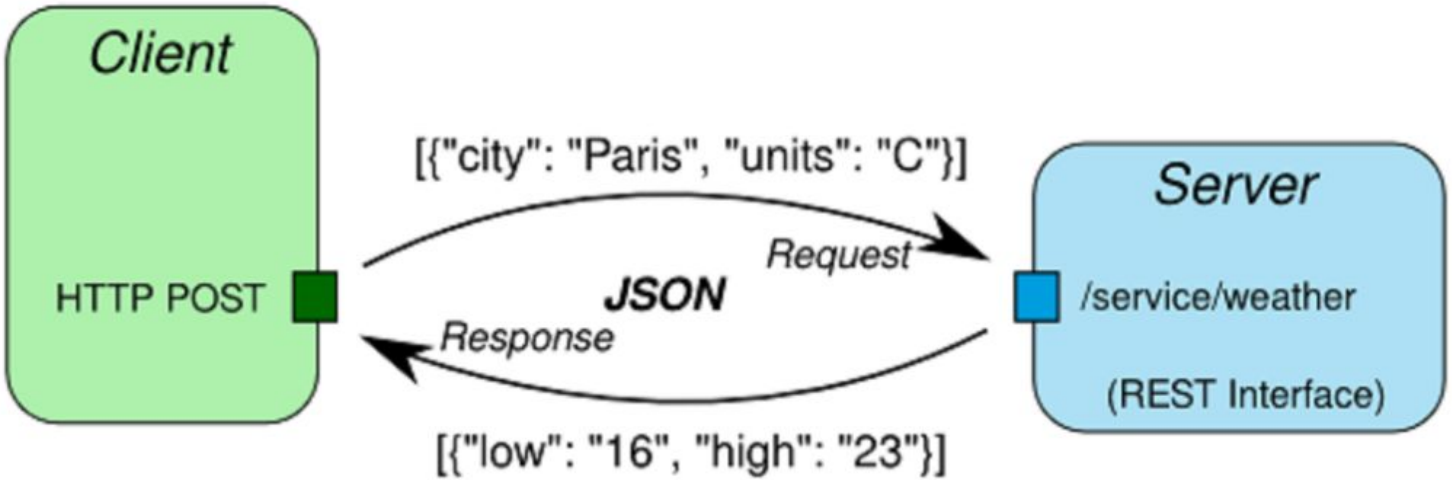
Spring Boot - REST services



REST = REpresentational State Transfer

REST principles were defined by **Roy Fielding** in his PhD dissertation in 2000.

Key terms: **API, Client, Resource, RESTful web application, Endpoint, HTTP method, Server.**

When a RESTful API is called, the server will transfer to the ***client*** a ***representation of the state*** of the requested ***resource***.

This representation of the state can be in the following formats: **JSON, XML or HTML.**

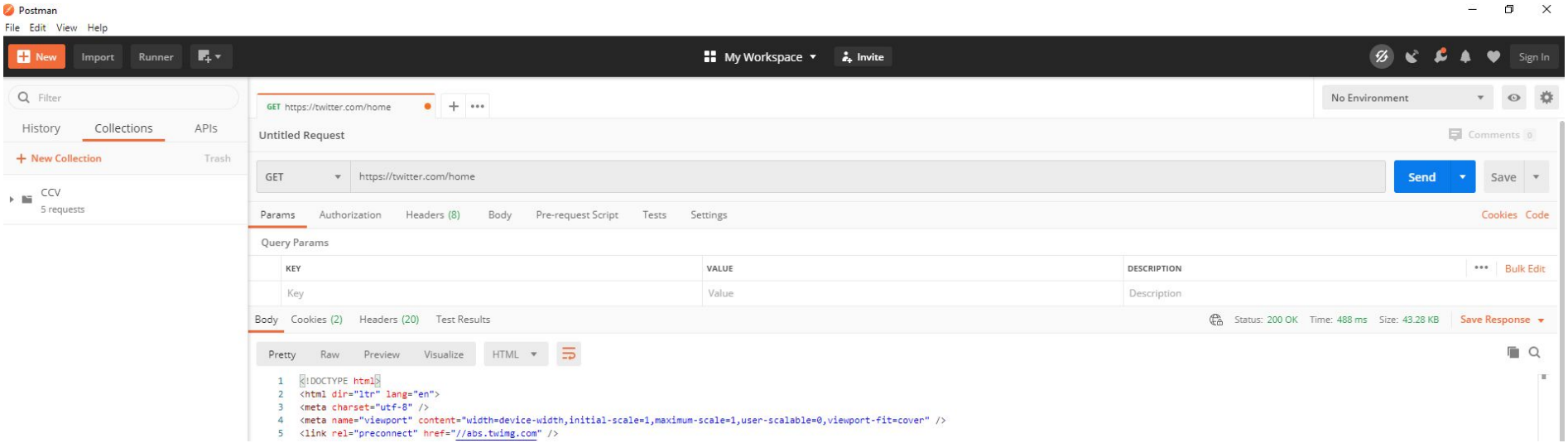
REST API methods: GET, POST, PUT, DELETE

For an API to be RESTful, it has to implement 6 constraints: **Uniform interface, Client - Server separation, Stateless, Layered system, Cacheable, Code-on-demand (optional)**.

