MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Shongo Reservation System Backend REST API

Bachelor's Thesis

FILIP KARNIŠ

Brno, Spring 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Shongo Reservation System Backend REST API

Bachelor's Thesis

FILIP KARNIŠ

Advisor: RNDr. Miloš Liška, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Filip Karniš

Advisor: RNDr. Miloš Liška, Ph.D.

Acknowledgements

I would like to thank my advisor RNDr. Miloš Liška, Ph.D., for leadership and valuable comments during the creation of my work. I would also like to thank my family and friends for their support and the possibility to write this thesis without interruption.

Abstract

The main goal of this bachelor thesis is to design and implement REST API for the reservation system Shongo, developed and operated by the CESNET association. The API will expose the Shongo back-end (Controller module) to a new front-end, which is a goal of another thesis.

This paper first introduces the Shongo system, describes its current state, and discusses necessary changes in its architecture. After describing the used technologies, the accomplished implementation is presented, and the resulting API is defined. The implemented REST API is ready to expose the Shongo system to the higher layers, especially to the new front-end. The result is a complete separation of the Shongo system's front-end and back-end.

Keywords

reservation system Shongo, REST API, CESNET, Java, Spring, videoconference

Contents

List of Figures

Introduction

Videoconferences have become a big topic in the last few years. With the globalization of the companies and an increasing number of people working remotely, the demand for methods that can be used for remote communication is growing in direct proportion. Especially now, when the world pandemic stormed the lives of all mankind and limited the physical contact between people, they can appreciate the impact of this technology. The children and students can learn from their homes by communicating with their teachers "live" in the videoconferences mentioned earlier. Jobs that require communication considerably but do not require human contact can take full advantage of this technology. However, the technology is bound to the available resources used for the videoconferences.

The academic community, as well as any large company, has a certain amount of resources at its disposal. For effective usage, the management of these resources is required. For this purpose, the CESNET association created the Shongo system. The Shongo system manages available resources, makes reservations for these resources, and notifies participants about an upcoming videoconference.

This thesis was assigned to implement REST API for the Controller module (Section 1.2.1) of the Shongo system. The rest of the introduction focuses on describing this work's chapters.

In Chapter 1, the current reservation system Shongo will be introduced. Furthermore, this chapter discusses the current state of the Shongo architecture, its flaws, and how the architecture will change after the work on this thesis is done.

Next, the technologies used to finalize the assignment of this thesis have to be adequately explained. This explanation and a brief introduction to RESTful API can be found in Chapter 2.

When the reader is acquainted with the technologies used, the thesis continues to explain how they were used to complete the assignment in Chapter 3. This chapter describes the structure of the code added to the Shongo system — the configuration, implementation and documentation of the REST API as the assignment requests.

Last but not least, the final API is described in Chapter 4, so the reader can imagine how the client can use the implemented REST API to interact with the Shongo system described in Chapter 1.

Finally, Chapter 5 summarizes the thesis and presents the thoughts about what can be done next.

1 Reservation System Shongo

This chapter introduces the subject of this work, the reservation system Shongo. Information for this chapter was drawn mainly from the Shongo API documentation [1]. However, this documentation is outdated and is only partially correct. Since the documentation was introduced, multiple theses and other works have contributed to the development of the Shongo system. Thus, much information has been drawn from these, especially the theses concerning inter-domain extension [2] and redesign of the system [3].

Currently, at version v0.9.3, Shongo is a generic system used for managing resources. Its primary function is to manage resource reservations for virtual videoconferencing meetings and any physical resources, such as physical meeting rooms, vehicles or parking places. It allows administrators to define available videoconferencing resources (Pexip or Adobe Connect Server) and physical resources (rooms, vehicles, parking places). Afterwards, users can request a reservation for the available resources. Furthermore, the system provides remote control and recording of ongoing virtual meetings using the Cisco TCS recording server, user notifications, and detailed resource usage statistics [4].

The system is developed as an open-source project available at Github repository[1] by CESNET. CESNET is an association of universities and the Academy of Sciences of the Czech Republic. The association offers a wide range of services and develops a national electronic infrastructure for science and education. The e-infrastructure includes a computer network, computing grids, data repositories, and collaborative environments [5].

The system is currently deployed in these domains:

- `https://meetings.cesnet.cz/` – reservation system for H.323/SIP and Adobe Connect virtual rooms within CESNET videoconferencing infrastructure
- `https://meetings.cesnet.cz/ceitec/` – reservation system for physical rooms of CEITEC — Central European Institute of Technology

1. `https://github.com/shongo/shongo`

- `https://meetings.cesnet.cz/cuni/` – reservation system for Adobe Connect virtual rooms of Charles University[2].

Users can log in using their profile from any organization belonging to the Czech academic identity federation[3]. After logging in, they can freely use the system and request reservations for available resources.

1.1 Reservations

As already mentioned, the primary task of the Shongo system is to manage reservations. This section describes the fundamental concepts behind the reservation logic.

Physical reservations are very straightforward:

1. A user requests a reservation of a physical resource.
2. If the request is permitted, he can use the physical resource for a specified period.

On the other hand, virtual meeting reservations are more complicated. Virtual rooms represent the allocation of a specific virtual resource. There are two types of virtual rooms:

- **Permanent rooms** are devoted to long-term reservations. This type reserves virtual resource — name and a unique identifier of room (URL, phone number, SIP URI) — but without capacity reservation for the chosen resource (period, periodicity, number of participants). The user has to book capacity separately after creating the permanent room.
- **One-time (ad-hoc) rooms** are intended for one-time reservations of virtual resources. The Controller generates a name and unique identifier according to available values from the range configured in the resource specification. That results in a new room name and identification for each reservation. A capacity is reserved

2. `https://cuni.cz/`
3. Czech academic identity federation [online]. 2017 [visited on 2022-03-09]. Available from: `https://www.eduid.cz/`.

when the room is created, and the room is deleted after the requested time slot ends.

After reserving the room, the user can set up roles for other users or groups (owner, user, reader) and add other participants with specific permissions (administrator, presenter, participant). Suppose the reservation is permitted (sufficient capacity is available). In that case, the user who created the reservation and the participants added by the user can join the videoconference during the requested time slot.

1.2 Architecture

Explaining what parts of the system were added, modified, or replaced without fundamental knowledge of the system architecture would be confusing and perhaps impossible to comprehend. This section explains the system's architecture and components with a closer look.

The system's architecture is composed of a few separate modules. Their interconnections are shown in Figure 1.2.

1.2.1 Controller

The controller module is the heart of the Shongo system. Its main function is to plan requested reservations (*Scheduler* sub-component) and allocate necessary resources. Its secondary roles are informing users about their reservations (*NotificationManager* sub-component) and administrating remote virtual resources (*Executor* sub-component) through connected connectors. Furthermore, last but not least, it is responsible for saving this information into the relational database *PostgreSQL*.

These functionalities are made available for other components with the *API* (Application programming interface) via *XML-RPC* protocol.

