Rest, Spring Boot web

Juraj Kollár

Client-server architecture

- A computing model where **clients** request services and **servers** provide them over a network.
- Client initiates communication by sending requests (web browser)
- Server waits for requests, processes them and sends responses (web server, database server)
- The system is written as client-server
- Examples:
  - Web browser – web server
  - Email client – email server
  - Game client – game server

Java Web Services

- Former de-facto standard in Java
- Based on SOAP protocol
- XML is the only format
- Highly standardized (WSDL, XML Schema, WS-security)
- Heavy use of JAX-WS specification (Java API for XML Web Services)
- Too heavy, too verbose, too complex, cumbersome to work with

RESTful Web Services

- More recent approach to same problem (client-server architecture)
- Based on existing well known & well established HTTP protocol
- Format typically JSON (XML or other possible)
- Not tied to Java
- More lightweight, not so strict about standards
- Some optional standardization (JSON Schema, OpenAPI)
- In Java world standardized by JAX-RS (Java API for RESTful Web Services)
- In Spring Boot JAX-RS is not usually followed

What is REST?

- REST = REpresentational State Transfer
- An architectural style for networked applications
- Introduced by Roy Fielding in 2000

What are RESTful Web Services?

- Web services that follow REST principles
- Use standard HTTP methods
- Lightweight, stateless, and easy to use
- Application (or micro-service) has some functionality, logic
- To outside world it exposes its functionality as RESTful API (its functions are REST endpoints)
- Application to application “talks” via their APIs
- Front-end application calls back-end’s API (decoupling)

Basic Components

- Client – sends requests
- Server – returns responses
- Resources – identified using URLs (e.g., /users/123)

Common HTTP Methods

- GET – Read/Retrieve data
- POST – Create a new resource
- PUT – Update an existing resource (fully)
- DELETE – Remove a resource
- PATCH – special kind of update where you specify only those attributes of resource that you want to update

RESTful URL Example

- Base URL: https://api.example.com
- GET /users – Get all users
- GET /users/1 – Get user with ID 1
- POST /users – Create a new user

REST URL detals

- https://api.example.com:8080/myapp/api/v1/users/12345?active=true&sort=desc

Part	Description
https://	**Protocol** – Secure HTTP used to access the resource
api.example.com	**Host** – Domain name or IP address of the server
:8080	**Port** – Optional; specifies the port the server listens on (default is 80 or 443 for HTTPS)
/myapp	**Application Prefix (Context Path)** – Entry point to the web application on the server
/api/v1	Optional API versioning
/users/12345	**URL Path** – Resource path: /users is the collection, 12345 is the **path parameter** (a specific user ID)
?active=true&sort=desc	**Query Parameters** – Extra info for filtering/sorting the response (not part of the resource path)

Resource Representation

- Data usually in JSON or XML format
- Example JSON:
- { "id": 1, "name": "Alice", "email": "alice@example.com" }

REST call

- A client (web browser, API client, CURL…) makes a RESTful HTTP request:
- `GET https://api.example.com:8080/myapp/api/v1/users/12345?active=true&sort=desc`
- Server (back-end application) handles the request and responds
- HTTP status: 200 OK
- Response body (as JSON):
- `{ "id": 12345, "name": "Alice", "email": alice@example.com, " active " : true }`

HTTP headers

- An HTTP request (and response) can have HTTP headers which are some additional information or meta-information about the request
- Good to know headers (often confused):
- Content-type – when client is sending request this is the MIME type of what is being sent (in POST, PUT, PATCH requests), when server is sending it is the actual MIME type of the response
- Accept – client sends what MIME type it will understand (accept)
- Well known content types: application/json, application/xml, text/plain, application/pdf, multi-part/form-data

Statelessness

- Each request is independent
- No session or memory of previous requests
- Benefits: simplicity and scalability

