

Building REST API with Spring Boot

Building REST API with Spring Boot - Goals

In **28**
Minutes

- **WHY** Spring Boot?
 - You can build REST API WITHOUT Spring Boot
 - What is the need for Spring Boot?
- **HOW** to build a great REST API?
 - Identifying Resources (/users, /users/{id}/posts)
 - Identifying Actions (GET, POST, PUT, DELETE, ...)
 - Defining Request and Response structures
 - Using appropriate Response Status (200, 404, 500, ..)
 - Understanding REST API Best Practices
 - Thinking from the perspective of your consumer
 - Validation, Internationalization - i18n, Exception Handling, HATEOAS, Versioning, Documentation, Content Negotiation and a lot more!



```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

Building REST API with Spring Boot - Approach

In **28**
Minutes

- **1: Build 3 Simple Hello World REST API**
 - Understand the magic of Spring Boot
 - Understand fundamentals of building REST API with Spring Boot
 - @RestController, @RequestMapping, @PathVariable, JSON conversion
- **2: Build a REST API for a Social Media Application**
 - Design and Build a Great REST API
 - Choosing the right URI for resources (/users, /users/{id}, /users/{id}/posts)
 - Choosing the right request method for actions (GET, POST, PUT, DELETE, ..)
 - Designing Request and Response structures
 - Implementing Security, Validation and Exception Handling
 - Build Advanced REST API Features
 - Internationalization, HATEOAS, Versioning, Documentation, Content Negotiation, ...
- **3: Connect your REST API to a Database**
 - Fundamentals of JPA and Hibernate
 - Use H2 and MySQL as databases



```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

What's Happening in the Background?



- Let's explore some **Spring Boot Magic**: Enable Debug Logging
 - **WARNING**: Log change frequently!
- **1: How are our requests handled?**
 - **DispatcherServlet** - Front Controller Pattern
 - Mapping servlets: `dispatcherServlet urls=[/]`
 - Auto Configuration (`DispatcherServletAutoConfiguration`)
- **2: How does **HelloWorldBean** object get converted to JSON?**
 - @ResponseBody + JacksonHttpMessageConverters
 - Auto Configuration (`JacksonHttpMessageConvertersConfiguration`)
- **3: Who is configuring error mapping?**
 - Auto Configuration (`ErrorMvcAutoConfiguration`)
- **4: How are all jars available(Spring, Spring MVC, Jackson, Tomcat)?**
 - **Starter Projects** - Spring Boot Starter Web (`spring-webmvc`, `spring-web`, `spring-boot-starter-tomcat`, `spring-boot-starter-json`)

Social Media Application REST API

- Build a REST API for a Social Media Application
- **Key Resources:**
 - Users
 - Posts
- **Key Details:**
 - User: id, name, birthDate
 - Post: id, description

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

Request Methods for REST API

- **GET** - Retrieve details of a resource
- **POST** - Create a new resource
- **PUT** - Update an existing resource
- **PATCH** - Update part of a resource
- **DELETE** - Delete a resource

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

- **Users REST API**

- Retrieve all Users
 - GET /users
- Create a User
 - POST /users
- Retrieve one User
 - GET /users/{id} -> /users/1
- Delete a User
 - DELETE /users/{id} -> /users/1

- **Posts REST API**

- Retrieve all posts for a User
 - GET /users/{id}/posts
- Create a post for a User
 - POST /users/{id}/posts
- Retrieve details of a post
 - GET /users/{id}/posts/{post_id}

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```


Response Status for REST API

- Return the **correct response status**
 - Resource is not found => 404
 - Server exception => 500
 - Validation error => 400
- **Important Response Statuses**
 - 200 — Success
 - 201 — Created
 - 204 — No Content
 - 401 — Unauthorized (when authorization fails)
 - 400 — Bad Request (such as validation error)
 - 404 — Resource Not Found
 - 500 — Server Error

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

Advanced REST API Features

- Documentation
- Content Negotiation
- Internationalization - i18n
- Versioning
- HATEOAS
- Static Filtering
- Dynamic Filtering
- Monitoring
-

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]
```

REST API Documentation

- Your REST API consumers need to understand your REST API:
 - Resources
 - Actions
 - Request/Response Structure (Constraints/Validations)
- **Challenges:**
 - Accuracy: How do you ensure that your documentation is upto date and correct?
 - Consistency: You might have 100s of REST API in an enterprise. How do you ensure consistency?
- **Options:**
 - 1: Manually Maintain Documentation
 - Additional effort to keep it in sync with code
 - 2: Generate from code

The screenshot displays a REST API documentation interface for a GET endpoint. At the top, a blue bar shows the HTTP method 'GET' and the endpoint path '/jpa/users/{id}/posts'. Below this, a 'Parameters' section contains a table with columns 'Name' and 'Description'. A single parameter is listed: 'id', marked as 'required' with a red star. Its description is 'integer(\$int32) (path)', and there is a text input field containing the value 'id'. The 'Responses' section below features a table with columns 'Code' and 'Description'. A response with status code '200' is shown, described as 'OK'. Underneath, the 'Media type' is set to 'application/hal+json' in a dropdown menu, with a note 'Controls Accept header.' Below that, there are links for 'Example Value' and 'Schema'. The 'Schema' link points to a JSON schema displayed in a dark-themed code block:

