МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный исследовательский университет)

Институт №8

«Компьютерные науки и прикладная математика»

Кафедра 806

«Вычислительная математика и программирование»

Курсовой проект по дисциплине «Базы данных»

Тема: «Разработка базы данных, API и клиентского приложения для информационной модели "Спортивная секция"»

Студент: Чернышева С.П.

Группа: М8О-313Б-22

Преподаватель: Крылов С.С.

Оценка:

Дата:

Москва, 2024

Содержание

Введение

Данное веб-приложение, реализованное на языке программирования Java, предоставляет пользователю возможности онлайн бронирования кортов и получения информации о спортивной секции.

Администраторам и владельцу приложение позволяет контролировать аренду, действия авторизованных пользователей, расписание занятий и так далее.

Для хранения и управления данными использована СУБД – PostgreSql, а клиентское приложение реализовано с помощью фреймворка Vaadin.

Структура базы данных:

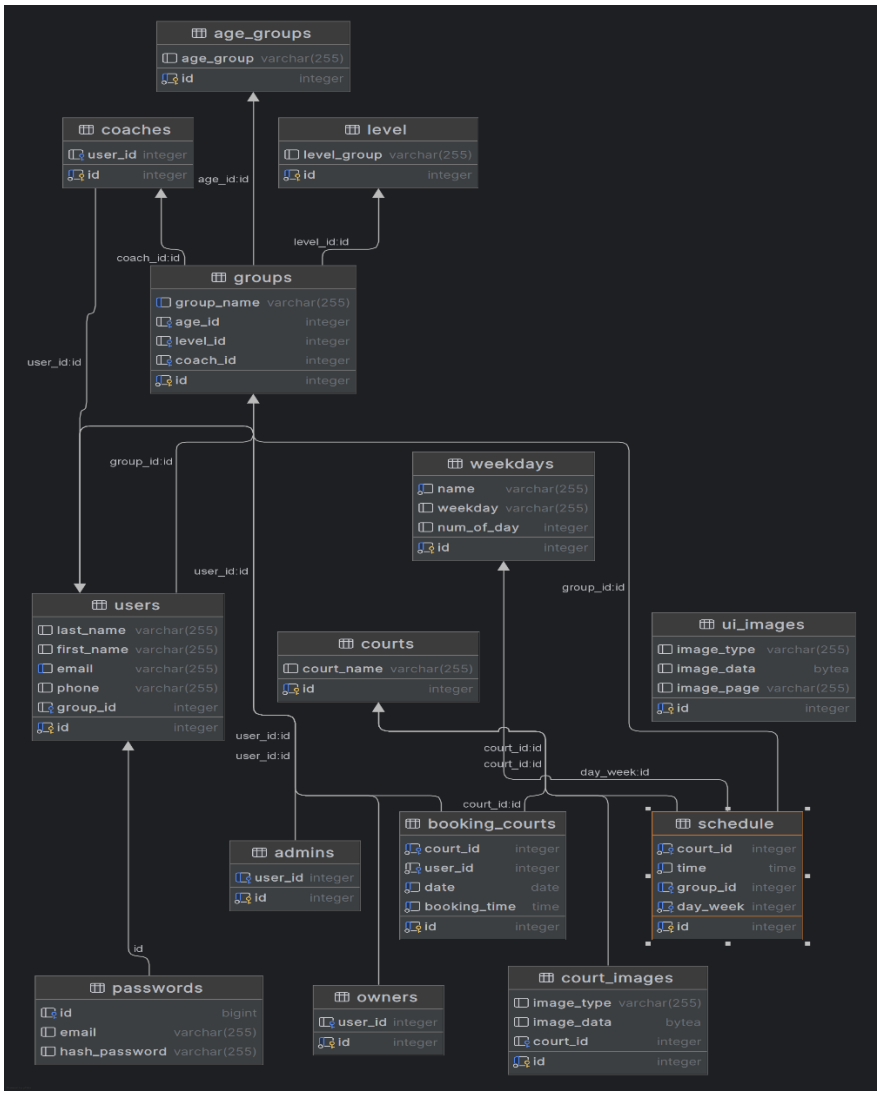

Рисунок 1. Структура базы данных

Чтобы иметь доступ к возможностям приложения, пользователь должен быть зарегистрирован и авторизован.

Роли в приложении разделены на следующие:

1. Рядовой пользователь
   Возможности:
   1) Бронирование кортов
   2) Доступ только к своим бронированиям
   3) Отменить аренду заранее
   4) Доступ к расписанию занятий групп и контактам тренеров
   5) Изменение своих контактов
2. Администратор
   Возможности:
   1) Все, разрешенные рядовому пользователю
   2) Доступ ко всем пользователям, их контактам, информации о группе и тренере в ней
   3) Доступ ко всем бронированиям, возможность отменить любое из них заранее
   4) Записать аренду корта на пользователя
   5) Изменение расписания, добавление новой группы, изменение тренера в группе
   6) Добавление пользователя в группу и исключение из нее
   7) Создание копии базы данных и восстановление ее из копии
   8) Просмотр загруженности кортов на текущий день
3. Владелец
   Возможности:
   1) Все, разрешенные администратору
   2) Право на изменение ролей пользователей

1. Авторизация и роли

1.1 Логика

Авторизация реализована с помощью SpringSecurity и хранения почты с зашифрованным паролем в базе данных.

Разделение ролей реализовано следующим образом:

В базе данных созданы 4 таблицы: всех пользователей - users, тренеров - coaches, администраторов - admins и владельцев - owners.

В таблице users хранятся основные данные о пользователе – имя, фамилия, номер телефона, почта и так далее.

Остальные таблицы имеют в качестве аттрибута внешний ключ на идентификатор пользователя в таблице users. Если его id есть в одной из перечисленных выше таблиц, значит помимо роли рядового пользователя ему добавляются соответствующие таблицам.

Эта проверка производится на этапе авторизации в конфигурации SpringSecurity.

Таблицы, созданные в базе данных для реализации задачи аутентификации:

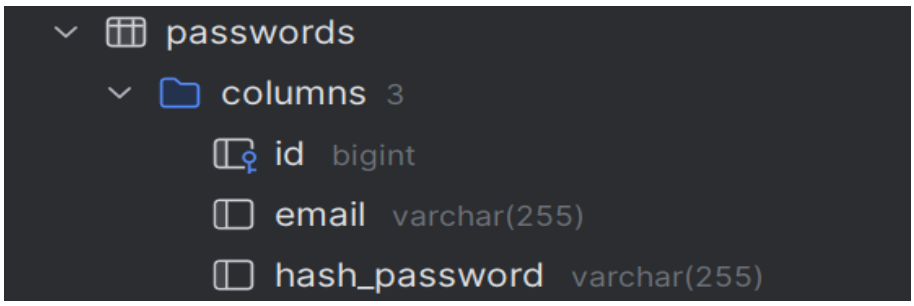

Рисунок 2. Таблица паролей

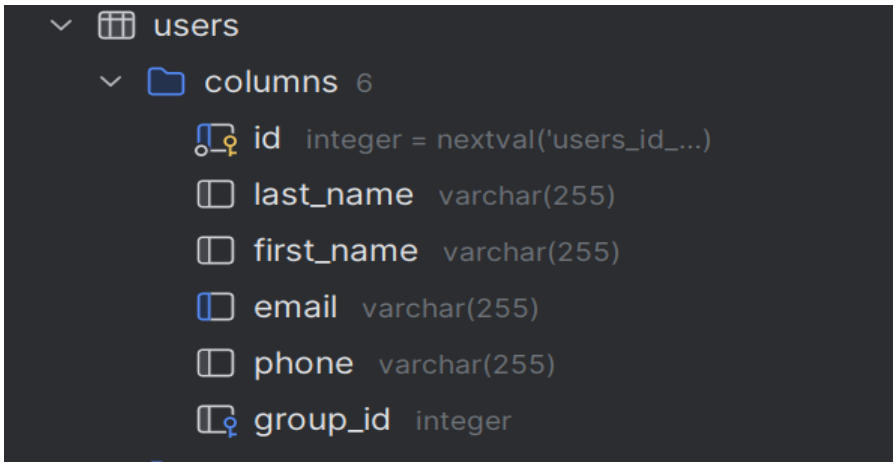

Рисунок 3. Таблица пользователей

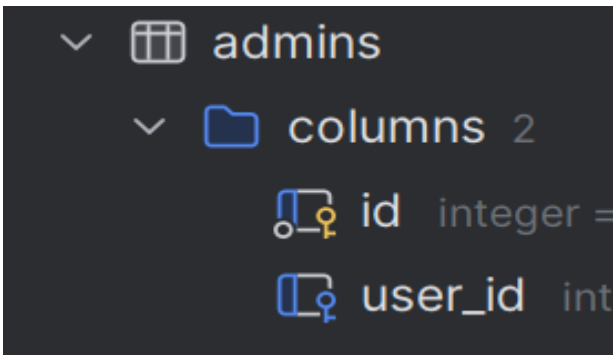

Рисунок 4. Таблица администраторов

1.2. Реализация

Базовый класс SecurityConfig представляет конфигурацию безопасности для приложения, реализованного с использованием Spring Security в Java. Основная задача этого класса - настройка процессов аутентификации и авторизации пользователей, включая использование JWT-токенов, работу с регистрацией/входом и доступом к различным уровням пользователя.

Метод passwordEncoder() возвращает экземпляр BCryptPasswordEncoder, который нужен для шифрования паролей пользователей.

AuthenticationManager обрабатывает процесс аутентификации пользователей (проверка входа, пароля и т.д.).

Метод securityFilterChain() задаёт правила для различных URL маршрутов:
- Разрешён доступ без авторизации к /login, /register, /error, /api/.
- Для /admin/ требуются права ADMIN или OWNER.
- Остальные запросы требуют аутентификации.

Метод UserDetailsService отвечает за загрузку данных о пользователе из хранилища.

1. Проверяет, есть ли в базе пользователь с указанным email.

Если пользователь не найден, выбрасывается исключение `UsernameNotFoundException`.

2. Получает роли пользователя (например, USER, ADMIN, OWNER) и добавляет их в SimpleGrantedAuthority.
3. Возвращает объект User, который используется внутри Spring Security для аутентификации.

Листинг #1. Реализация класса SecurityConfig

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    private final JwtRequestFilter jwtRequestFilter;

    @Autowired
    public SecurityConfig(@Lazy JwtRequestFilter jwtRequestFilter) {
        this.jwtRequestFilter = jwtRequestFilter;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity
http, UserDetailsService userDetailsService, PasswordEncoder
passwordEncoder) throws Exception {
        AuthenticationManagerBuilder auth =
http.getSharedObject(AuthenticationManagerBuilder.class);

auth.userDetailsService(userDetailsService).passwordEncoder(password
Encoder);
        return auth.build();
    }
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/login").permitAll()
                        .requestMatchers("/register").permitAll()
                        .requestMatchers("/error").permitAll()
                        .requestMatchers("/api/**").permitAll()

.requestMatchers("/admin/**").hasAnyAuthority("ADMIN", "OWNER")
                        //
.requestMatchers("/home").hasAnyRole("USER", "ADMIN")
```

```
                        .anyRequest().authenticated())
                .formLogin(form -> form
                        .loginPage("/login")
                        .successHandler((request, response,
authentication) -> {
                            // Перенаправление на нужный путь
                            response.sendRedirect("");
                        })
                )
                .logout(Customizer.withDefaults());

        http.addFilterBefore(jwtRequestFilter,
UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    @Bean
    public UserDetailsService
userDetailsService(IAuthorizeRepository authorizeRepository,
                                                    IUserRepository
userRepository, IAdminRepository adminRepository) {
        return email -> {
            Optional<UserModelAuthorization> userEntity =
authorizeRepository.getByEmail(email);
            if (userEntity.isEmpty()) {
                throw new UsernameNotFoundException("User not
found");
            }
            Optional<User> userOpt =
userRepository.findByEmail(email);
            if (!userOpt.isPresent()) {
                throw new UsernameNotFoundException("User not
found");
            }
            User user = userOpt.get();
            List<SimpleGrantedAuthority> authorities = new
ArrayList<>();
            authorities.add(new SimpleGrantedAuthority("USER"));
            if (user.getAdmin() != null) {
                authorities.add(new
SimpleGrantedAuthority("ADMIN"));
            }
            if (user.getOwner() != null) {
                authorities.add(new
SimpleGrantedAuthority("OWNER"));
            }
            if (user.getCoach() != null) {
```

```
                authorities.add(new
SimpleGrantedAuthority("COACH"));
            }
            return new
org.springframework.security.core.userdetails.User(
                    userEntity.get().getEmail(),
                    userEntity.get().getHashPassword(),
                    authorities
            );
        };
    }
}
```

SecurityUtils — это утилитарный (вспомогательный) класс для работы с данными текущего аутентифицированного пользователя в контексте безопасности приложения. Он использует Spring Security для получения информации об аутентификации, ролях и функциях пользователя.

Функционал:

1. Метод getCurrentUserEmail() возвращает email текущего аутентифицированного пользователя. Если пользователь не аутентифицирован, возвращается null.
2. Метод deleteAuth() удаляет текущую аутентификацию из контекста безопасности, фактически "выходя" из учётной записи пользователя.
3. Метод getUserRoles() возвращает роли (авторизации) текущего пользователя как Collection<? extends GrantedAuthority>. Если пользователь не аутентифицирован, возвращается Optional.empty().
4. Метод isAdminOrHigher() возвращает true, если у пользователя есть роль ADMIN или OWNER.
5. Метод getAllUserRoles() возвращает список всех ролей текущего аутентифицированного пользователя. Если пользователь не аутентифицирован, возвращается пустой список.

Листинг #2. Класс SecurityUtils

```
public class SecurityUtils {

    public static String
getCurrentUserEmail() {
        Authentication authentication =
SecurityContextHolder.getContext().getA
uthentication();
        if (authentication != null &&
authentication.isAuthenticated()) {
```

```
            Object principal =
authentication.getPrincipal();
            if (principal instanceof
UserDetails) {
                return ((UserDetails)
principal).getUsername(); // Возвращает
email, так как мы используем email в
качестве имени пользователя
            }
        }
        return null; // Если
пользователь не аутентифицирован
    }

    public static void deleteAuth() {

SecurityContextHolder.getContext().setA
uthentication(null);

SecurityContextHolder.clearContext();
    }

    private static
Optional<Collection<? extends
GrantedAuthority>> getUserRoles() {
        Authentication authentication =
SecurityContextHolder.getContext().getA
uthentication();
        if (authentication != null &&
authentication.isAuthenticated()) {
            Object principal =
authentication.getPrincipal();
            if (principal instanceof
UserDetails userDetails) {
                return
Optional.of(userDetails.getAuthorities(
));
            }
        }
        return Optional.empty();
    }

    public static boolean
isAdminOrHigher() {
        return getUserRoles().map(roles
-> roles.stream()
                        .anyMatch(role
-> role.getAuthority().equals("ADMIN")
```

```
||
role.getAuthority().equals("OWNER")))
                .orElse(false);
    }

    public static boolean isCoach() {
        return getUserRoles().map(roles
-> roles.stream()
                        .anyMatch(role
->
role.getAuthority().equals("COACH")))
                .orElse(false);
    }

    public static boolean isOwner() {
        return getUserRoles().map(roles
-> roles.stream()
                        .anyMatch(role
->
role.getAuthority().equals("OWNER")))
                .orElse(false);
    }

    public static List<String>
getAllUserRoles() {
        return getUserRoles()
                .map(roles ->
roles.stream()
                        .map(role ->
role.getAuthority())

.collect(Collectors.toList()))

.orElse(Collections.emptyList());
    }
```

2. Архитектура

2.1 Связь с базой данных. Репозитории

Для взаимодействия с базой данных использован фреймворк Hibernate, позволяющий удобно работать с данными из базы, используя объектно-реляционное отображение (ORM).

Репозиторий – слой взаимодействия приложения с базой данных, который содержит логику для выполнения операций доступа, модификации,

извлечения данных и управления их состояниями. Репозиторий выступает в роли посредника между бизнес-логикой приложения и базой данных.

Для каждой отдельной сферы – аренда кортов, расписание, пользователи и так далее, создан отдельный репозиторий.

Листинг #3. Репозиторий на примере IScheduleRepository

```
@Repository
public interface IScheduleRepository extends JpaRepository<Schedule,
Integer> {

    @Query(value = "SELECT * FROM GET_SCHEDULE_FOR_DAY_WEEK(:name,
:courtId)", nativeQuery = true)
    public List<Schedule> getScheduleOnWeekDayName(String name, int
courtId);

    @Query(value = "SELECT * FROM
GET_BOOKING_TIME_FOR_DAY_WEEK(:courtId, :weekDayId)", nativeQuery =
true)
    List<Time> getBookingTimeForDayWeek(int courtId, int weekDayId);
}
```

Для выполнения пункта о запрещении прямой передачи SQL-кода в базу данных используем для запросов процедуры и функции.

Листинг #4. Пример функции.

```
CREATE OR REPLACE FUNCTION GET_BOOKING_HOURS_BY_COURT_ID_AND_DATE(
    court_id_param INT,
    booking_date_param date
) RETURNS TABLE (times TIME) AS $$
BEGIN
    RETURN QUERY
        SELECT booking_time
        FROM booking_courts b
        WHERE b.court_id = court_id_param AND b.date =
booking_date_param;
END;
$$ LANGUAGE plpgsql;
```

Аналогично реализованы и другие репозитории.

2.2 Сервисы. Слой между UI и базой данных.

Сервис - слой приложения, который содержит бизнес-логику. Этот слой находится между репозиторием (слоем данных) и контроллером (слоем взаимодействия с пользователем).

Сервисы используются для обработки данных, исполнения операций и выполнения основного функционала приложения. Они позволяют сосредоточить логику в одном месте, что делает код более организованным и легко тестируемым.

Аналогично репозиториям, сервисы разделены на отдельные сферы. Каждый метод в них выполняется асинхронно для избежания блокировки потока.

Листинг #5. Пример сервиса

```
@Component
@Service
public class CourtService {
    private final ICourtRepository courtRepository;
    @Autowired
    public CourtService(ICourtRepository courtRepository) {
        this.courtRepository = courtRepository;
    }

    @Async
    public CompletableFuture<List<Court>> getCourtsAsync() {
        return
CompletableFuture.supplyAsync(courtRepository::findAll);
    }
    @Async
    public CompletableFuture<Optional<Court>> getCourtByIdAsync(int
id) {
        return CompletableFuture.supplyAsync(() ->
courtRepository.findById(id));
    }
}
```

2.3 Клиентское приложение. UI

Как упоминалось ранее, клиентское приложение реализовано с помощью Vaadin. Для каждой страницы – отдельный класс, который при необходимости вызывает нужный метод сервиса.

Листинг #6. Класс, служащий для отображения расписания занятий.

```
@Route("schedule")
@PageTitle("Расписание занятий")
public class ScheduleView extends HorizontalLayout {
    private final ScheduleService scheduleService;
    private final UserService userService;
    private final CoachService coachService;
    private final GroupService groupService;
    private User user;

    @Autowired
    public ScheduleView(ScheduleService scheduleService, UserService
userService, CoachService coachService, GroupService groupService) {
        this.scheduleService = scheduleService;
        this.userService = userService;
        addClassName("schedule-view");
        setSizeFull();

        String email = SecurityUtils.getCurrentUserEmail();
        Optional<User> userOpt =
userService.getUserAsync(email).join();
        if (!userOpt.isPresent()) {
            UI.getCurrent().access(() -> {
                Notification.show("Что-то пошло не так, попробуйте
авторизоваться еще раз", 5000, Notification.Position.MIDDLE);
                SecurityUtils.deleteAuth();
            });
        }
        User user = userOpt.get();
        List<Weekday> weekdays =
scheduleService.getAllWeekdaysWithSchedules().join();

        VerticalLayout mainLayout = new VerticalLayout();
        mainLayout.setSizeFull();

mainLayout.setJustifyContentMode(FlexComponent.JustifyContentMode.ST
ART);
        mainLayout.setAlignItems(FlexComponent.Alignment.CENTER);

        add(createSidebarView(ScheduleView.class, UI.getCurrent(),
user));

        HorizontalLayout weekLayout = new HorizontalLayout();
        weekLayout.setWidth("80%");

weekLayout.setJustifyContentMode(FlexComponent.JustifyContentMode.CE
NTER);
```

```
        for (Weekday weekday : weekdays) {
            VerticalLayout weekdayColumn = new VerticalLayout();
            weekdayColumn.addClassName("weekday-column");
            weekdayColumn.setWidth("100%");

            // Создание Span с отступом сверху
            Span weekdayTitle = new Span(weekday.getWeekday());
            weekdayTitle.getStyle().set("font-weight", "bold");
            weekdayTitle.getStyle().set("text-decoration",
"underline");
            weekdayTitle.getStyle().set("margin-top", "10px"); //
Устанавливаем отступ сверху

            weekdayColumn.add(weekdayTitle);

            List<Schedule> schedules = weekday.getSchedules();
            for (Schedule schedule : schedules) {
                weekdayColumn.add(createScheduleCard(schedule));
            }

            weekLayout.add(weekdayColumn);
        }

        mainLayout.add(weekLayout);
        add(mainLayout);
        if (SecurityUtils.isAdminOrHigher()) {
            VerticalLayout vl = new VerticalLayout();
            vl.add(addScheduleButton());
            vl.add(addScheduleAddingButton());
            vl.add(addGroupAddingButton());
            add(vl);
        }
        this.coachService = coachService;
        this.groupService = groupService;
    }

    private VerticalLayout createScheduleCard(Schedule schedule) {
        VerticalLayout card = new VerticalLayout();
        card.getStyle().set("background-color", "#FFFFFF")
                .set("padding", "15px")
                .set("border-radius", "10px")
                .set("box-shadow", "0px 4px 8px rgba(0, 0, 0, 0.1)")
// Увеличиваем тень для более выраженного эффекта
                .set("margin", "10px 0") // Добавляем отступы между
карточками
                .set("color", "#333"); // Возможно, лучше выбрать
немного более светлый цвет текста для контраста с фоном
        card.setWidth("auto");
```

```
        card.setHeight("auto"); // Делаем высоту автоматической,
чтобы текст не обрезался
        card.setPadding(true); // Устанавливаем отступы внутри
карточки
        card.setAlignItems(FlexComponent.Alignment.CENTER); //
Центрируем элементы по вертикали
        SimpleDateFormat sdf = new SimpleDateFormat("HH:mm");

        Text group = new Text(schedule.getGroup().getName() + "  ");
        Text hour = new Text(sdf.format(schedule.getTime()) + "  ");
        Text court = new Text(schedule.getCourt().getCourtName() + "
");
        Coach coach = schedule.getGroup().getCoach();
        Text coachField = (coach == null) ? new Text("Нет") : new
Text(coach.getUser().getLast_name());
        card.add(group, hour, court, coachField);
        if (SecurityUtils.isAdminOrHigher()) {
            Button delete = getDeleteButton(schedule.getId());
            Button edit =
getEditCoachButton(schedule.getGroup().getId());
            card.add(delete, edit);
        }
```

3. Создание резервных копий базы данных.

Для создания резервной копии используется утилита pg_dump, для восстановления – pg_restore. Эта функция доступна только администраторам и владельцам.

Для реализации используется 2 сервиса – DatabaseAdminService, DatabaseBackupService. В первом производится подготовка временных файлов, после чего вызываются методы createBackup и restoreDatabase из DatabaseBackupService, куда и передаются подготовленные временные файлы.

Листинг #7. Методы createBackup и restoreDatabaseFromFile из DatabaseAdminService.

```
public String createBackup() throws IOException,
InterruptedException {
    // Определяем путь для временного файла
    String filePath = System.getProperty("java.io.tmpdir") +
File.separator + "backup.tar";
```

```
    // Вызываем метод для создания резервной копии
    databaseBackupService.createBackup(filePath, "sport_section",
"postgres");

    return filePath;
}


public void restoreDatabaseFromFile(InputStream inputStream) throws
IOException, InterruptedException {
    // Сохраняем переданный файл во временную директорию
    String tempFilePath = System.getProperty("java.io.tmpdir") +
"uploaded_backup.tar";
    Files.copy(inputStream, Paths.get(tempFilePath),
StandardCopyOption.REPLACE_EXISTING);

    // Восстанавливаем базу данных из резервной копии
    databaseBackupService.restoreDatabase(tempFilePath,
"sport_section", "postgres");

    // Удаляем временный файл
    Files.delete(Paths.get(tempFilePath));
}
```

Рассмотрим методы из DatabaseBackupService. Метод createBackup передает необходимые параметры в класс ProccessBuilder, включая путь к pg_dump, после чего получает результат выполнения в переменную exitCode, проверяя ее на равенство нулю и отлавливая возможные ошибки.

Листинг #8. Метод createBackup.

```
public void createBackup(String filePath, String dbName, String
username) throws IOException, InterruptedException {
    // Конфигурация ProcessBuilder
    ProcessBuilder pb = new ProcessBuilder(
            PG_DUMP_PATH,
            "-U", USERNAME_DB,
            "-F", "t", // Формат TAR
            "-f", filePath, // Указываем путь для сохранения
            DB_NAME // Имя базы данных
    );

    pb.environment().put("PGPASSWORD", PASSWORD);

    Process process = pb.start();

    int exitCode = process.waitFor();
```

```
    if (exitCode != 0) {
        String errorOutput = new
String(process.getErrorStream().readAllBytes()); // Ошибки
        String output = new
String(process.getInputStream().readAllBytes()); // Общий лог
        throw new IOException("Ошибка при создании резервной
копии!\n" + errorOutput + "\nЛог программы:\n" + output);
    }
}
```

Аналогично работает и restoreDatabase, только вместо пути к pg_dump, передается путь к pg_restore.

Для отображения функционала на frontend используется класс AdminDatabaseView и кнопка “Управление копиями бд” в левой панели.

4. Функции и процедуры

4.1 Функции

Функция в базе данных — это переиспользуемый блок кода, который выполняет определённую задачу. Она может принимать входные параметры, выполнять операции (например, вычисления, модификацию данных) и возвращать результат. В данном приложении реализовано несколько функций.

Большая часть из них нужна для реализации выборки из базы данных, например, чтобы отобразить все бронирования пользователя используется функция GET_BOOKING_COURTS_BY_USER_ID.

Листинг #9. Функция базы данных для поиска всех бронирований пользователя.

```
CREATE OR REPLACE FUNCTION GET_BOOKING_COURTS_BY_USER_ID(
    user_id_ INT
) RETURNS TABLE (bk booking_courts) AS $$
BEGIN
    RETURN QUERY
        SELECT id, court_id, user_id, date, booking_time
        FROM booking_courts
        WHERE user_id = user_id_;
END;
$$ LANGUAGE plpgsql;
```

Аналогично реализованы и другие функции для выборки данных.

Триггерные функции — это специальные функции, которые выполняются автоматически при срабатывании триггера. Они используются для выполнения действий до или после конкретного события в таблице или представлении.

В данном приложении задействована одна триггерная функция – проверка времени бронирования при добавлении новой записи аренды. Хоть frontend и не предоставляет возможности выбора времени ранее 7-00 и позднее 22-00, во избежание ошибок была реализована следующая триггерная функция:

Листинг #10. Триггерная функция для контроля времени бронирования

```
CREATE OR REPLACE FUNCTION public.check_booking_time()
RETURNS trigger
LANGUAGE plpgsql
AS $function$
BEGIN
    IF NEW.booking_time < '07:00:00'::time OR NEW.booking_time >
'22:00:00'::time THEN
        RAISE EXCEPTION 'Время должно быть между 07:00 и 22:00';
    END IF;
    RETURN NEW;
END;
$function$;
```

4.2 Процедуры

Процедуры - программные блоки, которые хранятся и выполняются на уровне базы данных. Они позволяют выполнять заранее определённые последовательности операций, подобных функциям, но с возможностью иметь более сложный синтаксис, транзакционное управление и выполнение без возвращения значений.

В данном приложении процедуры реализованы для модификаций базы данных – добавления, удаления и изменения объектов.

Листинг #11. Пример процедуры для изменения тренера в группе

```
CREATE OR REPLACE PROCEDURE update_coach_in_group(group_id_ int,
coach_id_ int)
    LANGUAGE plpgsql
AS $$
BEGIN
```

```
    UPDATE groups SET coach_id = coach_id_ WHERE id = group_id_;
END;
$$;
```

5. Вывод

Таким образом, создано веб-приложение на основе Java, предназначенное для управления данными спортивных секций. Приложение построено с использованием ORM Hibernate, базы данных PostgreSql и многоуровневой архитектуры, включающей уровни DAO (Data Access Object), сервисный слой и слой представления. Данное приложение предоставляет возможности для регистрации клиентов, управления расписанием тренировок, аренды кортов, а также управления данными о тренерах и секциях.

6. Источники

1. Фримен Э., Робсон Э. - Изучаем Java. (15.02.2023)
2. Бауэр Г., Кинг К. - Hibernate. Официальная документация. (20.03.2023)
3. Электронный ресурс https://hiberbook.com/en/ (5.04.2023)
4. Электронный ресурс https://spring.io/guides/gs/accessing-data-postgres/ (10.05.2023)
5. Электронный ресурс https://habr.com/ru/articles/654321/ (22.05.2023)
6. Электронный ресурс https://thorben-janssen.com/jpa-hibernate-tutorial/ (29.05.2023)
7. Электронный ресурс https://www.postgresql.org/docs/9.6/index.html (1.06.2023)
8. Электронный ресурс https://dzone.com/articles/quick-guide-to-hibernate-with-postgresql-in-spring (2.06.2023)

7. Приложение

Весь код находится по ссылке https://github.com/SChernysheva/course_project

Sql скрипт для базы данных находится по ссылке course_work_db/sql_script.sql at master · SChernysheva/course_work_db