1.2.2 Connector

The connector is a component that is responsible for establishing, maintaining and monitoring connections to remote virtual resources (Pexip, H.323 MCU, Adobe Connect, Zoom) and utilizing their functionalities. First and foremost, that includes creating, maintaining and deleting

virtual meeting rooms based on users' reservations. The connector contains implementations of *Device Agent*s, and each *Device Agent* provides a specific device (e.g. Adobe Connect, Pexip) features to the Controller using the device's *API*. Moreover, it enables the system to save data (screenshots, for example) from the device to the local file system.

The Controller communicates with the connector using the *JADE* (Java Agent DEvelopment) framework, which enables the connection of a single connector to multiple controllers. This can be done, for example, in order to distribute the load.

1.2.3 Authentication Server

The authentication server authenticates incoming requests for the controller component, authenticates users, finds users from various identity providers and obtains information about users for the web client. Besides that, it also manages the authentication layer of the web conference system Adobe Connect. This layer uses the user's information gained after logging in via system Shibboleth[4].

1.2.4 Web Client

The web client is the system's primary interface for users. The *Spring-MVC* and *AngularJS* frameworks generate web pages with data gained from the Controller's services. That results in server-side rendering as in typical Model-View-Controller (MVC) architecture illustrated in Figure 1.1.

Thanks to this component, users can authenticate via OpenID Connect [5] and request a reservation of any available resource comfortably via the web browser.

1.2.5 Command-line Client

The command-line client is the Controller's additional interface used for administration. It allows system administrators to manage re-

4. Shibboleth, `https://www.shibboleth.net/`
5. OpenID Connect, `https://openid.net/connect/`

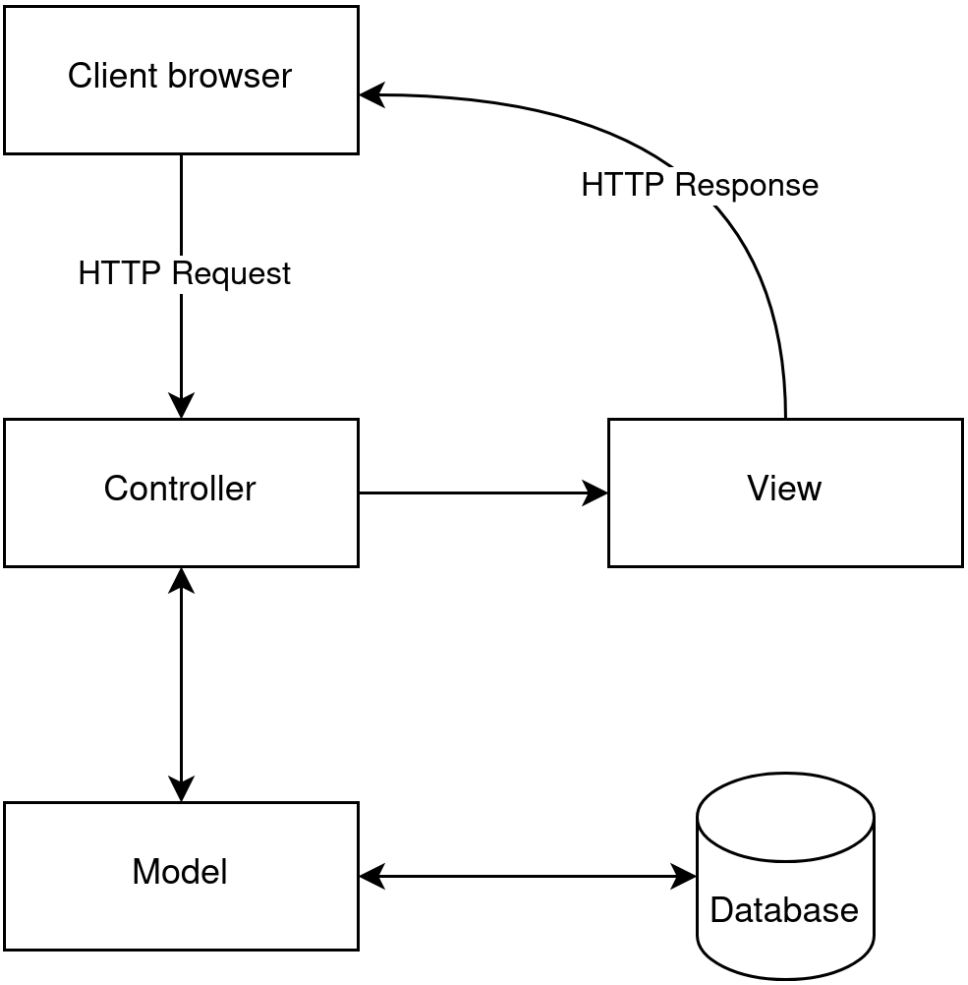

Figure 1.1: Model-View-Controller architecture

sources, users, groups and reservation requests which is not possible using the web client.