Postman (I)

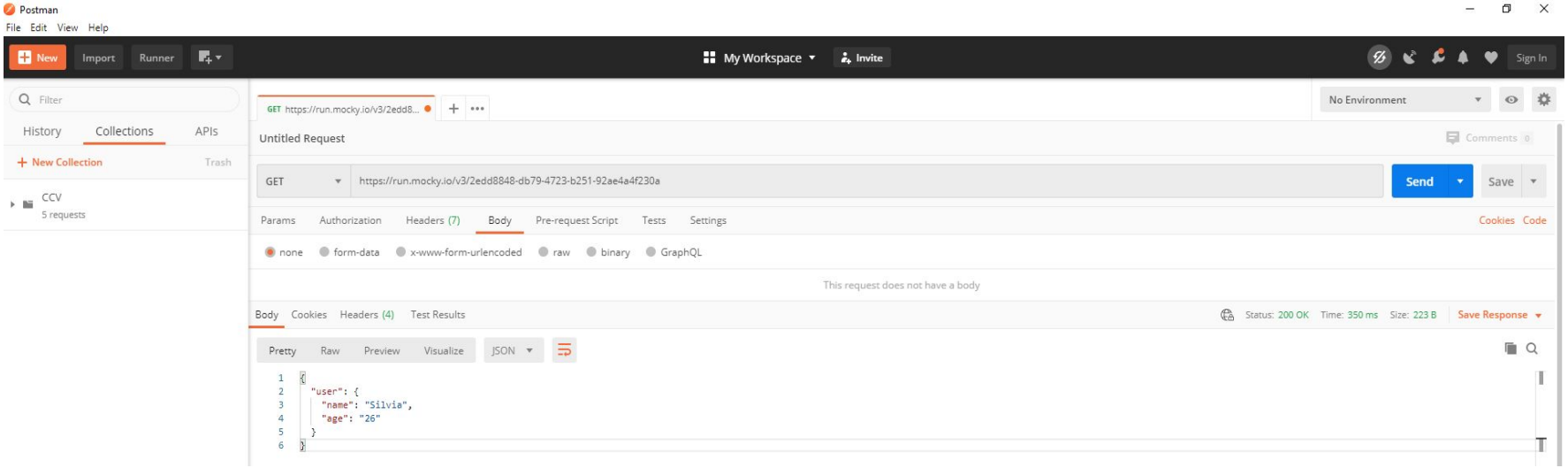
https://www.postman.com/

- Define complex requests => quickly send REST requests directly.
- Inspect Responses.
- We can see the full list of features here: https://www.postman.com/postman-features/.



Postman (II)

For testing purpose we will define mock endpoints - https://designer.mocky.io/design and test them in Postman



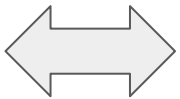
Frameworks for developing REST APIs in Java

- Spring MVC Framework - we will discuss about it in our session
- JAX-RS - more info: https://www.baeldung.com/jax-rs-spec-and-implementations
- Restlet - more info: https://livebook.manning.com/book/restlet-in-action/chapter-1/31
- Jersey - more info: https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/getting-started.html
- RESTEasy - more info: https://www.baeldung.com/resteasy-client-tutorial
- Apache CFX - more info: https://www.baeldung.com/introduction-to-apache-cxf

Spring MVC Framework

- Spring MVC is a complete HTTP oriented MVC framework managed by the Spring Framework and based in Servlets.
- Contains classes annotated with **@Controller**, where we can implement methods accessed using different HTTP requests. The response from a web application is generally view (HTML + CSS + JavaScript) => The job of @Controller is to provide the model object and find a view.
- REST API just returns data in form of JSON or XML because most of the REST clients are programs. This can also be done with traditional @Controller and use @ResponseBody annotation but since this is the default behavior of RESTful Web services, Spring introduced **@RestController** which combined the behavior of @Controller and @ResponseBody together.

```
@Controller
@ResponseBody
public class TestController {
  // your logic
}
```



```
@RestController
public class TestController {
  // your logic
}
```

Spring Boot (I) - Overview

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run" - https://spring.io/projects/spring-boot.

