ОТЧЕТ  
ПО РАСЧЕТНО-ГРАФИЧЕСКОЙ РАБОТЕ  
по дисциплине «Разработка приложений баз данных»  
«Использование технологии REST API для создания приложений баз данных»  
**Вариант 15**

Факультет: АВТ Преподаватель: Менжулин С. А.

Группа: АВТ-943

Студент: Фокин Е.К.

Цель работы:

Получение практических навыков использования технологии REST API для создания приложений баз данных.

Задание:

1. Для разработанной объектной модели из лабораторной работы №2 классов-сущностей реализовать сохранение данных объектов в реляционной базе данных посредством разработанного протокола REST API, для чего выполнить разделение на клиентскую и серверную части приложения.
2. Функции клиентской части приложения: интерфейс пользователя, REST API клиент, объектная модель предметной области.
3. Функции серверной части приложения: REST API сервер, объектная модель предметной области, взаимодействие с СУБД (допускается использование любой ORM/ODBC/JDBC/libpq).
4. Для обмена данными между клиентской и серверной частью приложения использовать JSON. Допускается использование gRPC только в комплекте с gRPC-Gateway.
5. Использовать различные языки программирования в клиентской и серверной части приложения. Исключить из выбора пары Java + Kotlin, C + C++, JS framework + JS framework.
6. Для создания интерактивной документации использовать Swagger UI на стороне серверной части приложения.

Реализовать интерфейс для демонстрации работы любым удобным способом.Задание варианта:

15) БД «Гостиница»

База данных должна содержать сведения о следующих объектах:

База данных должна содержать сведения о следующих объектах:

Распределение номеров с указанием общего количества мест в номере, количества свободных мест и проживающих.

Гости - фамилия, имя, отчество, пол, адрес, дата рождения, номер паспорта, дата выдачи, учреждение, выдавшее паспорт, номер комнаты, номер стоянки, регистрационный номер автомобиля, дата въезда, дата выезда, список оказанных услуг (наименование услуги, количество, цена).

Адресные данные горничных и расписание их дежурств.

Ход работы:

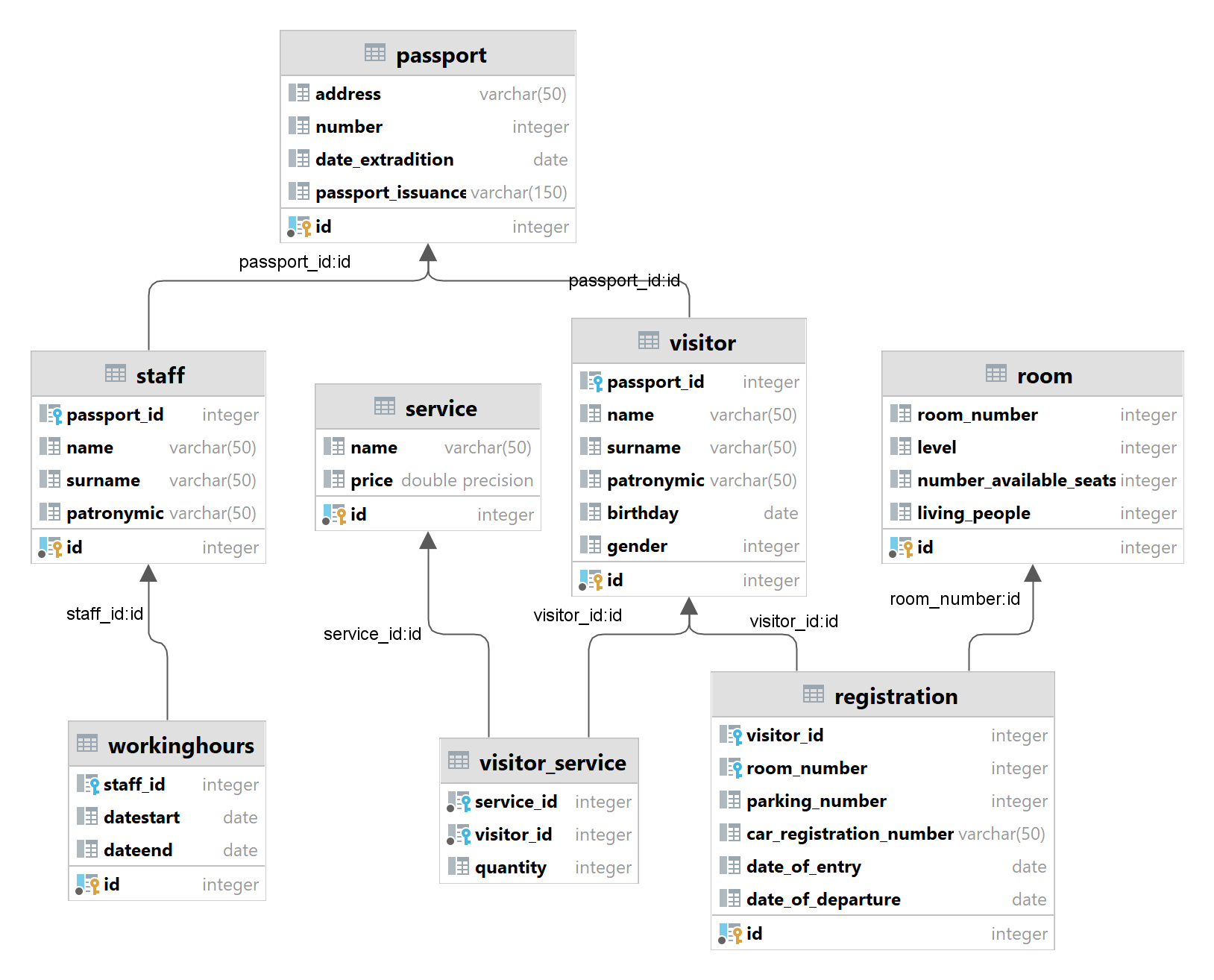


Рис. 1 Схема базы данных

Для начала, разберемся с бэкендом:

Было принято решение расширить 2 лабораторную работу с использованием Hibernate. Перенесем классы мапинга на рельсы SpringBootApplication.

Пример маппинга:

@Entity  
@Table(name = "registration")  
@Data  
public class Registration{  
  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 @Id  
 @Column(name = "id")  
 private Long id;  
  
 @Column(name = "parking\_number")  
 private Integer parkingNumber;  
 @Column(name = "car\_registration\_number")  
 private String carRegistrationNumber;  
 @Column(name = "date\_of\_entry")  
 private Date dateOfEntry;  
 @Column(name = "date\_of\_departure")  
 private Date dateOfDeparture;  
  
 @ManyToOne  
 @JoinColumn(name = "visitor\_id", referencedColumnName = "id", nullable = false)  
 private Visitor visitorByVisitor;  
  
 @ManyToOne  
 @JoinColumn(name = "room\_number", referencedColumnName = "id", nullable = false)  
 private Room roomByRoom;  
}

Далее создадим абстрактный класс для основных операций REST API сервиса:

public abstract class BasePersistentService<T> {  
  
 protected abstract <Repo extends JpaRepository<T, Long>> Repo getRepo();  
  
 public T save(T entity){  
 return getRepo().save(entity);  
 }  
  
 public List<T> findAll(){  
 return getRepo().findAll();  
 }  
  
 public void delete(Long id){  
 getRepo().deleteById(id);  
 }  
}

Переопределим для всех таблиц методы edit():

@Service  
@RequiredArgsConstructor  
public class RegistrationService extends BasePersistentService<Registration> {  
  
 @Getter(AccessLevel.*PROTECTED*)  
 private final RegistrationRepository repo;  
  
 public Registration edit(Long id, Registration registration){  
 if (id == null){  
 return null;  
 }  
 Registration registrationFromDb = repo.findById(id).orElse(null);  
 if (registrationFromDb == null) return null;  
 BeanUtils.*copyProperties*(registration, registrationFromDb, "id");  
 return repo.save(registrationFromDb);  
 }  
}

*Spring Boot* обладает большим функционалом, но его наиболее значимыми особенностями являются: управление зависимостями, автоматическая конфигурация и встроенные контейнеры сервлетов.

Для каждого класса нашей таблицы создадим контроллеры для осуществления основных операций:

@RestController  
@RequestMapping("/registration")  
@RequiredArgsConstructor  
public class RegistrationController {  
  
 private final RegistrationService registrationService;  
  
 @ApiOperation("Receive all registration")  
 @GetMapping  
 public List<Registration> getAllpassport() {  
 return registrationService.findAll();  
 }  
  
 @ApiOperation("Create a registration")  
 @PostMapping  
 public Registration create(@RequestBody Registration registration){  
 return registrationService.save(registration);  
 }  
  
 @ApiOperation("Edit a registration")  
 @PutMapping("/{id}")  
 public Registration edit(@PathVariable Long id, @RequestBody Registration registration){  
 return registrationService.edit(id, registration);  
 }  
  
 @ApiOperation("Delete a registration")  
 @DeleteMapping("/{id}")  
 public void delete(@PathVariable Long id){  
 registrationService.delete(id);  
 }  
}

Проверим работу сервера, который запускается на <http://localhost:8080/>

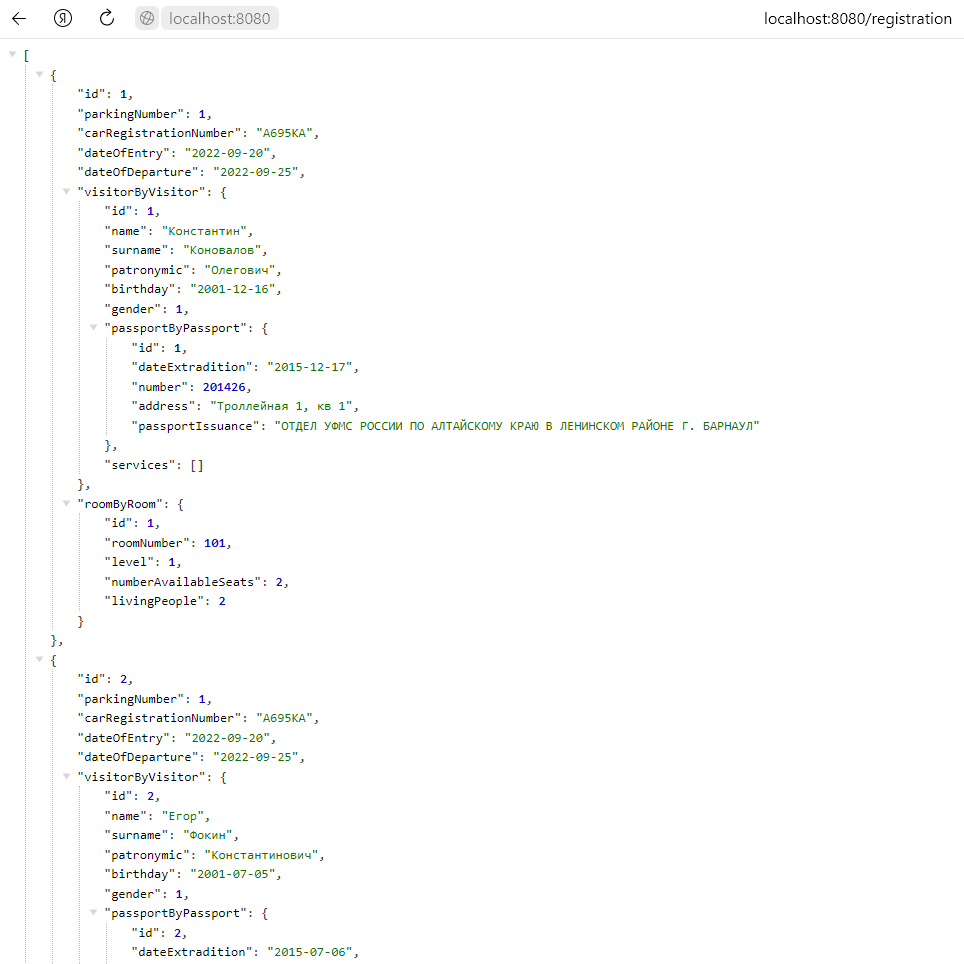


Рис. 2 Пример вывода в JSON

Данные сериализуются в формат JSON.

Swagger2 – это проект с открытым исходным кодом, используемый для генерации документов API REST для веб-сервисов RESTful. Он предоставляет пользовательский интерфейс для доступа к нашим веб-службам RESTful через веб-браузер.

Чтобы включить Swagger2 в приложении Spring Boot, вам необходимо добавить следующие зависимости в наш файл конфигурации сборки.

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.7.0</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.7.0</version>

</dependency>

Пример создания Swagger:

@Configuration  
@EnableSwagger2  
public class SwaggerConfiguration {  
  
 @Bean  
 public Docket api() {  
 return new Docket(DocumentationType.*SWAGGER\_2*)  
 .select()  
 .apis(RequestHandlerSelectors.*any*())  
 .paths(PathSelectors.*any*())  
 .build();  
 }  
}