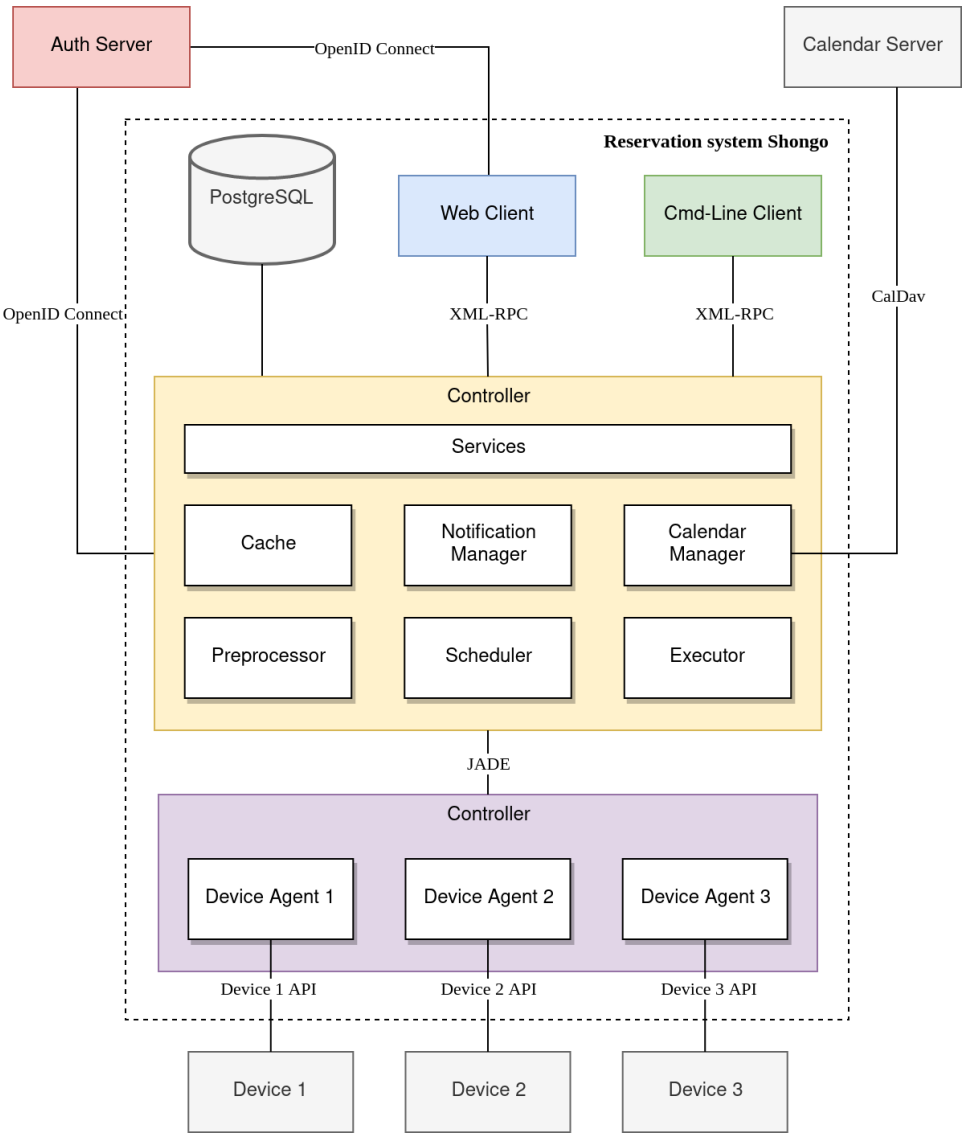

Figure 1.2: Shongo architecture

1.3 What to Change and Why

The main reason for changes is that the architecture concentrates on modularity; however, the system cannot be observed or managed from external sources. The second reason is the primary user interface — the *Web Client* — and its server-side rendering, which is not as powerful as today's JavaScript frameworks like Angular or React. That is why developing a new front-end became a goal of another thesis [7]. Nevertheless, until now, the whole communication within the system has been done via Spring Framework in Java. In order to achieve communication between Shongo and the new Web Client (and possibly other systems or services), which is not part of the Shongo system, a new API has to be implemented.

Comparing the old system architecture shown in Figure 1.2 and the new architecture shown in Figure 1.3 reveals that the *Web Client* component was moved out of the Shongo system. The new *Web Client* is now developed and maintained separately at the new Github repository[6]. Another thesis was devoted to this part of the changes [7]. Additionally, the *Web Client* communicates with the *Controller* via *HTTP* requests for the *REST API* server sub-component instead of previous direct communication via *XML-RPC* services. The implementation of said *REST API* is the subject of this thesis.

6. `https://github.com/shongo/shongo-frontend`

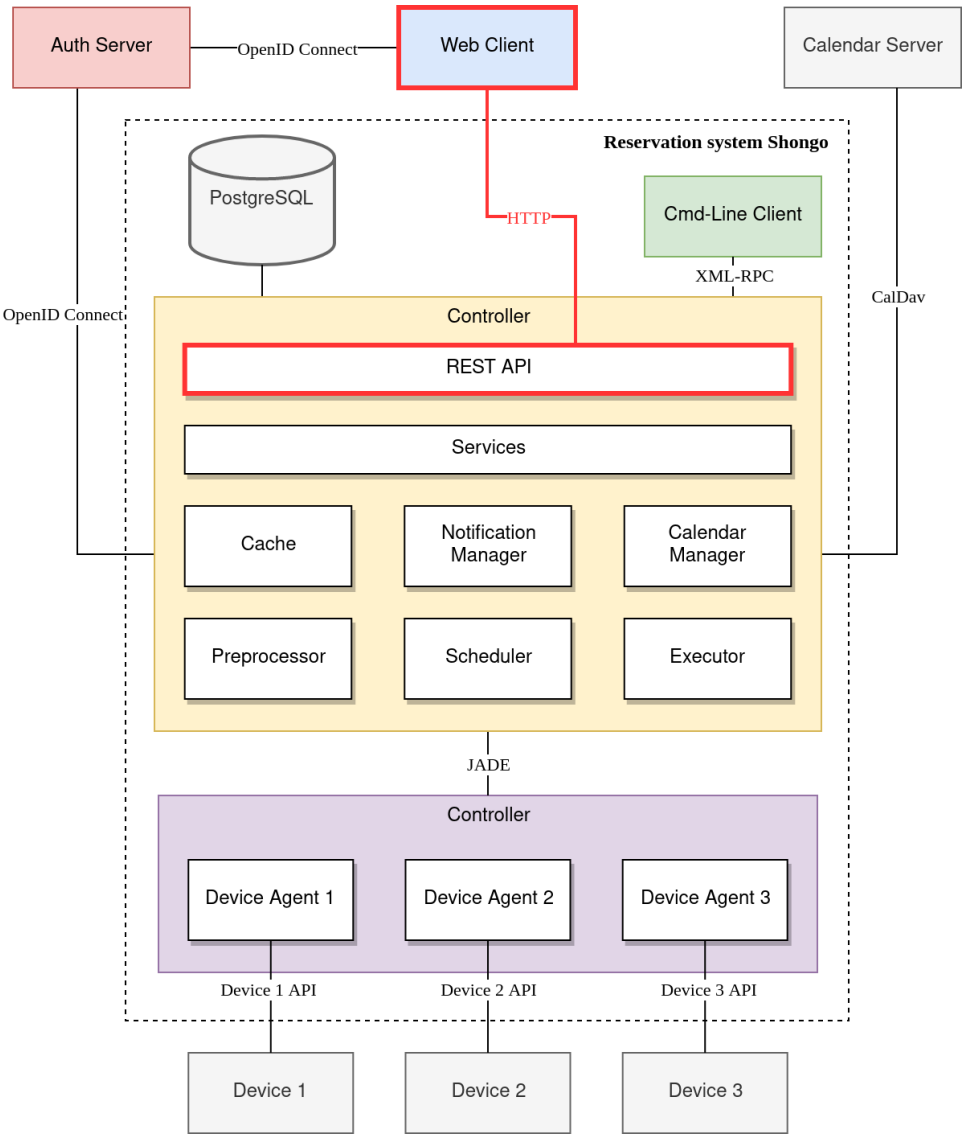

Figure 1.3: New Shongo architecture

2 Used Technologies

This chapter will introduce the technologies used to implement the REST API for the Shongo system.

2.1 Representational State Transfer

Representational State Transfer or `REST` is an architectural style for distributed hypermedia systems. Fielding introduced the REST first time in 2000 [8].

2.1.1 REST design principles

Unlike other API designs such as XML-RPC or SOAP, REST APIs support any format for communication. However, rules have to be followed so the API can be called *RESTful*.

These rules are called REST design principles — also known as architectural constraints [9]:

1. **Uniform interface** – All API requests for the same resource should have the same form disregarding the origin of the request. The same piece of data, such as the name or email, belongs to one uniform URI (uniform resource identifier).

2. **Client-server decoupling** – The client and server sides of the application should be completely independent of each other. The client should have only one piece of information — the URI of the requested resource. This rule implies that the client cannot interact with the server in any different ways. Analogically, the server should not affect the client in any other way than passing requested data to it.

3. **Statelessness** – Each request includes all the information needed. In other words, the server is not allowed to store any data related to the request, and there are no server-side sessions.

4. **Cacheability** – As low as possible, requests should be made to reduce traffic and improve performance, so as many as possible

resources should be cacheable. The server should also inform the client whether the given data in the response is suitable for caching.