- Features
  - Create stand-alone Spring applications
  - Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
  - Provide 'starter' dependencies to simplify build configuration
  - Automatically configure Spring and 3rd party libraries whenever possible
  - Provide production-ready features such as metrics, health checks, and externalized configuration
  - No code generation and no requirement for XML configuration

Spring Boot (II) - Setup

We will start a Spring Boot application by using: https://start.spring.io/

Things to notice:

- The Spring Boot starter web dependency (build RESTful web applications using Spring MVC and using Apache Tomcat as the default embedded container).

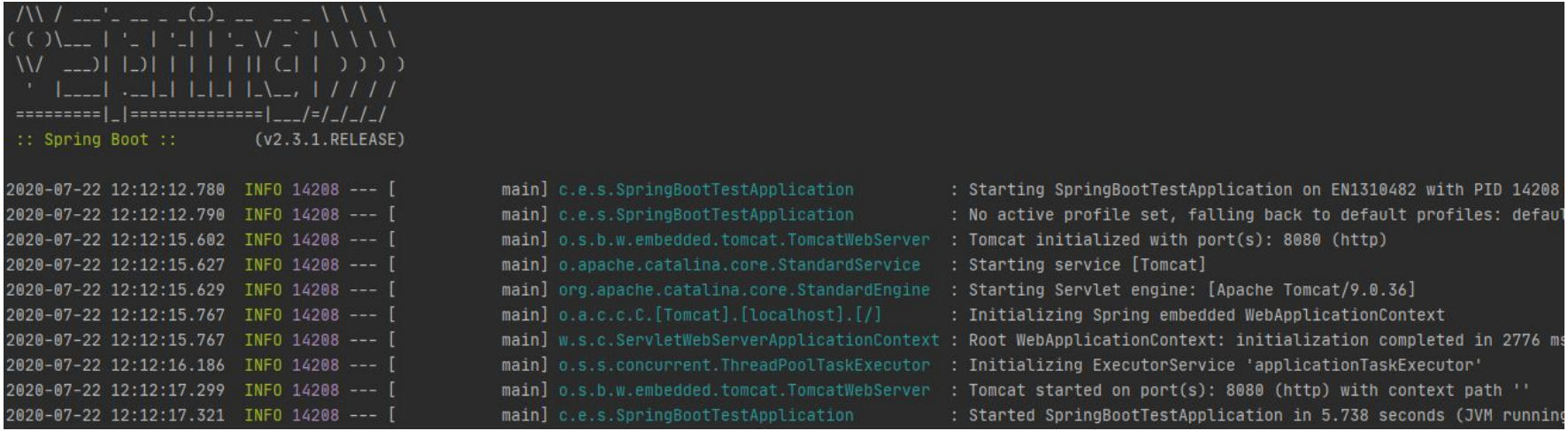
```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

- @SpringBootApplication annotation -> more info: https://docs.spring.io/spring-boot/docs/2.1.13.RELEASE/reference/html/using-boot-using-springbootapplication-annotation.html
- Starting the Spring Boot application by creating a main class and calling the run method from SpringApplication (this method starts whole Spring Framework):

```
SpringApplication.run(SpringBootTestApplication.class, args);
```

Spring Boot (II) - Setup

We have now a web application running on port 8080:



We should note that at this point we have no resources available and no resources.

Spring Boot (III) - Setup

Let’s build our first endpoint:

- An API that doesn’t consume anything and returns the text: “Hello world!”
- **Notice**:
  - **@RestController** annotation on the controller class
  - **@GetMapping** annotation
  - The param **produces** from the @GetMapping annotation
  - The **media type** used: MediaType.APPLICATION_JSON_VALUE
  - In addition:
    - **@RequestMapping** annotation on the controller class
    - The param value from the **@GetMapping** annotation
    - **@PathVariable**("value")
    - **@RequestBody**

@RestController

- Applied to a class to mark it as a **request handler.**
- Used to create RESTful web services using Spring MVC.
- Takes care of mapping request data to the defined request handler method. Once response body is generated from the handler method, it converts it to JSON or XML response.

```
@RestController
public class TestApplicationController {
}
```

@RequestMapping

- Used to map web requests to Spring Controller methods;
- Can be applied to a @RestController annotated class and at method/endpoint level
- More info: https://www.baeldung.com/spring-requestmapping

```
@RestController
@RequestMapping("/testApplication")
public class TestApplicationController {

   @RequestMapping(value = "/hello/{name}",
           method = GET,
           headers = "Accept=application/json")
   public String hello(@PathVariable("name") String name) {
       return "Hello " + name + "!";
   }
}
```

@GetMapping

- Shortcut Annotation - method-level composed annotation to make easier the handling @RequestMapping

```
@RestController
@RequestMapping("/testApplication")
public class TestApplicationController {

