THE SPRING BOOT REST API APPLICATION “introtelecom”

1. INTRODUCTION

The Spring Boot REST API application “**introtelecom**” provides an introduction to the mobile phone service management. Mobile phone services are an area of modern technology that requires the most complex software solutions nowadays. This is a demo application and it simplifies many details related to mobile services, to show and explain the mobile service basics, how to:
- register the phones, grant the monthly package frame to each phone
- register customers (owners of physical phones) and users (owners of software accounts).
- provide add-ons (additional services during the month) to customers
- create a record of mobile service (Service Detail Record -SDR, its type, duration and amount)
- provide the user with details of the current information (what amount of service is left until the end of the month)
- generate monthly bill facts for the previous month

Nowadays there are many types of mobile services. **Basic types of mobile services** used in this application:
- calls (**cls**),
- short message service (**sms**),
- internet (**int**),
- applications and social media (**asm**),
- international calls (**icl**)
- roaming (**rmg**)

Also, there are two **payment models** here:
- prepaid - you pay first and then use the services
- postpaid - payment is made after the delivered service, in the end of each month.

2. LIST OF TECHNOLOGIES

- **Spring Boot** – packaging and deployment for REST (REpresentational State Transfer) API
- **Spring Data JPA** – ORM and transaction-driven data handling
- **Hibernate ORM** – used together with Spring Data JPA
- **Spring Security framework** – for logging and access activities
- **Spring Boot Validation** (with **Hibernate Validator**) – for input and database data validation
- **PostgreSQL** – RDBMS data layer
- **Swagger** – UI for REST API and documentation
- **JWT** – JSON Web Token for authentication
- **MapStruct** – to map between different object models (e.g. entities and DTOs)
- **Lombok** – java library that eliminates boilerplate code
- **Slf4j-with-Logback** library for logging

A standard Java servlet container (**Tomcat**) is used as the server platform, as part of the Spring Boot package.
Maven is used as the build system.
IntelliJ IDEA IDE
The code is **Java 8** – compliant.

More about Spring Boot architecture [1],[2],[3].

3. APPLICATION FUNCTIONALITY AND USER ROLES

A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services.
When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, can be delivered in one of several formats via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text. [4]
JSON is used here.

After running the project, open a browser and enter the following URL:

```
http://localhost:8080/swagger-ui/index.html
```

The entrance to the application is the Access Controller; the login service grants a JWT token to each **valid User** - with a valid username and password; with JWT token they can thhen access other resources in the application.

This application receives user requests through the Swagger UI. Requests are then sent to several REST controllers that forward data to services for further processing.

A good and well documented example of Spring Boot REST API with Swagger UI is given on the net [5],[6].

The application interacts with users - **User**. A User has two distinct roles:
1. ***ADMIN***, whose role is to control the entire application,
2. ***CUSTOMER***, who can get information related to his phone number and mobile services in general.

ADMIN role User can send requests to almost all REST controllers except the **Client** Controller; in addition to Access Controller, **CUSTOMER role User** can only send requests to the **Client** Controller.

Admin (administrator) and **Customer** are physical persons defined by some unique data - personal number and e-mail; they can access the application via the **User** roles.

Phone is represented by unique phone number. The phone number is the most important data which is the primary key or foreign key for almost every db table. Another important piece of information is the **package code** that determines phone's **package plan**. If we know the phone number, we can determine what package plan the phone uses.

Package plan defines which types of mobile services are included in the **package frame**, defines a specific package code, monthly quota and total monthly price of package. The picture **`docs/package_plans.png`** represents the plans used in this application.

Package frame represents a monthly quota of various services that are defined through a particular phone and package plan.

At the beginning of each month (the first day of the current month), package frames are granted to the phones. It is executed automatically.

Administrator can **only have one phone**, the same phone assigned to the **ADMIN role User.** This phone is for admin support.

Customer can have **more than one phone,** but each of these phones is assigned to only one **User (CUSTOMER role User)**.

If a customer does not have enough resources of a specific service by the end of the current month, he can demand an add-on service through the addon frame.

Addon (add-on service) defines types of add-on services through add-on codes. So, we have extra: add-on calls, add-on sms, add-on internet, add-on applications and social media, add-on international calls and add-on roaming. It also defines the add-on quota and the total add-on price for a specific service on a monthly basis.

Addon types are defined in the **addon** db_table.