5. **Layered system architecture** – Requests and responses can go through many layers on their way. There may be several different intermediaries in the communication. As a result, neither the server nor the client can assume they are communicating directly with the application or with an intermediary.

6. **Code on demand (optional)** – In some cases, the response can be executable code (Java applets, for example). In these cases, the code should only run on-demand.

2.2 Mockoon

Mockoon is an open-source tool for designing and mocking REST APIs. Because of its mocking capabilities, it was chosen to synchronize REST API and front-end development [7]. For this reason, it was also used as an unofficial designing tool for the API.

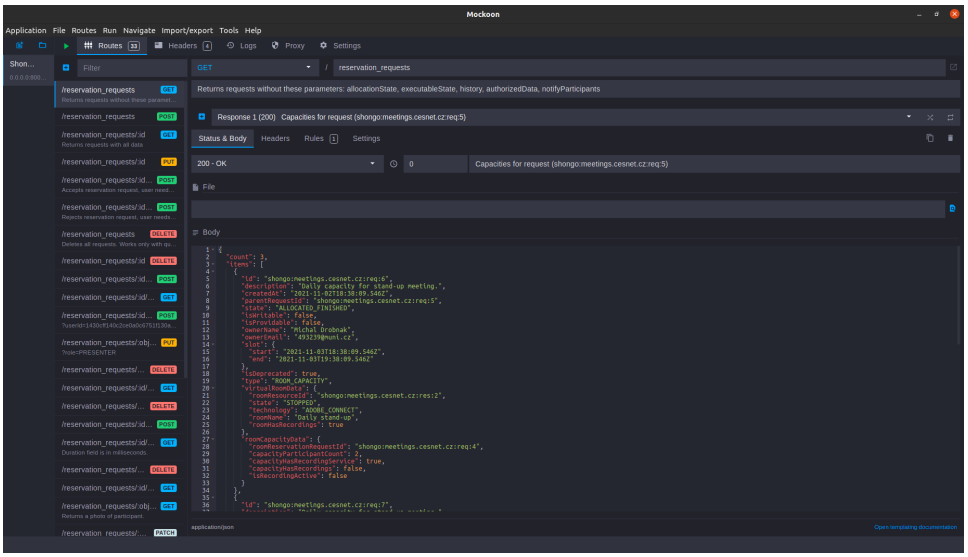

Figure 2.1: Mockoon

2.3 Jackson

Jackson[1], or "the Java JSON library", is an open-source suite of data-processing tools for Java. First of all, it includes `ObjectMapper`, which can serialize Java objects to JSON (JavaScript Object Notation) and deserialize them back. It also supports XML (Extensible Markup Language), YAML (YAML Ain't Markup Language), CSV (Comma-Separated Values), and many other formats. Jackson automatically uses public attributes, or rather getters, to serialize values for JSON and setters to deserialize values obtained from it.

2.4 Spring Framework

The most popular framework for Java applications is Spring, which offers a context for the application. First of all, the framework starts a container (Spring application context) that manages the application components — also known as *Beans* — and wires them together so they can use each other. This wiring is achieved by *dependency injection*. The components do not create and maintain different features they depend upon; instead, they rely on a separate entity (Spring application) to manage the dependent modules and inject them into components that need them [10].

The Shongo system is written in Java, and the Spring framework features were used for faster and easier development of the system. Its dependency injection feature was used to inject functionalities of common objects, such as services and caches, into other objects that need these functionalities. In addition, Spring's web features solved the fundamental issues of the *Web Client* implementation, such as HTTP request-response processing and MVC implementation.

The Spring framework also supports REST API implementation, and this work fully utilizes this feature. Spring uses embedded `ObjectMapper` from Jackson library (Section 2.3) internally to serialize and deserialize foreign formats (JSON in most cases) to Java objects automatically. The `ObjectMapper` is very useful when dealing with HTTP requests which usually use JSON for object formatting. Thanks to this feature, an API developer can easily use simple model objects to

1. `https://github.com/FasterXML/jackson`

acquire or provide data to the client via HTTP communication. This usage will be discussed further in Chapter 3.

2.5 Project Lombok

Lombok is an open-source Java library that generates frequently used Java code using simple annotations like `@Getter`, `@Setter`, `@Data` or `@Builder`. The entirely equivalent Listing 2.1 and Listing 2.2 are presented below to show an example of what is possible with Lombok [11].

Listing 2.1: Vanilla Java

```
public class DataExample {
  private final String name;
  private int age;
  private double score;
  private String[] tags;

  public DataExample(String name) {
    this.name = name;
  }

  public String getName() {
    return this.name;
  }

  void setAge(int age) {
    this.age = age;
  }

  public int getAge() {
    return this.age;
  }

  ...

  @Override public String toString() {...}

  @Override public boolean equals(Object o) {...}
```

```
  @Override public int hashCode() {...}
```

Listing 2.2: Java with Lombok

```
@Data public class DataExample {
  private final String name;
  @Setter(AccessLevel.PACKAGE) private int age;
  private double score;
  private String[] tags;
}
```

2.6 Postman

Postman is an API platform for building and using APIs. It offers an environment and workspaces to test extensive APIs. For these qualities, the Postman was used in the early stages to test the developed REST API.

2.7 OpenAPI Specification

The OpenAPI Specification (OAS) defines a standard interface to RESTful APIs, allowing humans and computers to understand the API's capabilities without access to source code or documentation. When properly defined, a consumer can understand and interact with the remote service with minimal implementation logic. Documentation generation tools can then use an OpenAPI definition to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use-cases [12].

The OAS can define endpoints, a data format and schemas for objects passed between server and client, and security specifications that make the access to the API stricter. The example of OAS is shown in Listing 2.3.

Listing 2.3: OpenAPI.json

```
{
    "openapi": "3.0.1",
```

```
    "info": {
        "title": "Shongo definition",
        "version": "v1"
    },
    ...
    "paths": {
        "/api/v1/reservation_requests": {
            "get": {
                ...
            },
            "post": {
                "parameters": [...]
                "responses": [...],
                "requestBody": {...}
            }
        }
    },
    "securitySchemes": {
        "api_key": {
            "type": "apiKey",
            "name": "api_key",
            "in": "header"
        }
    },
    ...
}
```

2.8 Swagger UI

Swagger UI enables anyone to visualize and interact with the API's resources. It can be automatically generated from OpenAPI Specification, resulting in user-friendly visual documentation [13].

