-Header", "Foo")
  .requestInterceptor(myCustomInterceptor)
  .requestInitializer(myCustomInitializer)
  .build();
```

Using the RestClient

- When making an **HTTP request** with the **RestClient**, the first thing to specify is which HTTP **method** to use
- This can be done with method(HttpMethod) or with the convenience methods **get(), head(), post(),** and so on
- Request **URL**
  - the request URI can be specified with the **URI methods**
  - The URL is typically specified as a String

```
int id = 42; // GET request to example.com/orders/42
restClient.get()
  .uri("https://example.com/orders/{id}", id)
  ....
```

Retrieving the Response

- The HTTP **response** is accessed by invoking **retrieve()**
- The **response body** can be accessed by using **body(Class)** or **body(ParameterizedTypeReference)** for parameterized types like lists
- The body method **converts** the response contents into various types
  - bytes can be converted into a **String**
  - **JSON** can be converted into objects using Jackson, and so on

Retrieving the Response

- Set up a **GET** request
- Specify the **URL** to connect to
- **Retrieve** the response
- Convert the **response** into a String
- Print the **result**

```
String result = restClient.get().uri("https://example.com")
  .retrieve().body(String.class);

System.out.println(result);
```

Retrieving the Response

- Access to the response status code and headers is provided through **ResponseEntity**
  - Set up a GET request for the specified URL
  - Convert the **response** into a **ResponseEntity**
  - Print the **result**

```
ResponseEntity<String> result = restClient.get().uri("https://example.com")
  .retrieve().toEntity(String.class);

System.out.println("Response status: " + result.getStatusCode());
System.out.println("Response headers: " + result.getHeaders());
System.out.println("Contents: " + result.getBody());
```

Retrieving the Response

- **RestClient** can convert JSON to **objects**, using the **Jackson** library
  - Using URI variables
  - Set the Accept header to **application/json**
  - Convert the **JSON response** into a **Pet domain** object

```
int id = ...;
Pet pet = restClient.get()
   .uri("https://petclinic.example.com/pets/{id}", id)
   .accept(APPLICATION_JSON)
   .retrieve()
   .body(Pet.class);
```

Retrieving the Response

- Create a Pet domain object
- Set up a **POST** request and the URL to connect to
- Set the **Content-Type** header to **application**/json
- Use pet as the **request** body
- Convert the response into a **response entity** with no body

```
Pet pet = ...
ResponseEntity<Void> response = restClient.post()
  .uri("https://petclinic.example.com/pets/new")
  .contentType(APPLICATION_JSON)
  .body(pet)
  .retrieve()
  .toBodilessEntity();
```

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

- 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

- 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

- Using HATEOAS

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

Rel & Href

- **rel** - describes the **relationship** between the Student resource and the URL
  - In example above - self, update, delete …
  - **describes** the **action** that's performed with the link
  - It's important that this **value** is intuitive as it **describes** the purpose of the link
- **href** - the **URL** used to perform the action described in rel

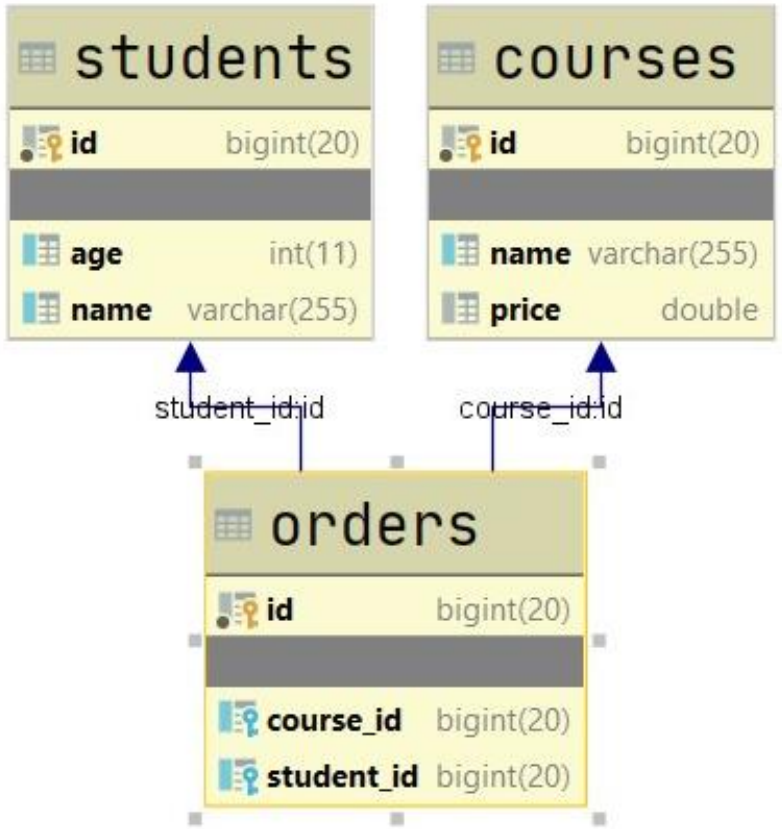
Using HATEOAS in Spring Framework

Software University

- Adding hypermedia links to RESTful responses is something you could implement on your own, but …
- **Spring HATEOAS** makes it **very easy**

```
<dependency>
        <groupId>org.springframework.hateoas</groupId>
        <artifactId>spring-hateoas</artifactId>
        <version>1.1.0.RELEASE</version>
</dependency>
```

Example App DB



- Our example app have small base with some relations between entities
  - We have **Students**, **Orders** and **Courses**

Prepare Controllers to Work

- If we implementing **RepresentationModel <T>** we can added links directly to our entity
- We need two methods from **WebMvcLinkBuilder**, that's why we must import them

