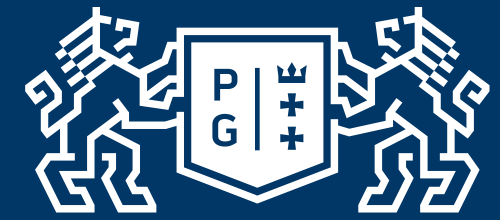

Spring MVC - REST Services

Internet Services Architectures

Michał Wójcik, Waldemar Korłub

Approach to web applications

Before the era of advanced personal devices (e.g. smartphone, smartwatch, etc.), typical server application would generate HTML documents which where rendered by internet browsers.

Today, the browser client is only one of the many channels for accessing services on the server:

- mobile applications - devices such as smartphones, tablets, smartwatches,
- software in devices such as Smart TV, Smart Car,
- intelligent assistants: Alexa, Siri.

New client applications require a programming API instead of HTML documents.

Approach to web applications

Since the server must provide the API implementing business logic anyway, the client in the browser can as well use it. Exposing the same server API to all client application types (mobile applications, websites, etc.) means much easier application maintenance.

Web browser client applications (web front-end) use JavaScript language based frameworks:

- Angular(Google),
- React(Facebook),
- Vue.js,
- Backbone.js
- ...

Web services

Web services - client and server applications communicating with each other via the web using the HTTP protocol:

- interoperability between applications running on different platforms and frameworks,
- descriptions processed by applications,
- use of XML or JSON,
- loosely coupled services can be combined to create complex operations.

RESTful Web Services

Basic assumptions:

- stateless, interaction should be immune to server restart,
- application server caching services and other elements can be used to improve performance, as long as the elements returned by the service are not dynamically generated and can be cached,
- possible description with WADL or Swagger,
- the producer and the consumer must handle the same context and the content sent,
- low data overhead, ideal for devices with limited resources,
- often used in conjunction with AJAX technology.

Resources

Resources should be arranged in a hierarchy (nouns instead of verbs):

- `api/books` - all books,
- `api/books/1` - book with index 1,
- `api/books/1/authors` – authors of the specified bookd,,
- `api/books/1/authors/1` – author with index 1 of the specified book.

Instead of indexes, other values that uniquely describe the item can be used:

- next index,
- domain key (e.g. isbn for books),
- generated UUID.

Resources

Parameter types:

- path param - parameter that is an element of the address, indicates resources,
- query param - the parameter appended to the address (after the `?` character), make the user's (client application) query more precise:
  - paging: `api/books?offset=10&limit=5` - download five books starting from 10.
  - filtering: `api/books?unread=true` - download only unread books,
  - sort: `api/books?sort=latest` - get books sorted by release date.

Resources operations

Resources management is done with HTTP methods:

- `PUT` - update resource,
- `GET` - retrieve resource,
- `POST` - create new resource,
- `DELETE` - delete resource.

operation	resource	effect
GET	`api/books`	fetch all books
GET	`api/books/1`	fetch single specified book
POST	`/api/books`	add new book
POST	`/api/books/1`	not used
PUT	`/api/books`	replace collection of books with new collection, rather not used
PUT	`/api/books/1`	update specified book
DELETE	`/api/books`	delete whole collection
DELETE	`/api/books/1`	delete specified book

REST as CRUD

operation	REST	SQL
create	POST	INSERT
read	GET	SELECT
update	PUT	UPDATE
delete	DELETE	DELETE

Request result

Request result should contain HTTP response code.

request	resource	example response code
GET	`api/books`	200 OK
GET	`api/books/1`	200 OK, 404 Not found
POST	`/api/books`	201 Created (+location), 403 Forbidden
POST	`/api/books/1`	405 Method not allowed
PUT	`/api/books`	405 Method not allowed
PUT	`/api/books/1`	200 OK, 401 Unauthorized
DELETE	`/api/books`	200 OK, 403 Forbidden
DELETE	`/api/books/1`	200 OK, 403 Forbidden, 404 Not found

Tools

Popular tools for testing REST API:

- Postman:
  - chrome plugin (downloaded from Chrome Web Store),
  - standalone application (downloaded from homepage);
- various different REST clients as FireFox or Chrome plugins,
- SoapUI - standalone,
- HTTP Client plugin delivered with Intelij IDEA.

Rest service

```java
@RestController
@RequestMapping("/shop")
public class ShopController {

    private OrderService service;

    public ShopController(OrdersService service) {
        //...
    }

    @GetMapping("/orders")
    public List<Order> listOrders(@RequestParam(required = false) String sort) {
        //...
    }

    @GetMapping("/orders/{id}")
    public ResponseEntity<Order> getOrder(@PathVariable UUID id) {
        //...
    }

    @PostMapping("/orders")
    public ResponseEntity<Void> addOrder(@Request BodyOrder order) {
        //...
    }

}
```

REST service

The above configuration allows to send the following HTTP requests (assuming the server was started on the default port 8080):

- GET na `localhost:8080/shop/orders` - returns all orders,
- GET na `localhost:8080/shop/orders?sort=asc` - returns all orders sorted ascending,
- GET na `localhost:8080/shop/orders/9eaee866-6eb3-11ea-bc55-0242ac130003` - returns order with specified id,
- POST na `localhost:8080/shop/orders` - adds new order.

REST service - annotations

Used annotations:

- `@RestController` - registering a class as a controller for REST services, an instance of the class will be created automatically,
- `@RequestMapping` - register based address for all class methods,
- `@GetMapping` - handling GET HTTP requests,
- `@PostMapping` - handling POST HTTP requests,
- `@PathVariable` - mapping path parameter to method argument,
- `@RequestParam` - mapping query parameter to method argument,
- `@RequestBody` - mapping request body (default JSON) to an object.

Binding parameters and responses

```java
@GetMapping("/orders")
public List<Order> listOrders(@RequestParam(required = false) String sort) {
    List<Order> orders;
    if (sort != null) {//optional query param
        orders = service.findAllSorted(sort);
    } else {
        orders = service.findAll();
    }
    return orders; //HTTP 200 code with body automatically converted to JSON array
}
```

```java
@GetMapping("/orders/{id}")
public ResponseEntity<Order> getOrder(@PathVariableUUID id) {
    Order order = service.find(id);
    if (order == null) {
        return ResponseEntity.notFound().build();//HTTP 404 error code
    else {
        return ResponseEntity.ok(order);//HTTP 200 code with body in JSON format
    }
}
```

Response body:

- automatically built (by default to JSON format) based on the object returned by the method,
- using the `ResponseEntity` class that allows to control all aspects of the HTTP response (code, body, headers).