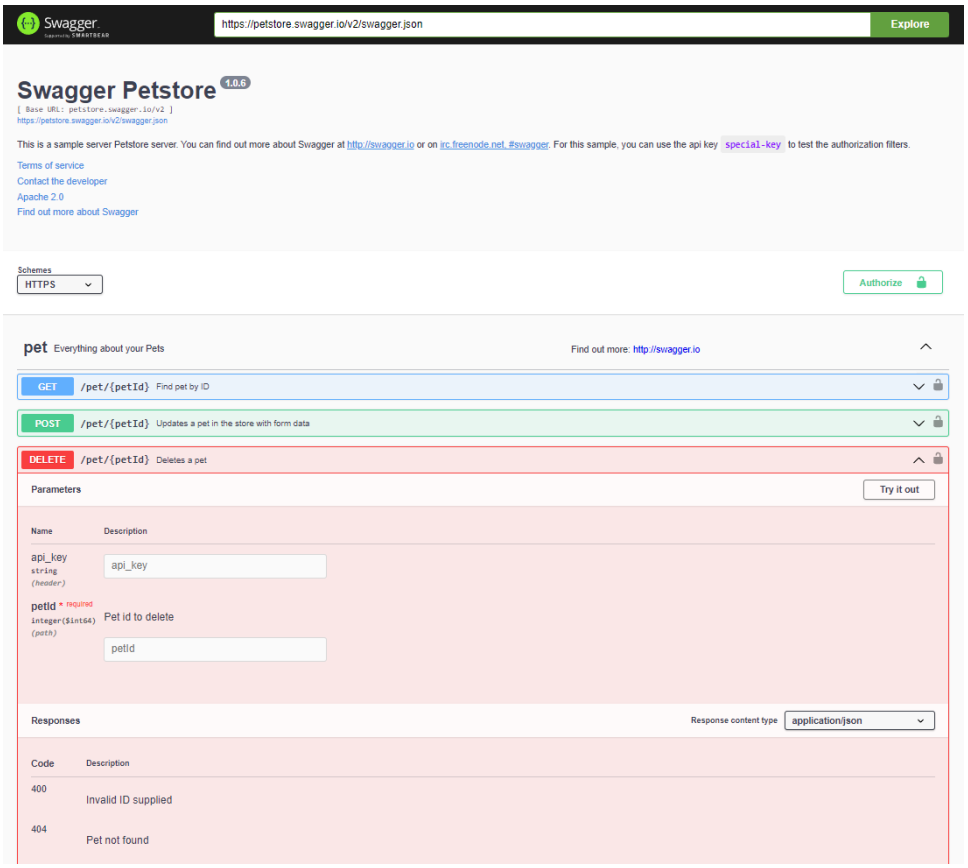

Figure 2.2: Swagger UI

3 Implementation and Documentation

All classes related to the REST API are stored in the Controller's package `cz.cesnet.shongo.controller.rest`. Aside from this package, the only changes made are in a few files that retrieve data from the database — where additional data were required to be loaded — and inter-domain implementation files[2] whose configuration has been merged with the REST configuration.

3.1 REST Server

The cornerstone of the implementation is the REST server which listens for incoming HTTP requests from clients and responds to them in the desired way. The REST server is implemented in the `RESTApiServer` class. At the start of the *Controller* service, the REST server is attached to the *Controller*.

`RESTApiServer` starts a *Jetty*[1] server and configures it according to the controller configuration. The server then listens on the host and port specified in the configuration.

3.2 Configuration

The REST server configuration is required before implementing API endpoints. These configuration files are stored in the `config` subpackage.

3.2.1 Configuration File Extension

First, the configuration of the REST server itself is essential. The controller configuration file `shongo-controller.cfg.xml` has been extended with the parameterization for the new code. Consequently, Shongo administrators can set the REST API server properties inside the `<rest-api>` XML element. An example of this new configuration is shown in Listing 3.1. There are several configurable sub-elements:

- `<host>` – the host where the REST API shall serve

1. `https://www.eclipse.org/jetty/`

- `<port>` – the port where the REST API shall serve
- `<origin>` – the allowed origins for the REST API CORS configuration
- `<ssl-key-store>` – the SSL (Secure Socket Layer) key store from inter-domain extension [2]
- `<ssl-key-store-type>` – the type of key store
- `<ssl-key-store-password>` – the password for key store

Listing 3.1: REST configuration example

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
    ...
    <rest-api>
        <host>meetings.cesnet.cz</host>
        <port>9999</port>
        <origin>http://localhost:4200</origin>
        <origin>https://meetings.cesnet.cz</origin>
        <ssl-key-store>keystore/server.keystore</ssl-key-store>
        <ssl-key-store-password>
            (password)
        </ssl-key-store-password>
    </rest-api>
    ...
</configuration>
```

3.2.2 Security

Next, the security and authentication had to be configured. For this purpose, *Spring security* is used. Spring security dependencies have been added to *Controller*'s `pom.xml`, and security configuration files were stored in `config.security` sub-package.

The web security is configured in the `SecurityConfig` class annotated with the Spring `@EnableWebSecurity` annotation, which allows this class to configure web security[14]. This configuration file defines what should happen to the incoming request. The request filters are

added to the request processing, and CORS (Cross-Origin Resource Sharing)[2] is configured here. The paths that should be skipped in the authentication process are also set here. At the moment, these include OpenAPI, Swagger, inter-domain endpoints[2], and endpoints responsible for processing problem reports.

Additionally, `AuthFilter` is added, which serves as a REST API middleware. It processes each request (omitting those configured in SecurityConfig) using the following steps:

1. Decode an access token wrapped in the HTTP request authorization header as a bearer token.
2. Check the token authorization.
3. Convert the token into a `SecurityToken` object, which contains an *access token* for Shongo Controller and cached information about the user to whom the token belongs.
4. Add this new object to the request attributes using the `TOKEN` key so that subsequent endpoint handling (next middleware or final method) can effortlessly work with it.

If the token is not present or is invalid, the server responds with an HTTP *401 UNAUTHORIZED* response code.

3.3 REST Controllers

This section uses information from Spring documentation [14]. The REST Controllers are responsible for operating the API endpoints and are stored in the `controllers` sub-package. An example of such a Controller can be observed in Listing 3.3.

The REST Controllers are annotated with `@RestController` annotation from the Spring framework, which combines two other annotations:

`@Controller` – makes the class auto-detectable through classpath scanning

2. `https://www.w3.org/TR/cors/`

`@ResponseBody` – binds the return values of the mapped methods to the HTTP response body

This annotation is combined with the `@RequestMapping` annotation, which uses the web request path to map the incoming request to the method that processes the request and responds to the client. `@RequestMapping` can also be parameterized, particularly with `path` (path to the endpoint), `consumes` (expected incoming body form), and `produces` (outgoing body form) parameters.

There are usually only few attributes in the controller classes. These attributes are generally Controller's services, and *dependency injection* is used to access these services. Thanks to Spring annotation `@Autowired`, they are resolved to *Beans* defined in `rest-api-servlet.xml` and injected into the controller class. The example of this file can be noticed in Listing 3.2.

Listing 3.2: rest-api-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=
"http://www.springframework.org/schema/beans"
       ...>
    <context:component-scan
        base-package="cz.cesnet.shongo.controller"/>
    <mvc:annotation-driven />
    <aop:aspectj-autoproxy />

    <bean id="controller" class=
    "cz.cesnet.shongo.controller.Controller"
        factory-method="getInstance"/>
    <bean id="configuration" factory-bean="controller"
        factory-method="getConfiguration"/>
    <bean id="controllerClient" class=
    "cz.cesnet.shongo.controller.ControllerClient">
        <constructor-arg>
            <bean factory-bean="configuration"
                factory-method="getRpcUrl"/>
        </constructor-arg>
    </bean>
```

```
    <bean id="controllerReservationService"
        factory-bean="controllerClient"
        factory-method="getService">
        <constructor-arg value=
        "cz.cesnet.shongo.controller.api.rpc.
        ReservationService" type="java.lang.Class"/>
    </bean>
    ...
    <bean id="cache" class=
    "cz.cesnet.shongo.controller.rest.Cache"/>
</beans>
```