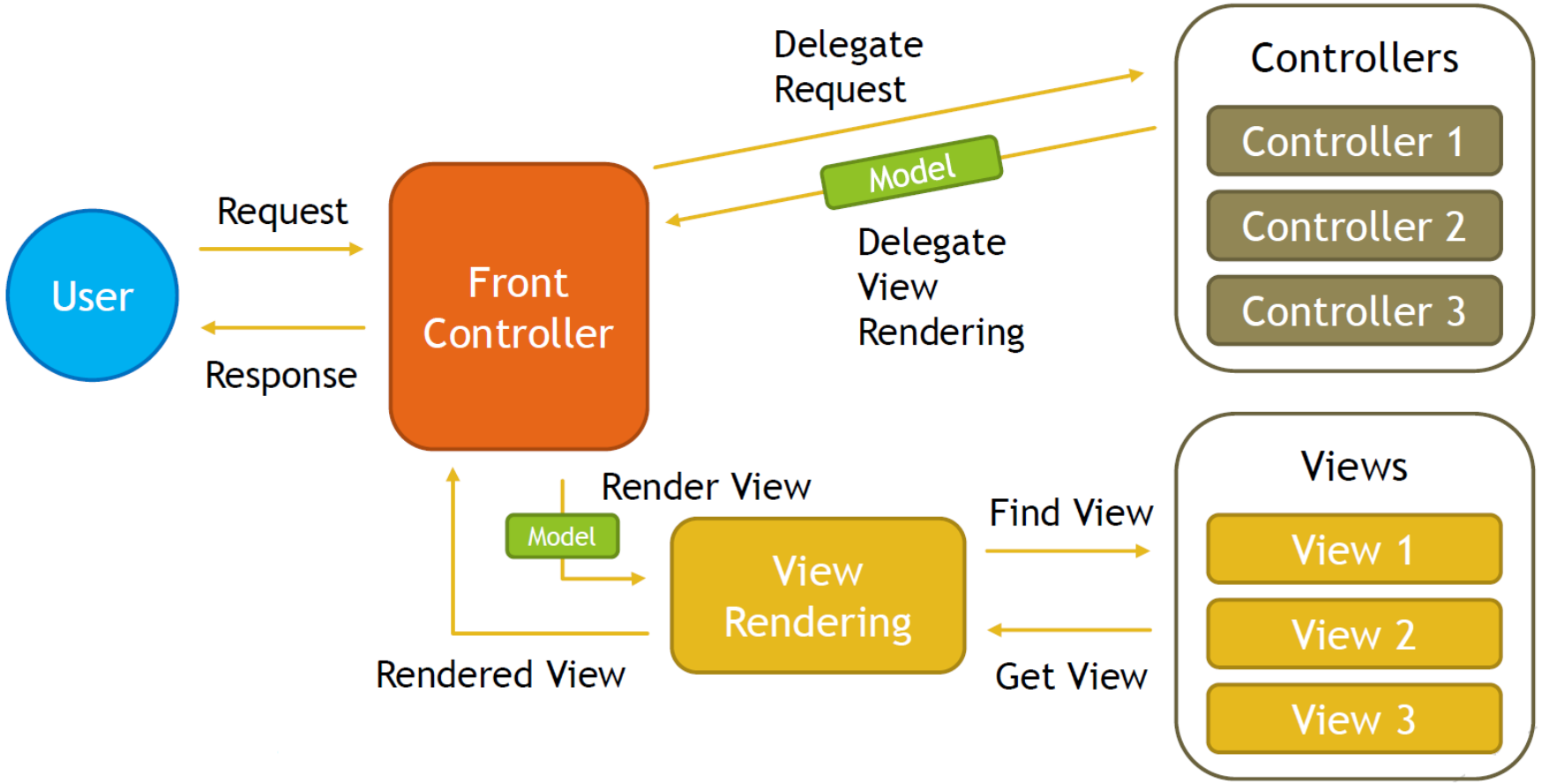
How this is done in Spring

- To handle REST API requests in Spring you need a Controller
- What is a Controller?
- It is one part of the Model-View-Controller design pattern
- What is MVC? See next

MVC (Model-View-Controller)

- It is design pattern which divides application into 3 interconnected component types:
- Model
  - data access, data structure, business logic, CRUD
  - Spring data jpa or other, repositories, custom repositories
- View
  - data representation, multiple representation of the same data
  - Thymeleaf, Jsp and jstl or other
- Controller
  - access request from user, issues command to model, modifies model, decides about view to use
  - Controller, rest controller classes
- It brings: increase code cohesion, increase code usability, reduce coupling, increase extensibility

Front controller



Controller

- Controller is annotation coming from Spring
- It marks a class as a Spring component with specific meaning
- It is a Controller from MVC pattern that can handle requests
- @Controller
- public class UsersController {}
- @RestController – for handling REST requests (unions @Controller with @ResponseBody)
- Behind the scenes Spring Boot scans for such classes and configures necessary things to work