Addon frame with addon code, represents the add-on monthly quota of a specific service.

SDR (**Service Detail Record**) represents recorded data for a specific service used by the phone during the month. This name is derived from an earlier term - **CDR** (**Call Detail Record**). But, nowadays in addition to the call service, we also have a bunch of new services: sms service, internet, various social media and application services, international calls and roaming.

SDR types are defined in the **service** db_table.

The **`docs/add-ons_and_SDRs.png`** picture represents the add-on quotas and SDR codes used in this application.

SDR input format has:
- phone number
- service code
- called number ("-" for internet, applications and social media)
- service start time [s]
- service end time [s]
- msg amount ("1" - sms, "0" - other services)
- mb amount [MB] (represents consumption for internet or applications and social media; "0.00" - for calls, sms, international calls and roaming)

an important parameter, derived from the input time data, is:
duration [min] = end time - start time
(for sms service the duration is irrelevant, so we use end time - start time =1s).

we can calculate how much of a particular service is left until the end of the month:
left(remaining) amount = total input amount - current output amount

for a specific service:
`total input amount` = `amount of service within the package frame` + `d*(amount of add-on service)`

where the parameter d is:
d=0 without add-on service
d=1 add-on service on demand(request)

for a specific service:
current output amount = sum of all amounts, calculated from the SDR data related to this specific service and specific phone, from the beginning of the current month until the current moment).

Sometimes a certain service is used up before the end of the month. When the customer wants to use this service again, the system signalizes EOS (End of Service) - which means this service is not available until the end of the current month.

In this application the EOS can be detected during service recording - that is, before saving the SDR. In the commercial applications service detail records are managed and stored by the system; here, in this demo, an admin user manages these records. Whenever we want to save a SDR, the EOS check is called. When a resource is used up and a service is abruptly terminated, this is reflected in the SDR input data - the software modifies them before inserting them into the database:

- for calls, the service is interrupted when total duration is reached; the end time is then adjusted to an appropriate value that fits into the total input duration.

- for sms, the service is interrupted when a customer wants to send another message above the total number of sms messages; the message will not be sent.

- for internet, the service is interrupted when the total input amount in MB for internet is reached; the input data for the MB amount is corrected to the appropriate value that fits into the total input amount; the second correction is for end time - this is reduced to half the duration of the input time; it is taken only for demo purposes, because internet consumption - MB/s is not constant and we cannot calculate the exact time completion.

- for social media and applications - similarly to internet service.

- for international calls, the service is interrupted when the total input amount (in currency unit, cu) is reached; the total output amount depends not only on the call duration, but also on the zones.

- for roaming calls - similarly to international calls.

In all these cases EOS note is set in the SDR record.

Nowadays, some of the services are unlimited for advanced users. In this demo application, the packages "13" and "14" have unlimited access to calls and sms services, and package "14" has unlimited access to the internet. This means that EOS is not applied in these cases.

NOTE: there is a difference in calculation for total output amount between calls on the one hand and international calls and roaming on the other.
Total output amount for calls is given in minutes - it depends only on the duration.
Total output amount for international calls and roaming is given in currency unit - which means that it depends not only on the duration, but also on the zone; for different zones there is a different price per minute (i.e. 4 cu/min for zone1, 8 cu/min for zone2)

Also, at the beginning of each month the application automatically generates **monthly bills** for the previous month. These data are recorded in the db table monthly_bill_facts.

```
total monthly price =  package price + d* add-on service
```

the value of d parameter is mentioned in the previous text above.

Monthly bills are generated for postpaid customers only.

Admin role User and **Customer** role User

Admin role and **Customer** role User have distinct views and responsibilities:

ADMIN role User:
- creates and manages admin, customer, user and phone data
- associates the package plan to specific phone
- gives the phone (phone number) to the customer
- associates specific phone to the user
- grants add-on frame to the CUSTOMER role user on demand
- creates an SDR (Service Detailed Record) for each service performed by the phone during the month
- can monitor the current information about service consumption for a specific phone during the month
- can monitor the monthly bills of a specific phone
- takes care of connection order:

for ADMIN role User
1. creates a Phone object
2. then creates an Admin object which contains that Phone object (phone number)
3. finally, creates (ADMIN role) User object and attaches it to that Phone

for CUSTOMER role User
1. creates a Customer object
2. then creates a Phone object
3. adds this Phone object to the Customer (through the JOIN table **customer_phone**)
4. finally, creates (CUSTOMER role) User object and attaches it to that Phone