```
[ { "id": 0, "description": "string" } ]
```

REST API Documentation - Swagger and Open API

- **Quick overview:**

- **2011:** Swagger Specification and Swagger Tools were introduced
- **2016:** Open API Specification created based on Swagger Spec.
 - Swagger Tools (ex:Swagger UI) continue to exist
- **OpenAPI Specification:** Standard, language-agnostic interface
 - Discover and understand REST API
 - Earlier called Swagger Specification
- **Swagger UI:** Visualize and interact with your REST API
 - Can be generated from your OpenAPI Specification

The screenshot displays the Swagger UI interface. On the left, the OpenAPI specification is shown in a code editor. The specification includes the following details:

- openapi:** "3.0.1"
- info:** { }
- servers:** []
- paths:** {
 - /posts:** {
 - get:** { }
 - post:** { }
 - /posts/{id}:** {
 - get:** { }
 - put:** { }
 - delete:** { }
 - patch:** { }

On the right, the details for the **GET /jpa/users/{id}/posts** endpoint are shown. The **Parameters** section includes a required path parameter **id** of type **integer(\$int32)**. The **Responses** section shows a **200 OK** response with a media type of **application/hal+json**. An example value for the response is provided in a code block:

```
[  
  {  
    "id": 0,  
    "description": "string"  
  }  
]
```

Content Negotiation

- **Same Resource - Same URI**
 - **HOWEVER Different Representations** are possible
 - Example: Different Content Type - XML or JSON or ..
 - Example: Different Language - English or Dutch or ..
- How can a consumer tell the REST API provider what they want?
 - Content Negotiation
- Example: Accept header (MIME types - application/xml, application/json, ..)
- Example: Accept-Language header (en, nl, fr, ..)

```
localhost:8080/users
[
  {
    "id": 1,
    "name": "Adam",
    "birthDate": "2022-08-16"
  },
  {
    "id": 2,
    "name": "Eve",
    "birthDate": "2022-08-16"
  },
  {
    "id": 3,
    "name": "Jack",
    "birthDate": "2022-08-16"
  }
]

<List>
  <item>
    <id>2</id>
    <name>Eve</name>
    <birthDate>1987-07-19</birthDate>
  </item>
  <item>
    <id>3</id>
    <name>Jack</name>
    <birthDate>1997-07-19</birthDate>
  </item>
  <item>
    <id>4</id>
    <name>Ranga</name>
    <birthDate>2007-07-19</birthDate>
  </item>
</List>
```

Internationalization - i18n

- Your REST API might have consumers from around the world
- How do you customize it to users around the world?
 - Internationalization - i18n
- Typically **HTTP Request Header - Accept-Language** is used
 - Accept-Language - indicates natural language and locale that the consumer prefers
 - Example: en - English (Good Morning)
 - Example: nl - Dutch (Goedemorgen)
 - Example: fr - French (Bonjour)
 - Example: de - Deutsch (Guten Morgen)

METHOD GET SCHEME // HOST [":" PORT] [PATH ["?" QUERY]] http://localhost:8080/hello-world-internationalized

QUERY PARAMETERS

HEADERS 1/2 Form BODY XHR does not

Accept-Language : fr x

+ Add header Add authorization

Response

200

HEADERS pretty BODY Bonjour

connection: keep-alive
content-length: 7 bytes

METHOD GET SCHEME // HOST [":" PORT] [PATH ["?" QUERY]] http://localhost:8080/hello-world-internationalized

QUERY PARAMETERS

HEADERS 1/2 Form BODY XHR does not allow

Accept-Language : nl x

+ Add header Add authorization

Response

200

HEADERS pretty BODY Goede Morgen

Content-Type: text/plain; charset=UTF-8
Content-Length: 12 bytes

Versioning REST API

- You have built an amazing REST API
 - You have 100s of consumers
 - You need to implement a breaking change
 - Example: Split name into firstName and lastName
- **SOLUTION: Versioning REST API**
 - **Variety of options**
 - URL
 - Request Parameter
 - Header
 - Media Type
 - **No Clear Winner!**

localhost:8080/v1/person

```
{  
  "name": "Bob Charlie"  
}
```

localhost:8080/v2/person

```
{  
  "name": {  
    "firstName": "Bob",  
    "lastName": "Charlie"  
  }  
}
```

Versioning REST API - Options

- **URI Versioning - Twitter**
 - *`http://localhost:8080/v1/person`*
 - *`http://localhost:8080/v2/person`*
- **Request Parameter versioning - Amazon**
 - *`http://localhost:8080/person?version=1`*
 - *`http://localhost:8080/person?version=2`*
- **(Custom) headers versioning - Microsoft**
 - SAME-URL headers=[X-API-VERSION=1]
 - SAME-URL headers=[X-API-VERSION=2]
- **Media type versioning (a.k.a “content negotiation” or “accept header”) - GitHub**
 - SAME-URL produces=application/vnd.company.app-v1+json
 - SAME-URL produces=application/vnd.company.app-v2+json