RESTful annotations

- @RequestMapping – class-level and method level annotation for mapping web requests onto methods
- Method-level mapping annotations
- @GetMapping (for mapping GET requests)
- @PostMapping (for mapping POST requests)
- @PutMapping (for mapping PUT requests)
- @DeleteMapping (for mapping DELETE requests)
- @PatchMapping (for mapping PATCH requests)

Basic REST Controller Example

```
@RestController
@RequestMapping("/api/hello")
public class HelloController {

    @GetMapping
    public String sayHello() {
        return "Hello, world!";
    }
}
```

- @GetMapping -> responds to HTTP GET
- Returns ”Hello, world!” directly as HTTP response body

REST Controller with Path Variables

```
@GetMapping("/users/{id}")
public String getUser(@PathVariable int id) {
    return "User ID: " + id;
}
```

- Call example: GET /api/users/42 -> “User ID: 42”

REST Controller using Query Parameters

```
@GetMapping("/search")
public String searchUser(@RequestParam String name) {
    return "Searching for: " + name;
}
```

- Call example: GET /api/search?name=Alice -> ”Searching for: Alice”

REST Controller returning JSON Objects

```
@GetMapping("/user")
public User getUser() {
    return new User(1, "Alice");
}
record User(int id, String name) {}
```

- Spring automatically converts objects -> JSON using Jackson
- Response: { "id": 1, "name": "Alice" }

REST Controller handling POST Requests

```
@PostMapping("/users")
public String createUser(@RequestBody User user) {
    return "Created user: " + user.name();
}
```

- @RequestBody maps JSON → Java object
- Example request: { "id": 5, "name": "Bob" }

Rest Controller returning ResponseEntity

```
@PostMapping("/users")
public ResponseEntity<String> createUser(@RequestBody User user)
{
    return ResponseEntity
            .status(HttpStatus.CREATED)
            .body("User created: " + user.name());
}
```

- Use ResponseEntity to customize status, headers, and body.

Spring Boot Dependencies

- Let’s use Spring Boot version 3.5.5
- Your Maven POM module needs to extend Spring Boot’s parent:

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.5.5</version>
</parent>
```

Spring Boot Dependencies (2)

- Spring boot is modular and each “module” is a dependency called a “starter”
- Such starter module will auto-configure what is needed with meaningful defaults (convention over configuration)
- We need to run a web server to serve our REST requests, therefore we need spring-boot-starter-web
- This will bring embedded web server – Tomcat by default

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

Spring Boot Application

- Use annotation @SpringBootApplication
- Enables auto-configuration
- Makes a component scan
- Run your Spring Boot application with main method:

```
@SpringBootApplication
public class Application {

	public static void main(String[] args) {
		SpringApplication.run(Application.class, args);
	}
}
```

API

- API = Application Programming Interface
- Pedals, steering Wheel, shift stick
- In REST context it is the set of API endpoints
- Front-end to back end talks via back-end’s API
- Applications talk to each other via APIs

The problem with API

- Developers write APIs, but how do others know how to use them?
- Confusion over:
  - What endpoints exist?
  - What data to send/receive?
  - What errors can occur?
- Manual documentation gets outdated!
- (Advanced) API versioning

OpenAPI

- OpenAPI – standard for describing RESTful API
- Swagger – Tools that work with API
  - It was created before OpenAPI was standardized, so they are sometimes used interchangeably

OpenAPI example

```
paths:
 /users:
  get:
   summary: Get all users
   responses:
    '200':
     description: A list of users
```

- Written in YAML (more often) or JSON
- Tells us:
  - Endpoint – /users
  - Method (GET)
  - Expected response

Swagger tools

- Swagger UI
  - Converts OpenAPI files into a website
  - Click to test endpoints!
  - Live demo: https://petstore.swagger.io
- Swagger Editor
  - Write and edit OpenAPI specs
- Swagger codegen
  - Generate client / server code

Swagger in Spring

- Let’s you generate documentation from your Spring REST Controllers
- You need to add dependency to springdoc-openapi-starter-webmvc-ui
- It will include a hidden Controller that will provide endpoint which will show you Swagger generated from your Controllers
- You need to tell it on which URL it will serve the endpoint
- springdoc.api-docs.enabled=true
- springdoc.api-docs.path=/api-docs
- springdoc.swagger-ui.enabled=true
- springdoc.swagger-ui.path=/swagger

Customizing web server

- Change context path - server.servlet.context-path=/fots
- Change port - server.port=8081

Error handling in Spring REST