The Customer can have more than one phone. In that case the above procedure is the same, excluding the 1st step (no need for creation, because the Customer object already exists)

***CUSTOMER* role User (for a specific phone)**:
- can be informed about the current service consumption - current info
- can be informed about the details of the monthly bills from previous months
- can change user password
- can be informed about the package plans and add-ons

__Application handles:__

- **Login** (login control with the support of Spring Security framework and BCrypt password encoding feature with the JWT token granted to the valid User)
- **User URL access** (access control with the support of Spring Security framework)
- **Exception Handling** (when an exception is thrown - due to invalid data)
- **Automatic generation** (of package frames and monthly bills)
- **Object mapping** (between different object models - e.g. entities and DTOs)
- **Input data form validation** and **custom validation** (support of Spring Boot Validation)
- **Repository search**
- **Generation of response as a JSON output data**

The REST API APPLICATION “**introtelecom**” can be categorized as a **__real-time dependent__** application. It means that package frames, add-on frames, SRD records and monthly bills are dependent on the current month. Package frames and monthly bills are generated automatically at the beginning of the month. Because of that, services for CurrentInfo and MonthlyBillFacts are not available from midnight to 12 o'clock on the 1st day of the month.
Add-on frames and SDR records are generated, with the help of ADMIN role User, during the current month.

As mentioned above, this demo app simplifies many things to show and explain the basics of mobile services. So, __for the sake of simplicity__:

-there are only six package plans (packages); in reality, every mobile provider offers a huge number of packages

- there are two zones for international calls and roaming; in reality there are at least three zones

- for roaming service there is no price for incoming calls

- roaming and international calls are not included in prepaid packages

- the customer can demand add-on service during the current month, and it is valid until the end of the month

- there are only six add-on services that correspond to basic types of mobile services used in this application

- customers can only demand add-on service types that are included in the monthly package frame

- customers cannot demand another add-on service of the same type, during the current month

- the packages and add-ons have the same prices for international calls and roaming

- for both types of payments (prepaid and postpaid), package frames start at the very beginning of the month; package frames and add-on frames expire at the very end of the month; so, no leftovers are carried over to the next month

- package frames start automatically at the very beginning of the month; currently, package frames cannot be activate later in the month

- the app does not handle how customers demand the specific add-on service; for demo purposes, ADMIN role user grants add-on frames to CUSTOMER role users in an arbitrary way

- the app does not handle how customers pay for services

- additional services for the mobile phone (call identification, call waiting, call forwarding, phone number display ban, MMS (Multimedia Messaging Service), voicemail...) are not included

- additional internet services (hosting, wi-fi zone, web mail...) are not included

- service recording (SDR) is handled by the ADMIN role user via input data
- and other things ...

NOTE: The names, codes, amounts and prices for package plans, add-on services, SDRs are arbitrary, created only for this application, and this data is not related to any real-world mobile operator.

NOTE: This demo app uses phone numbers, personal numbers, passwords, person's names, addresses, e-mails. Any similarities to real personal data, events, and terms are coincidental.

4. DESIGN CONSIDERATIONS

This is a classic REST API application with a Swagger client. It consists of:

- PostgreSQL database named "**intro_telecom**".
- Spring Boot application "**introtelecom**", designed as a Maven Project
- Swagger UI; receives user requests and forwards them to the Spring Boot application for further processing; also displays the REST response in JSON format.

4.1. DATABASE DESIGN

The (PostgreSQL) database schema "**`intro_telecom`**" is given in **`docs/intro_telecom-schema.png`**.

The script file that creates database "**`create_db.sh`**" is given in the code.

For database and tables creation, "**`intro_telecom_01_database.sql`**" and "**`intro_telecom_02_tables.sql`**" are used, respectively.

Database model has 13 data tables:
`phone` – main db table with phone number, package code and other data
`user_data` – to store data of each user who interacts with the application
`role` – can be ADMIN or CUSTOMER
`admin` – personal data of the administrator
`customer` – personal data of the customer
`customer_phone` – to assign a phone to the customer
`package_plan` – stores types of monthly packages
`package_frame` – monthly quota of a specific package type
`addon` – stores types of add-ons
`addon_frame` – monthly quota of a specific add-on type requested by the customer
`service` – stores types of mobile services
`sdr` – for storing data on each individual service record that customer performed via the phone number
`monthlybill_facts` – to store data about each individual monthly bill

Database tables are given in **`docs/intro_telecom-db_tables.png`**

Under the **`src/main/resources`** package there is the **`application.properties`** file which provides property values for the database connection and other parameters.

The basic data need to set roles, package plans, add-on service codes, codes for service detail records, data for the first ADMIN role User and its phone, are defined in "**`intro_telecom_03_tables_initial_population.sql`**".

In this app the **`QuickPasswordEncodingGenerator.java`** utility class is used to create the first encrypted ADMIN role User password.

ADMIN user gets access with username and password. The app generates JWS token. All other data can be populated through the Swagger UI, after the application is launched.

<u>NOTE:</u> Regarding the foreign keys, there is currently no action for ON DELETE.

4.2. PROJECT CREATION AND PACKAGES

This application was created and developed as a Maven Project in the IntelliJ IDEA IDE.

The application has the main package **`com.snezana.introtelecom`** and several subpackages (controller, dto, entity, enums, exception, mapper, repository, response, security, service, swagger, util, validation), the file **`pom.xml`** and Spring Boot application class **`IntrotelecomApplication.java`**.

Also, this app has several types of tests, which are described later in the text.

5. DESIGN PARTS

5.1. ENTITY

Entity classes correspond to tables in the database:

class	db tables
`AddOn.java`	**`addon`**
`AddonFrame.java`	**`addon_frame`**
`Admin.java`	**`admin`**
`Customer.java`	**`customer`**
`MonthlyBillFacts.java`	**`monthlybill_facts`**
`PackageFrame.java`	**`package_frame`**
`PackagePlan.java`	**`package_plan`**
`Phone.java`	**`phone`**
`PhoneService.java`	**`service`**
`Role.java`	**`role`**
`ServiceDetailRecord.java`	**`sdr`**
`User.java`	**`user_data`**

NOTE: **`customer_phone`** table is a join table with no relevant entity class. It depicts the relationship between customers and phones, i.e. it shows which phone belongs to a particular customer.

5.2. REPOSITORY

Each entity has its own corresponding repository interface that handles CRUD activities.

All repositories use the *Automatic Custom Query* methods (**`findBy…`**). According to these methods Spring Data automatically generates queries from the method names.

Additionally, there are *Manual Custom Query* methods (via the **@Query** annotation, using JPQL with name params), where we have manually defined queries to fetch data.

5.3. MAPPER AND DTO (Data Transfer Objects)

Mapper uses **MapStruct**, a code generator which simplifies the implementation of mapping between different object models (e.g. entities and DTOs).

Mapping is useful when transferring entities between project layers. We use DTOs instead of entities. DTOs encapsulate the data which we send to the controller. Important data that we do not want to send, are not carried in the DTOs (for example id or password). More about mapping and MapStruct [7], [8], [9], [10].

In most cases, a DTO is associated with a specific entity. Here, we have an additional case where the **Current Info View** is aggregated from several entities. **`CurrentInfo01ViewDTO.java`** is the parent class. Depending on the package plan (01, 02, 11, 12, 13, 14), or user role (ADMIN, CUSTOMER) we have several Current Info Views - child DTO classes.

5.4. DATA VALIDATION

In this application, input data validation must be done before storing/retrieving data into/from the database. Some validations refer to field validation and other to presence/absence of particular database data.

a) Field match validation - **`FieldMatch.java`** interface that defines the *FieldMatch* annotation and **`FieldMatchValidator.java`** class that enforces field matching rules (example [11]).
In this project, we use the `phoneNumber` and `password` field match validation.
Password match validation is common in almost every application.
In this application phone number is the most important data; it is the primary key or foreign key to almost every db table. Due to this importance, it is necessary to check the input data for the phone number.

NOTE: Password validation is not used with the **`ConstraintValidator`** interface that enforces password validation rules. It could be used in this app to improve performances.

Also, the **`ConstraintValidator`** interface can be used for rules that define phoneNumber (the phone number depends of the country, region, mobile provider and other parameters). Only basic regex checking is used here (number length is 10, leading 0 and the second number is between [1-9]).

b) Depending on the condition, the presence/absence of certain database data can be handled by retrieving valid data, or by throwing a specific exception (usually that item not found, or an illegal item field).

`JsonDateTimeDeserializer.java` is used to validate the input data of the **`LocalDateTime`** class.

5.5. SECURITY

`SecurityFilterChain` in **`SecurityConfig.java`** is a sequence of filters that Spring Security applies to each incoming HTTP request. These filters work together to perform various security-related tasks. The authorized URLs for both the ADMIN role and the CUSTOMER role are defined here.

If the user is not authorized for the given resources, an exception will be thrown and **`CustomAccessDeniedHandler.java`** will be called.

Also, if the user is not authenticated, an exception will be thrown and **`CustomAthenticationEntryPoint.java`** will be called.

`CustomUserDetailsService.java` is used to retrieve user-related data (username, password, authorities)

`CustomAuthorisationFilter.java` filter accesses the header of the request and determines if the **Authorization** key exists in the header. If the **authorizationHeader** is not null and starts with **`Bearer`**, it means that a token information exists. The filter checks the validity of the token, and if valid, it updates the **`SecurityContextHolder`** with the request user information.

`AuthenticationService.java` grants JWT token to valid user (with valid username and valid password).

More about Spring Security [12], [13], Security Filter Chain [14], what is JWT [15] and securing REST api with Spring Security and JWT [16],[17],[18].

5.6. EXCEPTION

The **`@ControllerAdvice`** annotation is used to define a class that will be called whenever an exception is thrown in the application. This class contains multiple methods, annotated with the **`@ExceptionHandler`** annotation, each one of them being responsible for handling a specific exception. [19], [20]

`CustomRestAPIExceptionHandler.java` class has several methods for handling exceptions.

5.7. RESPONSE

`RestAPIResponse` represents response parameters that can be viewed in JSON format. There are also **`SecurityResponse`** parameters that can be viewed in JSON format when a security exception is thrown.

5.8. SERVICE

Each controller has its own service methods that process data retrieved from the database or prepare data for insertion into the database. Services represent the business logic of this application. Service methods contain validation services, mapping and repository operations for handling user requests.

There is a specific exception, **`SchedulingConfiguration.java`**, that works independently of the controllers. The aim of this class is to automatically generate package frames and monthly bills at the beginning of the month. More about scheduled jobs [21], [22].

5.9. CONTROLLER

REST controllers pass data to services for further processing. There are several REST controllers:

`AccessController.java`
`AdminCustomerController.java`
`UserPhoneController.java`
`PackagesAddOnsPhoneServicesController.java`
`FramesSDRController.java`
`CurrentInfoMonthlyBillFactsController.java`
`ClientController.java`

`AccessController.java` is the entrance of the application; any **valid User** - with a valid username and password receives the JWS token; with a valid JWS token the user can access different URLs, which are defined in the Spring Security configuration.

ADMIN role User can send requests to all REST controllers except **`ClientController.java`**; Besides **`AccessController`**, **CUSTOMER role User** can only send requests to **`ClientController.java`**.

5.10. SWAGGER UI

Swagger UI is a well-known open-source tool for visualizing and interacting with RESTful APIs. The Swagger view is divided into several parts that represent controllers. Each controller is presented with its own service methods. The UI takes input data and sends them to service methods as parameters or JSON objects. The response is displayed in the JSON format.

More on Swagger UI [23] and SpringBoot REST api doc using Swagger [24], [25].

6. TESTING

Here, we have several test types:

`@DataJpaTest` "sliced" integration tests are used for repository testing.

For validation services **`JUnit`** testing with **`Mockito`** library is used.

For services, integration tests are used with **`@SpringBootTest`** and **`webEnvironment`** set to **`NONE.`** In that case Spring Boot creates an application context that does not contain servlets. **There is no need for servlet layer when we test the service layer method.**

To test thr controller layer the **`@WebMvcTest`** "sliced" tests are used. If we don't want to test other parts of application, we mock the services with the **`@MockBean`** annotation.

When we are writing integration tests that verify our application by accessing one of our HTTP endpoints we use configuration **`@SpringBootTest`** and **`webEnvironment`** set to **`RANDOM_PORT.`** Spring creates **`WebApplicationContext`** for our integration tests and starts the embedded servlet container on a random port. We can then inject auto-configured HTTP (**`TestRestTemplate`**) clients that point to the started application.

A real test database "**`test_intro_telecom`**" is used for particular types of testing (for repository, service layer, the whole application). Test properties are defined in **`"application-test.properties"`** file.

The script file that creates test database "**`create_test_db.sh`**" is given in the code.

For database and tables creation, "**`test_intro_telecom_01_database.sql`**" and "**`test_intro_telecom_02_tables.sql`**" are used, respectively.

The basic data needed to set roles, package plans, add-on service codes and codes for service detail records are defined in "**`test_intro_telecom_03_basic_data.sql`**". Test data are given in "**`test_intro_telecom_04_test_data.sql`**".

More about testing [26],[27],[28],[29],[30],[31],[32],[33],[34],[35],[36],[37].

7. IN BRIEF

This document describes development of the Spring Boot REST API application "**introtelecom**" with Maven dependencies and Swagger UI. The simplified demo application is intended to demonstrate and explain the basics of mobile services (basic management). PostgreSQL database is used. Spring features are used for application startup, security, data exchange, validation and controller access. Swagger UI is used to receive user requests and display REST responses in JSON format.

8. REFERENCES

[1] https://dzone.com/articles/spring-boot-architecture-and-workflow

[2] https://www.geeksforgeeks.org/spring-boot-architecture/

[3] https://levelup.gitconnected.com/understanding-spring-boot-architecture-6083e2631bc6

[4] https://www.redhat.com/en/topics/api/what-is-a-rest-api

[5] https://github.com/cbarkinozer/OnlineBankingRestAPI

[6] https://www.youtube.com/watch?v=BwTq_PTRzsU

[7] https://www.baeldung.com/mapstruct

[8] https://reflectoring.io/java-mapping-with-mapstruct/

[9] https://stackabuse.com/guide-to-mapstruct-in-java-advanced-mapping-library/

[10] https://bootify.io/spring-data/mapstruct-with-maven-and-lombok.html

[11] https://memorynotfound.com/field-matching-bean-validation-annotation-example/

[12] https://reflectoring.io/spring-security/

[13] https://howtodoinjava.com/spring-security/inmemory-jdbc-userdetails-service/

[14] https://medium.com/@tanmaysaxena2904/spring-security-the-security-filter-chain-e09e1f53b73d

[15] https://www.miniorange.com/blog/what-is-jwt-json-web-token-how-does-jwt-authentication-work/

[16] https://www.vincenzoracca.com/en/blog/framework/spring/jwt/

[17] https://medium.com/@inni.chang95/spring-security-implementing-jwt-authentication-in-a-restful-spring-boot-application-657493fc0ae6

[18] https://www.toptal.com/spring/spring-security-tutorial

[19] https://backendstory.com/spring-security-exception-handling/

[20] https://nikhilsukhani.medium.com/mastering-exception-handling-in-spring-boot-using-controlleradvice-and-exceptionhandler-e676b5dd62ed

[21] https://reflectoring.io/spring-scheduler/

[22] https://lankydan.dev/2018/02/04/running-on-time-with-springs-scheduled-tasks

[23] https://apidog.com/blog/what-is-swagger-ui/

[24] https://www.geeksforgeeks.org/spring-boot-rest-api-documentation-using-swagger/

[25] https://bell-sw.com/blog/documenting-rest-api-with-swagger-in-spring-boot-3/

[26] https://medium.com/@VAISHAK_CP/testing-spring-boot-applications-strategies-and-best-practices-ec4b34bc63b0

[27] https://www.baeldung.com/spring-boot-testing

[28] https://reflectoring.io/spring-boot-test/

[29] https://ashok-s-nair.medium.com/java-integration-testing-a-spring-boot-service-e44bdf1ddaa7

[30] https://www.arhohuttunen.com/spring-boot-webmvctest/

[31] https://www.arhohuttunen.com/spring-boot-integration-testing/

[32] https://rieckpil.de/spring-boot-unit-and-integration-testing-overview/

[33] https://rieckpil.de/guide-to-springboottest-for-spring-boot-integration-tests/

[34] https://stackoverflow.com/questions/58364490/difference-between-webenvironment-random-port-and-webenvironment-mock

[35] https://www.javaadvent.com/2023/12/mastering-spring-security-integration-testing-for-your-apps.html

[36] https://stackoverflow.com/questions/62984669/junit-test-cases-for-javax-custom-validator

[37] https://github.com/eugenp/tutorials/blob/master/jackson-modules/jackson-custom-conversions/src/test/java/com/baeldung/deserialization/CustomDeserializationUnitTest.java

NOTE: References were last updated on 19th July 2024.