```
localhost:8080/v2/person
{
  "name": {
    "firstName": "Bob",
    "lastName": "Charlie"
  }
}
```


Versioning REST API - Factors

- **Factors to consider**
 - URI Pollution
 - Misuse of HTTP Headers
 - Caching
 - Can we execute the request on the browser?
 - API Documentation
 - Summary: No Perfect Solution
- **My Recommendations**
 - Think about versioning even before you need it!
 - One Enterprise - One Versioning Approach

URI Versioning - Twitter

- `http://localhost:8080/v1/person`
- `http://localhost:8080/v2/person`

Request Parameter versioning - Amazon

- `http://localhost:8080/person?version=1`
- `http://localhost:8080/person?version=2`

(Custom) headers versioning - Microsoft

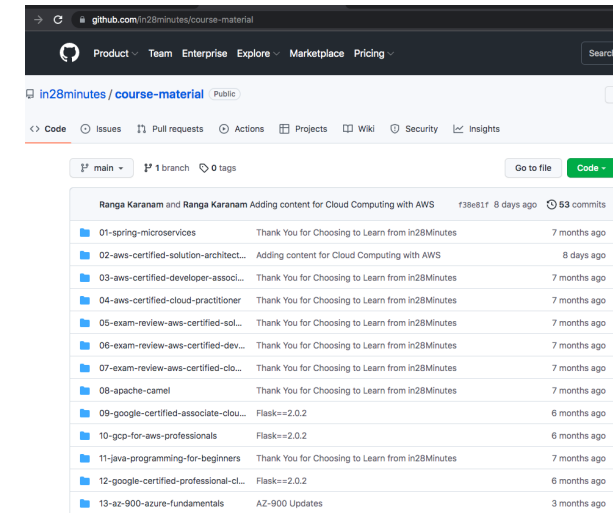
- SAME-URL headers=[X-API-VERSION=1]
- SAME-URL headers=[X-API-VERSION=2]

Media type versioning - GitHub

- SAME-URL produces=application/vnd.company.app-v1+json
- SAME-URL produces=application/vnd.company.app-v2+json

HATEOAS

- Hypermedia as the Engine of Application State (HATEOAS)
- Websites allow you to:
 - See **Data** AND Perform **Actions** (using links)
- How about enhancing your REST API to tell consumers how to perform subsequent actions?
 - HATEOAS
- **Implementation Options:**
 - **1: Custom Format and Implementation**
 - Difficult to maintain
 - **2: Use Standard Implementation**
 - **HAL (JSON Hypertext Application Language):** Simple format that gives a consistent and easy way to hyperlink between resources in your API
 - **Spring HATEOAS:** Generate HAL responses with hyperlinks to resources



```
{
  "name": "Adam",
  "birthDate": "2022-08-16",
  "_links": {
    "all-users": {
      "href": "http://localhost:8080/users"
    }
  }
}
```

Customizing REST API Responses - Filtering and more..

- **Serialization:** Convert object to stream (example: JSON)
 - Most popular JSON Serialization in Java: Jackson
- How about customizing the REST API response returned by Jackson framework?
- **1:** Customize field names in response
 - @JsonProperty
- **2:** Return only selected fields
 - **Filtering**
 - Example: Filter out Passwords
 - **Two types:**
 - **Static Filtering:** Same filtering for a bean across different REST API
 - @JsonIgnoreProperties, @JsonIgnore
 - **Dynamic Filtering:** Customize filtering for a bean for specific REST API
 - @JsonFilter with FilterProvider

```
localhost:8080/filtering-list
[
  {
    "field2": "value2",
    "field3": "value3"
  },
  {
    "field2": "value5",
    "field3": "value6"
  }
]

localhost:8080/filtering
{
  "field1": "value1",
  "field3": "value3"
}
```

Get Production-ready with Spring Boot Actuator

In **28**
Minutes

- **Spring Boot Actuator:** Provides Spring Boot's production-ready features
 - Monitor and manage your application in your production
- **Spring Boot Starter Actuator:** Starter to add Spring Boot Actuator to your application
 - `spring-boot-starter-actuator`
- Provides a number of endpoints:
 - **beans** - Complete list of Spring beans in your app
 - **health** - Application health information
 - **metrics** - Application metrics
 - **mappings** - Details around Request Mappings
 - and a lot more



Explore REST API using HAL Explorer

In **28**
Minutes

- **1: HAL (JSON Hypertext Application Language)**
 - Simple format that gives a consistent and easy way to hyperlink between resources in your API
- **2: HAL Explorer**
 - An API explorer for RESTful Hypermedia APIs using HAL
 - Enable your non-technical teams to play with APIs
- **3: Spring Boot HAL Explorer**
 - Auto-configures HAL Explorer for Spring Boot Projects
 - `spring-data-rest-hal-explorer`