   @GetMapping(value = "/hello/{name}", produces = MediaType.APPLICATION_JSON_VALUE)
   public String hello(@PathVariable("name") String name) {
       return "Hello " + name + "!";
   }
}
```

@PostMapping

- Shortcut Annotation - method-level composed annotation to make easier the handling @RequestMapping

```
@RestController
@RequestMapping("/testApplication")
public class TestApplicationController {

   @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
   public UserDto create(@RequestBody UserDto userDto) {
       // logic to save user
       return userDto;
   }
}
```

@PutMapping

- Shortcut Annotation - method-level composed annotation to make easier the handling @RequestMapping

```
@RestController
@RequestMapping("/testApplication")
public class TestApplicationController {

   @PutMapping(consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
   public UserDto update(@RequestBody UserDto userDto) {
       // logic to update user
       return userDto;
   }
}
```

@DeleteMapping

- Shortcut Annotation - method-level composed annotation to make easier the handling @RequestMapping

```
@RestController
@RequestMapping("/testApplication")
public class TestApplicationController {

   @DeleteMapping(value = "/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
   public Boolean remove(@PathParam("id") Long id) {
       // logic to delete user
       return true;
   }
}
```

RESTful application using Spring Boot (I)

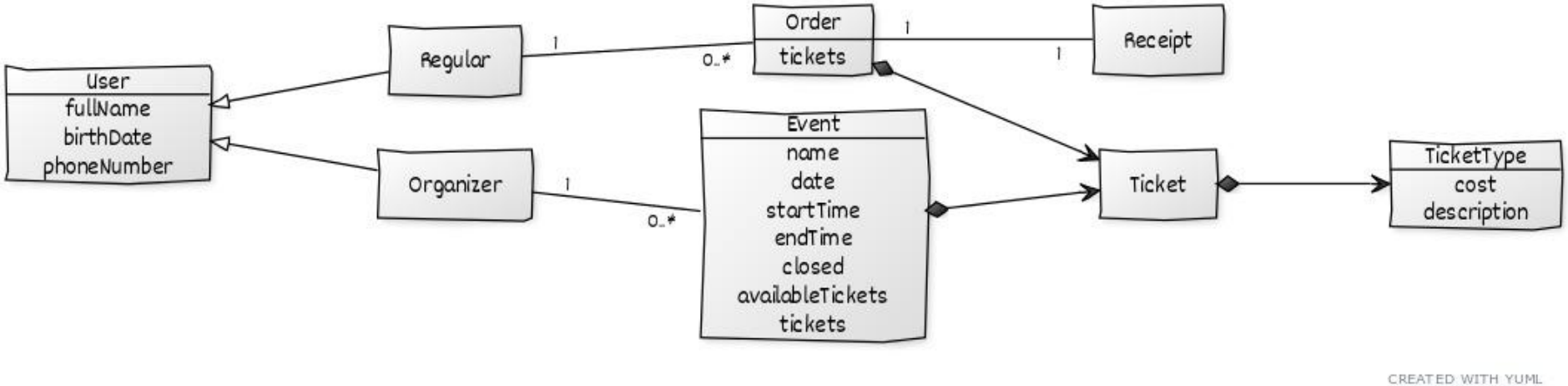
What we want to develop: A software for selling tickets online for different events (e-ticket)

Business use cases:

1. The software will allow two types of users: organizer and regular user;
2. As an organizer I want to be able to add an event and to customize the configuration for that event such that:
    a. I can add different types of tickets by specifying the number of available tickets, the cost of the ticket (in RON) and a description for the ticket;
    b. I can add information for the event such as the name, date, the start and end time of the event, if it's indoor or outdoor, the location and additional description of the event;
3. As an organizer I want to be able to close an event if necessary and to return the money for the tickets to the users;
4. As a regular user I want to be able to see all ongoing events;
5. As a regular user I want to be able to see all types of tickets for an event;
6. As a regular user I want to be able to buy tickets for an event;
7. As a regular user I want to be able to see all the tickets bought for upcoming events.

RESTful application using Spring Boot (II)

- Class diagram built with: https://yuml.me/diagram/scruffy/class/draw



RESTful application using Spring Boot (III)

Endpoints:

1. **Users**:
    a. Create user (POST)
    b. Update user (PUT)
    c. Get user (GET)
    d. Get users (GET)
    e. Delete users (DELETE)
2. **Events**
    a. Create event (POST) - this includes logic for adding available tickets (together with ticket types) for the event
    b. Update event (PUT)
    c. Get event (GET)
    d. Get events (GET)
    e. Delete events (DELETE)
3. **Tickets**
    a. Get tickets (GET) - for specific user
4. **Orders**
    a. Create order (POST) - this is the endpoint for buying tickets -> and includes updating also the event (decrement available tickets for the event) and creating the ticket (assigned to an event and corresponding to a user)