The last part of REST Controllers is the declaration and implementation of methods handling distinct REST resources. The methods are annotated with `@RequestMapping` or rather an exact mapping according to the selected HTTP method (`@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`). These annotations also take a path parameter, which is then concatenated to the class's `@RequestMapping` path parameter. In addition, the methods can retrieve HTTP request attributes annotated with `@RequestAttribute` (most importantly SecurityToken added by `AuthFilter`), HTTP request parameters annotated with `@RequestParam`, path parameters annotated with `@PathVariable` (and specified in the path as `"{variableName}"`) and also the request body annotated with `@RequestBody`.

Listing 3.3: ReservationRequestController.java

```
// Makes this path a Controller (auto-detectable) and binds
    return values of methods to the HTTP request body
@RestController
// Binds this controller to the path parameter
@RequestMapping("/api/v1/reservation_requests")
public class ReservationRequestController
{
    // Controller service used to manage reservations
    private final ReservationService reservationService;

    // @Autowired resolves the service to Bean defined in
        rest-api-servlet.xml
```

```
public ReservationRequestController(@Autowired
    ReservationService reservationService)
{
    this.reservationService = reservationService;
}

// Binds this method to the HTTP GET request with the path
    defined in @RequestMapping
@GetMapping
// Method returns ListResponse<ReservationRequestModel> in
    response body thanks to @RestController annotation
ListResponse<ReservationRequestModel> listRequests(
    // Gets security token from request attributes. The
        token was stored there when AuthFilter processed
        the request.
    @RequestAttribute(TOKEN) SecurityToken securityToken,
    // Reads optional request parameter allocation_state
        and stores it in allocationState variable
    @RequestParam(value = "allocation_state", required =
        false) AllocationState allocationState,
    ...)
{
    // Use Controller service to acquire requested data
    ReservationRequestListRequest request = new
        ReservationRequestListRequest();
    request.setAllocationState(allocationState);
    ...
    ListResponse<ReservationRequestSummary> response =
        reservationService.listReservationRequests(request);
    ...
    // Return the acquired data
    return response;
}

// Sets the response status for successful execution of
    the method
@ResponseStatus(HttpStatus.CREATED)
// Binds this method to the HTTP POST request with the
    path defined in @RequestMapping
@PostMapping
```

```
    void createRequest(
        @RequestAttribute(TOKEN) SecurityToken securityToken,
        // Reads the request body and deserializes it to the
            ReservationRequest object
        @RequestBody ReservationRequest request)
    {
        ...
        // Use Controller service to create reservation
            according to the requested data
        reservationService.createReservationRequest(
            securityToken, request.toApi());
    }

    // Sets possible HTTP responses
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200"),
            @ApiResponse(responseCode = "404", description =
            "Reservation request not found.", content =
            @Content),
    })
    // Binds this method to the HTTP DELETE request with the
        path defined in @RequestMapping concatenated with
        "/{id:.+}". The ':' marks the start of a regex used
        for the id variable (useful to accept only numbers,
        for example). The regex ".+" is used because shongo-id
        can include URL key characters like ':' or '-'.
    @DeleteMapping("/{id:.+}")
    void deleteRequest(
        @RequestAttribute(TOKEN) SecurityToken securityToken,
        // Reads the id variable defined in request path
        @PathVariable String id)
    {
        // Use Controller service to delete reservation with
            id defined in path
        reservationService.deleteReservationRequest(
            securityToken, id);
    }
    ...
}
```

3.4 Data Models

Model classes are mostly POJOs (Plain Old Java Objects) representing objects received or sent by endpoints. The models are stored in the `models` sub-package. Usually, attributes with getters and setters are the only content of these classes. Therefore, most use `@Data` annotation from the project Lombok (Section 2.5). *Jackson* can then serialize and deserialize these classes as described in Section 2.3. Attributes not eligible for serialization are annotated with `@JacksonIgnore`, and for the custom name of a field in the serialized object, `@JacksonProperty ("field_name")` is used. That does not mean that these classes cannot include anything else. Most importantly, they usually also contain `fromApi()` and `toApi()` methods which serve as converters from and to Controller API objects. The Listing 3.4 serves as an example of such a class.

Listing 3.4: RoomModel.java

```
// Generates constructor, getters, setters, equals, and
    hashCode functions
// These generated functions enable Jackson to (de)serialize
    this object
@Data
// Represents Controller's executable
public class RoomModel {

    // Ignores this attribute during (de)serialization
    @JacksonIgnore
    private Cache cache;

    private String id;
    private ExecutableSummary.Type type;
    private TimeInterval slot;
    private TechnologyModel technology;
    private RoomState state;
    // (De)serializes this attribute to license_count field
    @JsonProperty(license_count)
    private int licenceCount;
    ...
```

```
    // Creates this object from data acquired from the
       Controller service
    public static RoomModel fromApi(ExecutableSummary summary)
    {
        RoomModel roomModel = new RoomModel();
        roomModel.setId(summary.getId());
        roomModel.setType(summary.getType());
        roomModel.setSlot(new TimeInterval(summary.getSlot()));
        roomModel.setState(
                RoomState.fromRoomState(
                        summary.getState(),
                           summary.getRoomLicenseCount(),
                        summary.getRoomUsageState())
        );
        ...
        return roomModel;
    }
}
```

3.5 Error Handling

Quite many things can go wrong while processing a request. For this reason, the `error` sub-package was created for files concerning errors and their handling. For extraordinary cases, new exceptions were made, such as `UnsupportedApiException` or `ObjectInaccessible-Exception`. These (and any other) exceptions can be thrown during the request processing, in which case, the server responds with HTTP response 500 Internal Server Error and prints the *stacktrace* to the HTTP body by default. To provide better information for the client, `GlobalControllerExceptionHandler` class was implemented and annotated with Spring web annotation `@RestControllerAdvice`. The methods in the class are annotated with `@ExceptionHandler`, which takes an exception as a parameter. If this exception is thrown, the request processing intercepts and the annotated method runs instead. These methods can handle the exception and respond to the client with the `ResponseEntity` object. As a result, any object can be passed

into the HTTP response body with a desired HTTP response status code.

Listing 3.5: GlobalControllerExceptionHandler.java

```
@RestControllerAdvice
public class GlobalControllerExceptionHandler
{
    @ExceptionHandler(TodoImplementException.class)
    public ResponseEntity<String>
        handleTodo(TodoImplementException e) {
        return ErrorModel.createResponseFromException(e,
            HttpStatus.NOT_IMPLEMENTED);
    }
    ...
}
```

3.6 Miscellaneous

The `ClientWebUrl` class holds the String constants for the endpoints paths. All REST API endpoints paths have the default prefix — `/api/v1`.