```
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;
```

Main Work in Controller

```
...     // Without implementing RepresentationModel<T>

Optional<Student> studentOpt =
        this.studentRepository.findById(id);

    return studentOpt
        .map(s -> ResponseEntity.ok(
                EntityModel.of(s, getStudentLinks(s))))
                .orElse(ResponseEntity.notFound().build());


        // continue to next slide

...
```

Main Work in Controller

```
... // Without implementing RepresentationModel<T>
private Link[] getStudentLinks(Student student) {
    Link self = linkTo(methodOn(StudentsController.class)
        .getStudent(student.getId()))
        .withSelfRel();
    Link orders = linkTo(methodOn(StudentsController.class)
                .getAllOrdersByStudentId(student.getId()))
        .withRel("orders");
    return List.of(self, orders).toArray(new Link[0]);
}
```

Main Work in Controller

```
... // Implementing RepresentationModel<T>
Student student = this.studentService.findById(id);
        student.add(linkTo(methodOn(StudentsController.class)
                .getStudent(student.getId()))
                                        .withSelfRel());
         student.add(linkTo(methodOn(StudentsController.class)
                .deleteStudent(student.getId()))
                                        .withRel("delete"));
        // continue to next slide
...
```

Main Work in Controller

```
... // Implementing RepresentationModel<T>
        student.add(linkTo(methodOn(StudentsController.class)
                .updateStudent(student.getId(), student))
                        .withRel("update"));
        student.add(linkTo(methodOn(OrdersController.class)
                .findAllOrdersByUserId(student.getId()))
                        .withRel("orders"));

        return ResponseEntity.ok(student);
...
```

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

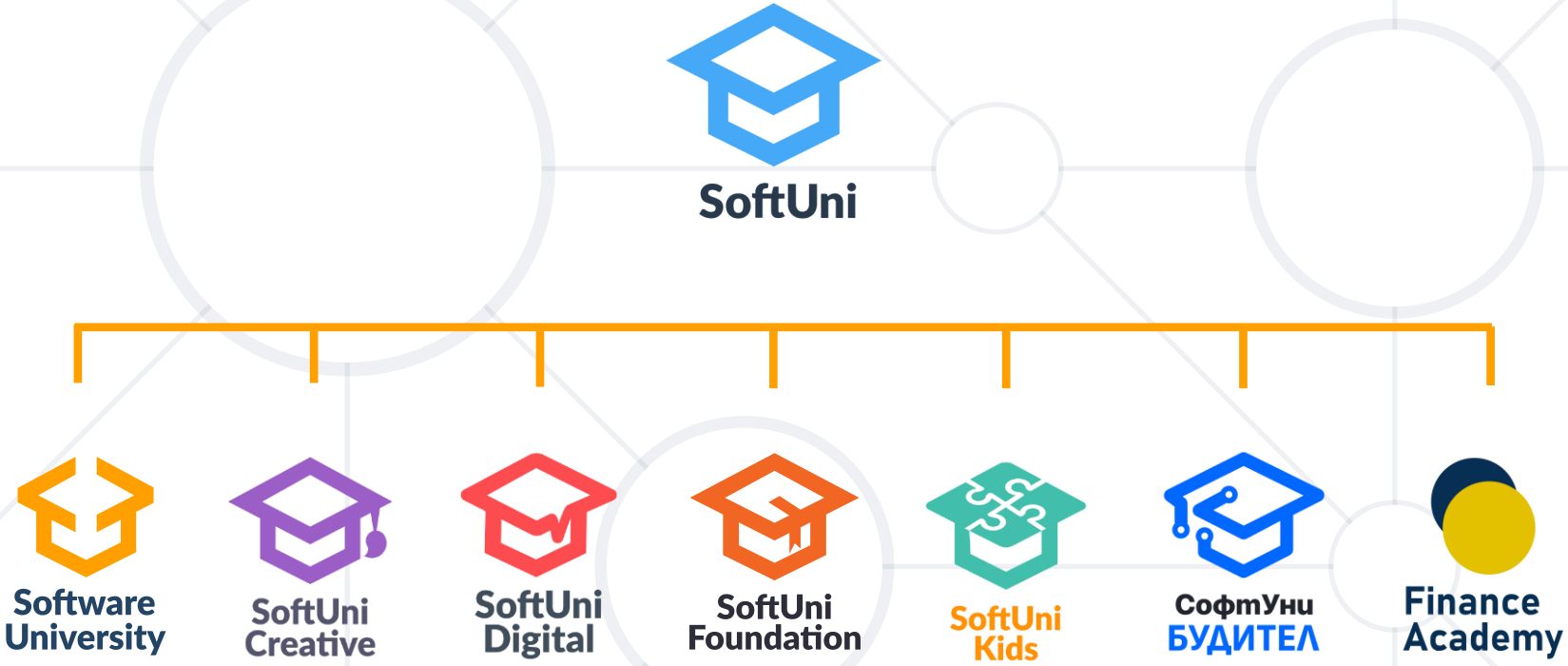
Negatives of Using HATEOAS

- 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

- REST API
  - RESTful Design
  - HTTP GET, POST, PUT, DELETE, PATCH
- REST with Spring
- Rest Client
- HATEOAS

Questions?



SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SUPER HOSTING .BG

createX

Postbank

Решения за твоето утре

DRAFT KINGS

VIVACOM

AMBITIONED

INDEAVR

Serving the high achievers

SOFTWARE GROUP

PHAR VISION

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

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is copyrighted content
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg