THE SPRING BOOT APPLICATION “DoctorPracticeDemo” - large integration

1. INTRODUCTION

The aim of the Spring Boot application “DoctorPracticeDemo” is to provide the way to schedule the doctor’s appointments and create, review and share medical reports.
This application has three distinct roles:

1. ***Administrator***, whose role is to control the whole application,
2. ***Doctor***, whose role is to create medical reports and open free terms for appointments,
3. ***Patient***, who can see his/her own medical reports and reserve the term of an appointment.

The project integrates many technologies into it. Some of them have already been introduced in my previous two projects “**VideoKlubProjekat**” [56] and “**NewVideoClubProject**” [57].

This document contains a short description of coding technologies used in application design, application rules, configuration and design details.

2. LIST OF TECHNOLOGIES

- **Spring Boot** – packaging and deployment
- **Spring MVC framework** – business layer
- **Spring Security framework** – for logging and access activities
- **Spring Data JPA** – ORM and transaction-driven data handling
- **Spring WebSocket and messaging** – for asynchronous message-driven multi-user communication (with SockJS functionality and Stomp functionality for chatting)
- **Spring Boot Validation** (with **Hibernate Validator**) – for server-side validation of input fields
- **Spring Email** – for sending emails
- **Hibernate** ORM– used together with Spring Data JPA
- **PostgreSQL** – RDBMS data layer
- **Thymeleaf** - Java template engine for processing and creating HTML, XML, JavaScript, CSS, and text, all in a servlet-compatible way. A more advanced alternative to JSP.
- **Bootstrap + Jquery**: front-end framework (css + js)
    - AJAX in front-end via jQuery standard library AJAX support
    - JSON data handling with **Jackson** library
- **Flying Saucer** as a CSS renderer that supports PDF output with **iText 5.x**.
- **Slf4j-with-Logback** library – for (server-side) logging

A standard Java servlet container (Tomcat) is used as a server platform.
Maven is used as build system.
The code is Java 8 – compliant.

3. APPLICATION’S FUNCTIONALITY AND USER ROLES

User authorization model uses the classic ‘user role’ paradigm. There are 3 roles implemented:

a. Administrator role:
- Has a CRUD functionality on admins, doctors and patients; also Add and Edit functionalities have input form validation
- Has a complete list of patients and their records; can view every patient’s single record (medical report) and print it to PDF.
- Has a complete list of doctors; can view every doctor’s appointment list and a scheduler – a schedule 4 weeks form for appointment terms.
- Has a ‘chat corner’ to communicate with other users (doctors/patients)
- Can change own password (with input form validation)

b. Doctor role:
- Can view his/her own user and personal data
- Has a complete list of patients and their records; can view every patient’s record (medical report) and print it to PDF.
- Can create new record for a patient and, after input data validation, save that report into database.
- Has a view to own appointments list and scheduler – ability to schedule appointment terms for up to 4 weeks in advance.
- Can open/cancel appointment terms.
- Has a ‘chat corner’ to communicate with other users (admin/doctors/patients)
- Can change his/her own password (with input form validation)
- Can send email to admin (with input data form validation)

c. Patient role:
- Can view his/her own user and personal data
- Has a complete list of own records; can view each of his own records in detail (medical report).
- Has a complete list of doctors available; can view every doctor’s scheduler – a schedule form for appointment terms up to 4 weeks in advance.
- Can reserve/cancel opened terms, for each of the available doctors.
- Has a ‘chat corner’ to communicate with other users (admin/doctors/patients).
- Can change his/her own password (with input form validation).
- Can send email to admin (with input data form validation).

Application controls:

- **Login** (login control with the support of Spring Security framework and BCrypt password encoding feature)
- **User page access** (access control with the support of Spring Security framework)
- **Simultaneous** admin/multi-user (doctors/patients) **work on schedule**, with information update in real-time (support of Spring WebSockets mechanism through event handling)
- **Session timeout** (with HTTP Session Listener and Spring Security)
- **Multiple concurrent logins prevention** (with the support of Spring Security)
- **Input data form validation** and **custom validation** (support of Spring Boot Validation)
- **Chatting activities** in real-time (support of Spring WebSockets mechanism through chat controller)
- **Sending email** by Gmail Server (support of Spring Email)
- **Creation of PDF reports**

Schedule rules

Different role participants have different activities:

Doctor:

Can open and cancel the terms in his/her own scheduler. When doctor opens the term a blue color is set. If doctor cancels the term, it will be returned into “colorless” state. If patient reserves the term, it becomes highlighted in green.

Doctor can cancel the opened (blue) term only. If the term has already been reserved by the patient (green term), doctor can’t cancel the term.

Patient:

Can reserve and cancel the opened (blue) terms. When patient reserves the opened term, the green color is set. If patient cancels the term, the green term returns to the blue term. Then, another patient can reserve this term.

In general, patient can view the opened (blue) terms and his/her own reserved (green) terms only. When another patient reserves opened term, the term is set to “colorless” state.

Administrator:

Administrator has no active role. He/she can track the scheduled activities of every doctor. Also, patients’ activities can be tracked by Admin. Opened terms are blue and reserved terms are green. When doctor cancels the term, it loses its highlight color and becomes “colorless”. When patient cancels the term, blue color is set, thus making it ‘available’ again for other patients.

4. DESIGN CONSIDERATIONS

Application has a 'classic' 3-tier structure.

- First tier is creation of the PostgreSQL database named "**doctor_practice**" , together with setting the primary data needed for the application startup.
- Mid-tier, "**DoctorPracticeDemo**", was designed as a Maven Project in Eclipse IDE. The first step was creation of skeleton Spring Boot application in Spring Initializr [2]. For start we needed some dependencies such as: PostgreSQL, Web, JPA, Security and Thymeleaf. After creation, the skeleton app was imported in Eclipse [3],[4],[5]. Other dependencies were introduced during application development.
- For front-end tier, ***Thymeleaf***, a Java template engine for processing and creating web pages is used.

4.1. DESIGN OF DATABASE (PERSISTENCE LAYER)

The scheme of PostgreSQL database "**doctor_practice**" is given in folder **`doc/`**.
SQL script file is given in folder **`sql/`**.

The initial idea for database design was taken from the site **Database Answers**. In the section **20. Reservation**, data model **Doctors Practice** was introduced [1]. This original model was modified to fit the new application requirements.

Database model has 9 data-tables:
`user_data` **–** for storing data of every user that interacts with application
`role` – ADMIN, DOCTOR, USER
`user_role` – for assigning role_id to every user
`record_type` – PHEX (physical examination), MDPR (medical prescription), RTSP (referral to specialist)
`patient_record` – medical reports of patients
`patient` – patient's data
`doctor` – doctor's data
`appointment_status_code` – CNLD (cancelled term by doctor), RSVD (reserved term by patient), FREE (free term opened by doctor), OTHR (other action).
`appointment` – for storing data about every single appointment term

1. For start we need to populate table '**`role`**':

```
INSERT INTO "public"."role" (role_id,role_type) VALUES
       (1,'ADMIN'),
       (2,'DOCTOR'),
       (3,'USER');
```

Also we need to populate table '**`record_type`**':

```
INSERT INTO "public"."record_type"
(component_code,component_description) VALUES
      ('PHEX','physical examination'),
      ('MDPR','medical prescription'),
      ('RTSP','referral to specialists');
```

And table '**`appointment_status_code`**':

```
INSERT INTO "public"."appointment_status_code"
(app_status_code,app_status_description) VALUES
      ('CNLD','cancelled term by doctor'),
      ('RSVD','reserved term by patient'),
      ('FREE','free term opened by doctor'),
      ('OTHR','other actions');
```

Spring Security provides the `PasswordEncoder` interface that uses *bcrypt* hashing function to encode the password. In this app, the utility class **`QuickPasswordEncodingGenerator.java`** was used to create first encrypted administrator password.

2. After creation of encrypted administrator password, we manually populated data in table '**`user_data`**' for the admin:

```
INSERT INTO "public"."user_data"
(user_id,username,password,email,address,telephone,user_status)
VALUES (1,'sneza','***','sneza@snnp.com','Nušićeva 93,
Beograd','7865435','ACTIVE');
```

where

'***' - represents encrypted administrator password supplied from utility class **`QuickPasswordEncodingGenerator.java`**

Also, we need to bond the unknown patient to free terms (the terms that do exist but have not yet been reserved by any patient). For that purpose we created the patient named '**unknown**', first in '**`user_data'`** table:

```
INSERT INTO "public"."user_data"
(user_id,username,password,email,address,telephone,user_status)
VALUES (9,'unknown','***','unknown@unk.un','No
address','100000','INACTIVE');
```

where

'***' - represents encrypted 'unknown' patient password and in table **'`patient`'**:

```
INSERT INTO "public"."patient"
(patient_id,hs_code,gender,birth_date,first_name,last_name,other,user
_id) VALUES (3,'00000000000','M',{d '2018-01-
01'},'Unknown','Unknown','This is a patient generated for Appointment
Records creation only.',9);
```

3. Finaly, for role assigning, in table '**`user_role`**' we inserted id of admin and id of his role (ADMIN):

```
INSERT INTO "public"."user_role" (user_id,role_id) VALUES (1,1);
```

and for 'unknown patient' (role – USER):

```
INSERT INTO "public"."user_role" (user_id,role_id) VALUES (9,3);
```

4.2. PROJECT CREATION AND PACKAGES

This application was created and developed as a Maven Project in Eclipse IDE.

Skeleton application has the main package **`com.snezana.doctorpractice`**, **`pom.xml`** – a main unit of work in Maven and Spring Boot application class **`DoctorPracticeDemoApplication.java`**.

`pom.xml` file has some dependencies that has already been included during the creation of Spring Boot application through *Spring Initializr*. Another dependencies: Spring WebSockets and messaging, JSON, Jackson, webjars (for Bootstrap, jQuery, Sockjs and Stomp), Spring Email and Flaying Saucer (for PDF creation) were included afterwords.

The packages in **`com.snezana.doctorpractice`** are:
- configurations
- controllers
- dto
- models
- repositories
- services
- utils
- validation

Under the package **`src/main/resources`** we created **`application.properties`** file which provides property values for database connection and mail support. Also, in that package there are the following directories:
- fonts
- static
- templates

In **`application.properties`** we set various application-wide properties, including e.g. our username and password for Gmail service.

5. DESIGN PARTS

According to the application demands and user roles we have several different parts integrated into the Spring Boot application.

5.1. MODELS

In **`models`** package we have entity classes that correspond to tables in database.

class	table in db
`Appointment.java`	**`appointment`**
`AppointmentStatusCode.java`	**`appointment_status_code`**
`Doctor.java`	**`doctor`**
`Patient.java`	**`patient`**
`PatientRecord.java`	**`patient_record`**
`RecordType.java`	**`record_type`**
`Role.java`	**`role`**
`User.java`	**`user_data`**

NOTE: **`user_role`** table is join table and has no relevant entity class.

`ChatWSMessage.java`, **`ChatWSOutputMessage.java`** are message models for chat activities and **`EvtWSMessage.java`** for scheduler activities.

5.2. SPRING SECURITY AND HTTP SESSION

In **`configuration`** package **`SecurityConfiguration.java`** class was created. It represents *Spring Security Java Configuration*. This configuration creates the *SpringSecurityFilterChain* which is responsible for all the security (protecting the application URLs, validating submitted username and passwords, redirecting to the log in form, etc) within our application.

In this file there is a definition for **`PasswordEncoder`** bean that uses *BCrypt* strong hashing function. Also *HttpSecurity* was configured. For authentication failure we have login error page. **`accessDenied`** page and **`multyConcLoginsExp`**, expired page for concurrent logins attempted by the same user are also defined. For session timeout detection **`CustomFilter`** was defined.

In the same package we have a **`CustomSuccessHandler.java`** for url determination where to redirect the user after login, based on the role of user.

In **`services`** package, **`CustomUserDetailsService.java`** is responsible for providing authentication details to Authentication Manager.

For HTTP session tracking and session timeout detection, **`SessionConfiguration.java`**, **`SessionListener.java`** and **`CustomFilter.java`** classes were created in **`configuration`** package (more about listeners in Spring Boot [6], [7]). Also in **`utils`** we have utility class **`TimeOut.java`** (singleton pattern) that supports session timeout detection.

5.3. INPUT DATA VALIDATION AND DTO (Data Transfer Objects)

Standard for performing validation is *Hibernate Validator*, which is the most widely used implementation of the *Bean Validation* specification. Creating sample spring-boot project with **`spring-boot-starter-web`** dependency, we add *hibernate-validator* dependency into our classpath, so we don't need any other dependencies. (More about Spring Bean Validation [21] – [27]).

Bean Validation works by defining constraints to the fields of a class by annotating them with specific annotations. Then, you pass an object of that class into a Validator which checks if the constraints are satisfied [24].

In this application, input form data validation has to be performed before storing data into database. Some of the validations are related to fields validation of a particular object and others are custom validations. For custom validations we need the interface that defines custom annotation and validator class that enforces the rules for validation. (More about the rules and regex in [27], [28], [29]).

To satisfy the just mentioned validation constraints, the **`validation`** package for custom validation was created. In this package we have two custom validations:

a) password validation – **`ValidPassword.java`** interface that defines *ValidPassword* annotation and **`PasswordConstraintValidator.java`** class that enforces the rules for validation. The rules for password validation, for this project, are:
- arbitrary number of lowercase characters
- At least one uppercase character
- At least one digit character
- At least one special character
- Rules for determining if a password contains a numerical, alphabetical or QWERTY keyboard sequences ('1234...', 'abcd….', 'asdf...')
- Rule for determining if a password contains whitespace characters

b) password match validation - **`PasswordMatches.java`** interface that defines *PasswordMatches* annotation and **`PasswordMatchesValidator.java`** class that enforces the rules for password matching.

In package **`dto`** *Data Transfer Object* classes were created. There are many interpretations of DTO objects in Java and this issue sometimes can cause ambiguities in code [30] - [34]. In our application, the form definition of these objects, was arranged to the basic DTO rules [31]:
- A data transfer object (DTO) is an object that is used to encapsulate data, and send it from one subsystem of an application to another.
- A DTO's data can be aggregated from several database tables or other data sources.
- The main benefit of using DTOs is that they can reduce the amount of data that needs to be sent '*across the wire*' in web applications.

Considering this, we have several DTO classes with annotations used for field validation. In that way, the entity classes are not overloaded with annotations. We have DTO objects (**`DoctorUserDto.java, DoctorUserEdto.java, PatientUserDto.java, PatientUserEdto.java, PatientRecordDto.java, UserDto.java, UserEdto.java`**) that collect data from input forms and check and send them to relevant model objects.

In addition to these classes there are:
`MailDto.java` - for mail validation
`AppointmentView.java` - for retreiving data into scheduler view
`ChangePasswordDto.java` - for password changing

5.4. SPRING DATA JPA AND REPOSITORIES

Spring Data JPA provides repository support for the Java Persistence API (JPA). It eases development of applications that need to access JPA data sources [8]. In brief, this feature provides *Spring Data* repository interfaces which are implemented to create JPA repositories. It reduces the amount of boilerplate code required to implement data access layers for various persistence stores.(More about Spring Data JPA [8] – [11]).

Every entity in **`model`** package has its corresponding repository interface.

class	repository interface
`Appointment.java`	**`AppointmentRepository.java`**
`AppointmentStatusCode.java`	**`AppointmentStatusCodeRepository.java`**
`Doctor.java`	**`DoctorRepository.java`**
`Patient.java`	**`PatientRepository.java`**
`PatientRecord.java`	**`PatientRecordRepository.java`**
`RecordType.java`	**`RecordTypeRepository.java`**
`Role.java`	**`RoleRepository.java`**
`User.java`	**`UserRepository.java`**

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

In addition to that, **`UserRepository.java`** uses the *Manual Custom Query* methods (via the **@Query** annotation). There, we have manually defined queries to fetch the data, and the queries are then bound with the relevant **`findBy`**… method.

5.5. TWO-WAYS SPRING WEBSOCKET FEATURE

We use WebSocket in web applications where the client and server need to exchange events at high frequency and with low latency, that is when information has to be updated in real-time. (More about Spring WebSocket feature [12] – [20] and [57]).

In this project Spring WebSocket feature was used in two different ways:
1) for ***appointment schedule*** - handling WebSocket events
2) for ***chat activities*** – using chat controller for sending and receiving messages

Handling WebSocket events

For scheduling of appointments we need real-time event that changes color of appointments. We have activities such as opening/cancelling terms by doctors and also reserving/cancelling terms by patients that produce color change of terms in schedule. For these activities, Spring WebSocket event handling was used.

In **`models`** package we have **`EvtWSMessage.java`** class as a message model. Messages as Java Objects are exchanged between the (multiple) clients and the (single) server.

In **`configurations`** package there is a Java configuration class **`EventWSConfig.java`**. The bean **`eventWSHandler`** maps WebSocket handler to a specific URL and enables *SockJS* functionality. SockJS allows for applications to use a WebSocket API, but also enables fallback options for browsers that don't support WebSocket protocol. SockJS uses JSON formatted arrays for messages.

In the same package, we have **`EventWSHandler.java`** class, a WebSocket server that extends **`TextWebSocketHandler`** class. In method **`sendEventWS()`** serialization of Java Object into JSON string is made.

In **`services`** package there is a service **`EventWSService.java`** which handles WebSocket events.

Chat activities

This application has a 'chat corner'. The chat application example [14] was used as startup code for this functionality.

In **`models`** package **`ChatWSMessage.java`** class represents a message model. Also, we have **`ChatWSOutputMessage.java`** class as a output message format for chat with the displayed time.

In **`configurations`** package there is a Java configuration class **`ChatWSConfig.java`**, where WebSocket endpoint with SockJS functionality and message broker is configured. Also we have **`ChatWSEventListener.java`** class which is a way to listen for socket connect and disconnect events.

In **`controllers`** package we have **`ChatController.java`** controller for sending and receiving messages. There are three methods there: **`sendMessage()`** for sending message where recipient gets the output message with displayed time, **`addUser()`** for adding username in chat session, and **`alreadyHere()`** for presence notification in chat corner.

Front-end activities

The second part of Spring WebSocket functionality is related to front-end activities. The specific code written in JavaScript was put in html pages (more about it in chapter 5.10.).

5.6. SPRING MAIL

Doctors and patients can send email to the administrator.
If we want to use Spring Email feature [35] - [40], **`application.properties`** file should be populated first, with relevant parameters (host, port, username, password …).

In **`dto`** package we have **`MailDto.java`** email format for sending emails. Some fields have Hibernate validator annotations due to the input form validation.

In **`configurations`** package there is a **`MailConfig.java`** configuration class. That class has a bean **`javaMailSender()`** that supports MIME messages. The sender was customized by configuration items from the **`spring.mail`** namespace, for Gmail configuration. Also we have **`MailComponent.java`** class for setting and sending email in html template (**`templates/fragments/emailHTML.html`**) [40].

NOTE: Before starting application, we need to populate **`application.properties`** file with our own mail parameters: username and password. Also, in **`MailComponent.java`** we should set our gmail address.

5.7. PDF ACTIVITIES

This project has an option to generate medical reports (records) in PDF format. To perform this task we have back-end and front-end activities.

To create PDF from HTML we use *Thymeleaf* as a template engine and *Flying Saucer* library as a XHTML renderer [41], [42], [43]. In **`configurations`** package there is a class **`ThymeleafTemplResConfiguration.java`** with **`templateResolver()`** bean that resolves html template.

In **`utils`** package we have **`PdfGeneratorUtil.java`** class and **`createPdf()`** method that provides template name, generates PDF and saves it to temporary location in our system. **`createPdf()`** method uses *Thymeleaf* to render HTML template and *flying-saucer-pdf* dependency uses this HTML to generate PDF.

Front-end activities

Front-end page **`pdfReportTempl.html`** which includes fragment **`pdfReport.html`** is set for PDF creation (chapter 5.10.).

5.8. USER SERVICE

In **`services`** package we have **`UserService.java`** interface and its implementation **`UserServiceImpl.java`**. Methods involved in interface were commented. These methods were used in service methods of controllers: **`LoginController.java`**, **`AdminController.java`**, **`DoctorController.java`** and **`PatientController.java`** through **@Autowired** annotation.
Static methods from **`SomeUtils.java`** utility class for date, time and string manipulations are also used in implemented methods.

5.9. CONTROLLERS

Controllers are the crucial part of application. We have different controllers for different user roles:

`LoginController.java`
`AdminController.java`
`DoctorController.java`
`PatientController.java`

`ChatController.java` was already described in chapter 5.5.

Each controller has its own service methods that process data taken from database or prepare data for insertion into database, and returns view names to the *DispatcherServlet* for view resolving.
Also, static methods from **`SomeUtils.java`** utility class are used in service methods.

Every service method of each particular controller has a proper Javadoc comment.

5.10. THYMELEAF AND HTML PAGES

Thymeleaf is Java template engine for processing and creating HTML, XML, JavaScript, CSS, and text [54]. (More about Thymeleaf [46] – [55]).

In **`src/main/resources`** we have directories:
- **`fonts`** (for UTF-8 characters in pdf files)
- **`static`** (css, images, js)
- **`templates`** (with **`fragments`** and html files)

In the **`templates`** directory there are the html files. Subdirectory **`fragments`** containes html files that are the parts of other html files.

Application has the following html pages:

In **fragments** subdirectory:
- **`emailHTML.html`** – html template for email sending
- **`footer.html`** – footer fragment
- **`header.html`** – basic header fragment
- **`headincl.html`** – head sources
- **`pdfReport.html`** – fragment with data for pdf medical report

General html pages:
- **`welcome.html`** – welcome application page
- **`login.html`** – entrance form with data checking
- **`accessDenied.html`** – information about access attempts to pages for which you are not authorized to access
- **`multiConcLoginsExp.html`** – expired page due to multiple concurrent logins being attempted by the same user
- **`pdfReportTemplate.html`** – template for pdf medical report

Admin html pages:
- **`admin.html`** – the main admin page with links to the other admin pages
- **`addAdm.html`** – add new admin (user form)
- **`addDoc.html`** – add new doctor (user form and doctor form)
- **`addPtn.html`** – add new patient (user form and patient form)
- **`admAppnmt.html`** – list of doctors with links to scheduler/(appointment list) activities
- **`admAppnmtLst.html`** – appointment list view of particular doctor
- **`admChat.html`** – chat corner for admin
- **`admPatRec.html`** – list of patients with links to info/(view /print records)
- **`admSchdl.html`** – 4 weeks schedule view of particular doctor
- **`admViewPrintOneRecord.html`** – one medical report of particular patient with “print to pdf” activity

- **admViewPrintRecords.html** – records list of particular patient
- **changePassword.html** – change password page
- **editAdm.html** – update admin activity (user form)
- **editDoc.html** – update doctor activity (user form and doctor form)
- **editPtn.html** – update patient activity (user form and patient form)
- **infoDoc.html** – info page about particular doctor
- **infoPtn.html** – info page about particular patient
- **usersAdm.html** – list of administrators with links to add/edit/delete activities
- **usersDoc.html** – list of doctors with links to info/add/edit/delete activities
- **usersPtn.html** – list of patients with links to info/add/edit/delete activities

Doctor html pages:

- **doctor.html** – the main doctor page with links to the other doctor pages
- **createNewRecord.html** – page for creating and saving medical report
- **dInfoPtn.html** – info page about particular patient
- **docAppnmtLst.html** – appointment list
- **docChangePassword.html** – change password page
- **docChat.html** – chat corner for doctor
- **docMail.html** – email form for sending mails to admin
- **docPatRec.html** – list of patients with links to info/ (view /print records)/ (create new record) activities
- **docSchdl.html** – 4 weeks schedule view of appointments
- **viewPrintOneRecord.html** – one medical report of particular patient with “print to pdf” activity
- **viewPrintRecords.html** – records list of particular patient

Patient html pages:

- **home.html** – the main patient page with links to the other patient pages
- **patAppnmt.html** – list of doctors with link to scheduler activity
- **patChangePassword.html** – change password page
- **patChat.html** – chat corner for patient
- **patMail.html** – email form for sending mails to admin
- **patSchdl.html** – 4 weeks schedule view of particular doctor
- **patViewOneRecord.html** – view of one own medical report
- **patViewRecords.html** – records list of own medical reports

For css support we have **main.css** and **chat.css** (**static/css** directory).
header.html and **footer.html** pages have thymeleaf tags for Spring Security feature.

Also, thanks to the *Thymeleaf*, we can use static methods from **SomeUtils.java** class directly in html pages: **admSchdl.html, docSchdl.html and patSchdl.html** [50].

Chat activities

`admChat.html`, **`docChat.html`** and **`patChat.html`** pages have JavaScript code integrated in them, for WebSocket chat functions for connection, sending and receiving messages.

Chat activities have SockJS functionality and *STOMP* (*Simple Text Oriented Messaging)* support. It is a messaging protocol that defines the format and rules for data exchange.

The following chat activities were implemented: CHAT, JOIN, LEAVE and ALREADY(here).

Schedule activities

`admSchdl.html`, **`docSchdl.html`** and **`patSchdl`**.html pages have JavaScript code for WebSocket event handling. Schedule activities have SockJS functionality. The initial idea for changing content view with next and previous buttons was given in [44], [45].

Administrator can track appointment activities on schedule pages but without active role. Active roles are reserved for doctors and patients. Every doctor can open and cancel the terms in his/her own schedule. Also, every patient can 'take' and 'free' the terms which have already been opened by a doctor. Complete description of schedule rules is given in chapter 3.

Email

Doctors and patients can send email massages to administrator in html form. Pages **`docMail.html`** and **`patMail.html`** have input form validation. Html template for email massages is given in **`emailHTML.html`** page with css settings.

PDF report

Medical reports (records) can be generated in PDF form by administrator and doctors. For that purpose **`pdfReport.html`** page, which contains relevant data, serves as a fragment in **`pdfReportTemplate.html`** page. The second one is a template with css settings for pdf page creation.

6. IN SHORT

Thank you for your patience! In this document you read about the development of Spring Boot application "**DoctorPracticeDemo**" in Eclipse IDE with Maven dependencies. Through various aspects, this application has a large number of technologies integrated. Spring features are used for booting application, security, data exchange, controller approach, real-time events, chatting, email, validation and pdf file creation.

7. REFERENCES

[1] http://www.databaseanswers.org/data_models/doctors_practice/index.htm

[2] https://start.spring.io/

[3] https://dzone.com/articles/configuring-spring-boot-for-postgresql

[4] https://www.tutorialspoint.com/spring_boot/index.htm

[5] https://howtodoinjava.com/spring-boot-tutorials/

[6] http://theblasfrompas.blogspot.com/2016/08/httpsessionlistener-with-spring-boot.html

[7] https://www.concretepage.com/spring-boot/spring-boot-listener

[8] https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

[9] https://www.javacodegeeks.com/2018/05/spring-data-jpa-tutorial.html

[10] https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa

[11] https://www.petrikainulainen.net/spring-data-jpa-tutorial/

[12] https://docs.spring.io/spring/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/websocket.html

[13] https://sunitkatkar.blogspot.com/2014/01/spring-4-websockets-with-sockjs-stomp.html

[14] https://www.callicoder.com/spring-boot-websocket-chat-example/

[15] https://www.sitepoint.com/implementing-spring-websocket-server-and-client/

[16] https://www.baeldung.com/spring-websockets-send-message-to-user

[17] https://github.com/bbranquinho/wpattern-frameworks-spring-mvc-websocket

[18] https://stackoverflow.com/questions/27158106/websocket-with-sockjs-spring-4-but-without-stomp

[19] https://keyholesoftware.com/2017/04/10/websockets-with-spring-boot/

[20] https://www.devglan.com/spring-boot/spring-websocket-integration-example-without-stomp

[21] https://www.baeldung.com/spring-mvc-custom-validator

[22] https://memorynotfound.com/custom-password-constraint-validator-annotation/

[23] https://careydevelopment.us/2017/05/24/implement-form-field-validation-spring-boot-thymeleaf/

[24] https://reflectoring.io/bean-validation-with-spring-boot/

[25] https://www.baeldung.com/javax-validation

[26] https://stackoverflow.com/questions/1972933/cross-field-validation-with-hibernate-validator-jsr-303

[27] http://www.techburps.com/spring-framework/spring-form-validation

[28] http://www.vogella.com/tutorials/JavaRegularExpressions/article.html

[29] http://www.passay.org/javadocs/org/passay/package-summary.html

[30] https://martinfowler.com/eaaCatalog/dataTransferObject.html

[31] https://www.quora.com/How-can-I-use-data-transfer-object-in-Java

[32] https://softwareengineering.stackexchange.com/questions/171457/what-is-the-point-of-using-dto-data-transfer-objects

[33] https://softwareengineering.stackexchange.com/questions/373284/what-is-the-use-of-dto-instead-of-entity

[34] https://stackoverflow.com/questions/1051182/what-is-data-transfer-object

[35] https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-email.html

[36] https://www.baeldung.com/spring-email

[37] https://o7planning.org/en/11145/spring-email-tutorial

[38] https://www.youtube.com/watch?v=MgOdvqvF6gk

[39] https://www.youtube.com/watch?v=FgEx6TfuGDQ&list=PLrOGbFgcVOIXQWVcEps2TcCQlGsMeCdmM&index=3

[40] https://www.youtube.com/watch?v=115D68xbxQ0&list=PLrOGbFgcVOIXQWVcEps2TcCQlGsMeCdmM&index=4

[41] https://www.oodlestechnologies.com/blogs/How-To-Create-PDF-through-HTML-Template-In-Spring-Boot

[42] https://tuhrig.de/generating-pdfs-with-java-flying-saucer-and-thymeleaf/

[43] https://tuhrig.de/generating-pdfs-with-java-flying-saucer-and-thymeleaf-part-2/

[44] https://stackoverflow.com/questions/27780577/how-to-change-content-with-next-and-previous-button-at-same-page

[45] http://jsfiddle.net/chintansoni/maysmhbr/4/

[46] https://www.thymeleaf.org/doc/tutorials/2.1/thymeleafspring.html

[47] https://www.baeldung.com/spring-boot-crud-thymeleaf

[48] http://dtr-trading.blogspot.com/2014/02/spring-mvc-4-thymeleaf-crud-part-1.html

[49] https://doanduyhai.wordpress.com/2012/04/14/spring-mvc-part-iv-thymeleaf-advanced-usage/

[50] https://github.com/thymeleaf/thymeleaf/issues/593

[51] https://stackoverflow.com/questions/28559817/displaying-active-navigation-based-on-contextual-data/33811623

[52] https://www.youtube.com/watch?v=fz5tc8VWxq4

[53] https://stackoverflow.com/questions/25687816/setting-up-a-javascript-variable-from-spring-model-by-using-thymeleaf

[54] https://www.baeldung.com/spring-thymeleaf-3

[55] http://jvmhub.com/2015/08/02/spring-boot-with-thymeleaf-tutorial-part-2-forms-with-validation/

[56] https://github.com/petroneris/video-club-project , The Spring MVC application "VideoKlubProjekat"

[57] https://github.com/petroneris/new-video-club1 , The Spring MVC application "NewVideoClubProject" - different story

NOTE: References were last updated on 10th January 2019.