`Cache` and `CacheProvider` classes supply a simple *cache* for several frequently retrieved entities represented by `ExpirationMap`, which stores the requested entities for a determined amount of time, so the REST API does not have to wait for a much slower database after each request.

3.7 Documentation

The OpenAPI (Section 2.7) specification of the just described REST API implementation is generated using `Springdoc` from the Spring framework (Section 2.4), so the users can have general and straightforward API documentation available to them. For this purpose, the *springdoc-openapi* dependency was added to *Controller*'s `pom.xml`, and the `OpenApiConfig` class was created. In this configuration file, Springdoc dependencies were imported using Spring's *@Import*, global API attributes were defined with `@OpenAPIDefinition`, and security was defined with `@SecurityScheme` annotation. This is shown in Listing 3.6.

Listing 3.6: OpenApiConfig.java

```
@Configuration
@EnableWebMvc
@OpenAPIDefinition(
        info = @Info(title = "Shongo API", version = "v1"),
        security = @SecurityRequirement(name = "bearerAuth")
)
@SecurityScheme(
        name = "bearerAuth",
        type = SecuritySchemeType.HTTP,
        scheme = "bearer"
)
@ComponentScan(basePackages = {"org.springdoc"})
@Import({
        SpringDocConfiguration.class,
        SpringDocWebMvcConfiguration.class,
        SwaggerConfig.class,
        SwaggerUiConfigProperties.class,
        SwaggerUiOAuthProperties.class,
        JacksonAutoConfiguration.class
})
class OpenApiConfig implements WebMvcConfigurer {
    ...
}
```

Springdoc generates OpenAPI specification based on Spring web annotations by default, but Springdoc annotations such as `@Operation` or `@ApiResponses` can customize the resulting specification. The resulting OpenAPI file is added to the REST server and is available at the `/v3/open-api` path [15].

The Swagger UI (Section 2.8) is then used for user-friendly visualization of the OpenAPI specification mentioned above. The user interface (web site) is acquired and saved in the project resources using the same *springdoc-openapi* dependency as for OpenAPI specification. After that, the default OpenAPI path is set to `/v3/open-api`. Finally, these resources are added to the REST API server and made available at the `/swagger-ui/index.html` path.

4 Final API

This chapter describes the final REST API implemented for the Shongo system. The complete generated (OpenAPI) documentation is available in Appendix A.

Most of the list endpoints return data in `ListResponse<Model>` format. This is done so that the pagination is possible for the resources that have many records. The example is shown in Listing 4.1. The `start` attribute represents the index of the first record in the `items` list. Finally, the `count` attribute informs the receiver about how many records are actually available in total. These endpoints also accept the `start` and `count` query parameters that set the requested *start* and a *maximum number* of items in `ListResponse`. In some cases, `sort` and `sort-desc` query parameters are accepted to set the property by which the entries will be sorted and whether they are supposed to be sorted in ascending or descending order.

Listing 4.1: ListResponse

```
{
    start: 100,
    count: 500,
    items: [
        {
            id: 101,
            name: "My room"
        },
        {
            id: 102,
            name: "Your room"
        }
    ]
}
```

4.1 Users and Groups

The client might need information about users and groups in the Shongo system. These resources are made available via endpoints implemented in `UserController`. The available endpoints include:

- **[GET] /api/v1/users** – Returns all users (`ListResponse<UserInformation>`) that match the `filter` query parameter and are part of the group with the `groupId` query parameter.
- **[GET] /api/v1/users/{userId}** – Returns information about a single user (`UserInformation`). The `userId` path variable defines the requested user.
- **[GET] /api/v1/groups** – Returns all groups (`ListResponse<Group>`) that match the `filter` query parameter.
- **[GET] /api/v1/users/{groupId}** – Returns information about a single group (`Group`). The `groupId` path variable defines the requested group.
- **/api/v1/settings** – Manages the user's settings. `SecurityToken` acquired from the `Authorization` HTTP header defines the user whose settings are being addressed. There are two available HTTP methods:

  GET – Returns the user's predefined settings (`SettingsModel`).

  PUT – Updates the user's predefined settings to the settings (`SettingsModel`) received in the request body.

4.2 Resources

The client might also need information concerning `Resources`. He might need information about what resources are available and how much these resources are used. To achieve that, the client can use the endpoints implemented in `ResourcesController`. The available endpoints include:

- **[GET] /api/v1/resources** – Returns all available resources (`List<ResourceModel>`) that users can reserve. The `technology` (`TechnologyModel`) and `tag` query parameters can filter the requested resources.
- **[GET] /api/v1/resources/capacity_utilization** – Returns utilization of all resources (`ListResponse<ResourceUtilizationModel>`) in interval specified by `interval_from` and `interval_to` query parameters. Also, a requested period can be specified by the `unit` (`Unit`) query parameter.
- **[GET] /api/v1/resources/{id}/capacity_utilization** – Returns detailed utilization information (`ResourceUtilizationDetailModel`) about a resource specified by `id` path parameter. The detail contains additional information about reservations that used the resource in the specified interval, thus utilizing the resource.

4.3 Reservation Requests

The following endpoints are the most important since they manage the central part of the Shongo system — reservation requests. The available endpoints include:

- **/api/v1/reservation_requests** – Manages the reservation requests. There are two available HTTP methods:
  - **[GET]** – Returns all reservation requests (`ListResponse<ReservationRequestModel>`). However, they can be filtered by many query parameters closely described in Table 4.1.
  - **[POST]** – Creates a new reservation request defined by `ReservationRequestCreateModel`, which is acquired from the request body.
- **/api/v1/reservation_requests/{id}** – Manages the reservation request specified by `id` path parameter. There are two available HTTP methods:
  - **[GET]** – Returns detailed information about the reservation request (`ReservationRequestDetailModel`). It contains additional information about the current state of the reservation

request, its authorized data (`RoomAuthorizedData`) holding access pins and aliases, and the history of the reservation request (`List<ReservationRequestHistoryModel>`) holding information about how the request changed over time.

[PATCH] – Modifies the reservation request with parameters specified by `ReservationRequestCreateModel`, which is acquired from the request body.

[DELETE] – Deletes the reservation request.

- **[POST] /api/v1/reservation_requests/{id}/accept** – The administrator or resource owner can use this endpoint to accept the reservation request awaiting confirmation, so it becomes ready for allocation and eventually turns into a reservation.

- **[POST] /api/v1/reservation_requests/{id}/reject** – The administrator or resource owner can use this endpoint to reject the reservation request awaiting confirmation.

- **[POST] /api/v1/reservation_requests/{id}/reject** – Reverts the modifications of the reservation request.

Table 4.1: List reservation requests parameters table.

Name	Description
resource	Filters entries that use the given *resource*
type	Filters entries with the given *type*
search	Filters entries that contain the *search*
participant_user_id	Filters entries that have a participant with the id *participant_user_id*
user_id	Filters entries that owner's id equals *user_id*
interval_from	Filters entries from this date
interval_to	Filters entries to this date
technology	Filters entries by the given *TechnologyModel*
parentRequestId	Filters entries that have *parentRequestId*
allocation_state	Filters entries that have *allocation_state*

4.4 User Roles

When a user decides to adjust the `UserRoles` of the reservation requests he has access to, he can use the endpoints implemented in `UserRoleController`. The available endpoints include:

- **/api/v1/reservation_requests/{id}/roles** – This endpoint manages user roles that are configured for the reservation request specified by the `id` path parameter. There are two available HTTP methods:

  [GET] – Returns all roles (`ListResponse<UserRoleModel>`) that are configured for specified reservation request.

  [POST] – Creates a new role defined by `UserRoleModel` acquired from the request body.

- **[DELETE] /api/v1/reservation_requests/{id}/roles/{entityId}** – Deletes the configured user role specified by `entityId` path parameter from reservation request specified by `id` path parameter.

4.5 Participants

If a client wants to adjust the users or groups that can participate in reserved meetings, he can use the endpoints defined in `ParticipantController`. The available endpoints include:

- **/api/v1/reservation_requests/{id}/participants** – Manages the participants configured for reservation request specified by `id` path parameter. There are two possible HTTP methods:

  [GET] – Returns all participants (`ListResponse<ParticipantModel>`) that are configured for the reservation request specified by the `id` parameter.

  [POST] – Creates a new participant defined by `ParticipantModel` acquired from the request body for the specified reservation request.

- **/api/v1/reservation_requests/{id}/participants/{participantId}** – Manages the participant specified by `participantId` path parameter

that is already configured for reservation request specified by `id` path parameter. There are two available HTTP methods:

[PUT] – Updates the existing participant's role with the participant role defined by the `role` (`ParticipantRole`) query parameter.

[DELETE] – Deletes the participant from the reservation request configuration.

4.6 Rooms

The endpoints discussed in this section concern the reservation recordings and are defined in `RoomController`. The available endpoints include:

- **[GET] /api/v1/rooms** – Returns all rooms (`ListResponse<RoomModel>`) that can be filtered by the participation of a participant specified by the `participant-user-id` query parameter.

- **[GET] /api/v1/rooms/{id}** – Returns a detailed information about room (`RoomAuthorizedData`) that contains additional information about pins and aliases.

4.7 Recordings

As described in Chapter 1, the virtual meetings can be recorded. The available recordings can be managed using the endpoints implemented in `RecordingController`. The available endpoints include:

- **[GET] /api/v1/reservation_requests/{id}/recordings** – Returns all recordings (`ListResponse<RecordingModel>`) of reservation request specified by `id` path parameter.

- **[DELETE] /api/v1/reservation_requests/{id}/recordings/{recordingId}** – Deletes the recording specified by `recordingId` path parameter from reservation request specified by `id` parameter.

4.8 Report

The users might run into a problem. That is where the `ReportController` comes in to inform the administrators about the problem. At the moment of writing this thesis, there is only one endpoint:

- **[POST] /api/v1/report** – Sends the `ReportModel` acquired from the request body to the system administrators configured in Controller's configuration.

5 Conclusion and Future Work

This work aimed to design and implement REST API, which exposes the reservation system Shongo's back-end (Controller module) to higher layers.

In conclusion, this work implemented a new functional API fulfilling REST principles for the Shongo system. As a result, the new front-end [7] can interact with the Shongo back-end, thus making the original `shongo-web-client` module obsolete. However, this functionality was tested just on the Shongo *development* server[1] since the production deployment was out of the scope of this thesis. Moreover, the API makes the Shongo system available for any application or service outside of the Shongo system.

The resulting REST API is documented with `Javadoc`, and the OpenAPI specification (Section 2.7) is generated and made available on the REST server. In addition, this specification is added to Spring UI (Section 2.8), which is also deployed on the REST server.

Several steps are yet to be taken before the end users can fully use the application. First and foremost, the functionality of the REST server with the new front-end has to be adequately tested on the development server, and the REST server must also be eventually deployed to production on the *meetings* server[2]. Additionally, the implementation of run-time management endpoints should be fine-tuned after deploying the new front-end. Finally, the next step could be more complex associations between Shongo exceptions and REST API `ExceptionHandler`.

1. `https://shongo-dev.cesnet.cz`
2. `https://meetings.cesnet.cz/`

A Attached Files

This work consists of the electronic version of the thesis and a zip archive `shongo.zip`, both publicly accessible in the Information System of Masaryk University[1]. The archive consists of 2 parts:

- The source code of the Shongo system with implemented REST API is available in the `shongo` directory.
- Generated OpenAPI specification (Section 2.7) for implemented REST API is available in the `openapi.json` file. The specification can be tested in Swagger UI (available at `https://editor.swagger.io/`, for example).

1. `https://is.muni.cz/th/jhvt4/`

Bibliography

1. ŠROM, Martin; HOLUB, Petr; RŮŽIČKA, Jan; LIŠKA, Miloš; PAVELKA, Ondřej; NOVAKOV, Ivan. *Shongo API* [online]. Brno: CESNET, z. s. p. o., 2013 [visited on 2022-03-09]. Available from: `https://github.com/shongo/shongo/blob/master/shongo-doc/shongo/api/api.pdf`.
2. PAVELKA, Bc. Ondřej. *Interdoménové rozšíření rezervačního systému Shongo*. 2016. MA thesis. Masaryk University.
3. PERICHTA, Bc. Marek. *Redesign a reimplementácia rezervačného systému Shongo*. 2020. MA thesis. Masaryk University.
4. *Shongo* [online]. Brno: CESNET, z. s. p. o., 2018 [visited on 2022-03-09]. Available from: `https://shongo.cesnet.cz/`.
5. *CESNET* [online]. Brno: CESNET, z. s. p. o., 2022 [visited on 2022-03-09]. Available from: `https://www.cesnet.cz/sdruzeni/`.
6. Czech academic identity federation [online]. 2017 [visited on 2022-03-09]. Available from: `https://www.eduid.cz/`.
7. DROBNAK, Michal. *Responzívny frontend pre službu CESNET Meetings*. 2022. Bachelor's thesis. Masaryk University.
8. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures* [online]. Irvine, 2000 [visited on 2022-03-09]. Available from: `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`. PhD thesis. University of California, Irvine.
9. REST APIs [online]. 2021 [visited on 2022-03-09]. Available from: `https://www.ibm.com/cloud/learn/rest-apis/`.
10. WALLS, Craig. *Spring in action*. Simon and Schuster, 2022.
11. *Project Lombok*. Project Lombok, 2022. Available also from: `https://projectlombok.org/`.
12. *OpenAPI Specification*. SmartBear Software, 2022. Available also from: `https://swagger.io/specification/`.
13. *Swagger*. SmartBear Software, 2021. Available also from: `https://swagger.io/tools/swagger-ui/`.

14. *Spring Framework 5.3.19 API*. VMware. Inc., 2022. Available also from: `https://docs.spring.io/spring-framework/docs/current/javadoc-api/`.

15. NASSLAHSEN, Badr. *Springdoc-openapi documentation*. 2021. Available also from: `https://springdoc.org/`.