Для проверки посмотрим по данному адресу: <http://localhost:8080/v2/api-docs>

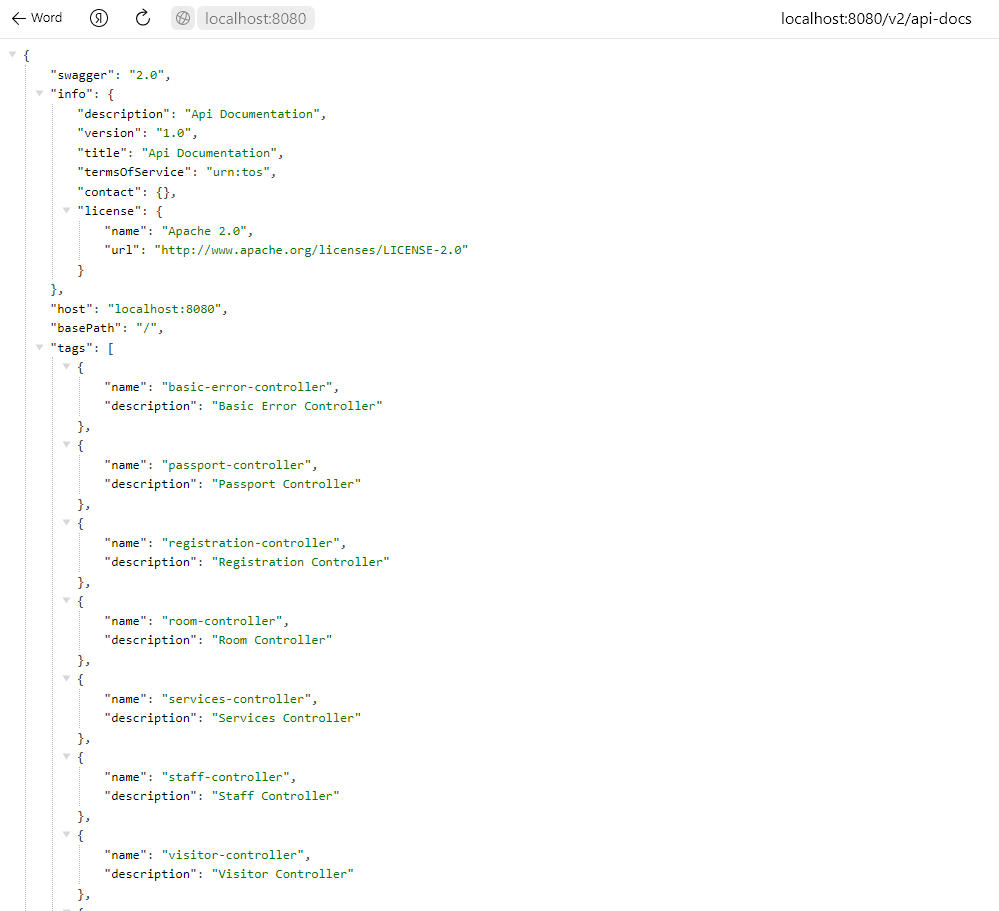


Рис. 3 Создание документации

Перейдем к фронту.

Разрабатывать клиентскую часть будем на TypeScript используя React.

Для того, чтобы наше приложение могло обращаться по API к серверу и для того, чтобы не писать каждый запрос вручную, используем приложение Swagger Codegen CLI, которое сгенерирует нужные классы для обращений к серверу, а также классы моделей предметной области.

Swagger Codegen - это генератор кода с открытым исходным кодом для создания серверных заглушек и клиентских SDK непосредственно из определяемого Swagger RESTful API. Исходный код для Swagger Codegen можно найти на GitHub: <https://github.com/swagger-api/swagger-codegen>

В настоящее время поддерживаются следующие языки/фреймворки:

* API клиенты: ActionScript, Ada, Apex, Bash, C# (.net 2.0, 3.5 or later), C++ (cpprest, Qt5, Tizen), Clojure, Dart, Elixir, Elm, Eiffel, Erlang, Go, Groovy, Haskell (http-client, Servant), Java (Jersey1.x, Jersey2.x, OkHttp, Retrofit1.x, Retrofit2.x, Feign, RestTemplate, RESTEasy, Vertx, Google API Client Library for Java, Rest-assured), Kotlin, Lua, Node.js (ES5, ES6, AngularJS with Google Closure Compiler annotations) Objective-C, Perl, PHP, PowerShell, Python, R, Ruby, Rust (rust, rust-server), Scala (akka, http4s, swagger-async-httpclient), Swift (2.x, 3.x, 4.x, 5.x), Typescript (Angular1.x, Angular2.x, Fetch, jQuery, Node);
* Заглушки сервера: Ada, C# (ASP.NET Core, NancyFx), C++ (Pistache, Restbed), Erlang, Go, Haskell (Servant), Java (MSF4J, Spring, Undertow, JAX-RS: CDI, CXF, Inflector, RestEasy, Play Framework, PKMST), Kotlin, PHP (Lumen, Slim, Silex, Symfony, Zend Expressive), Python (Flask), NodeJS, Ruby (Sinatra, Rails5), Rust (rust-server), Scala (Finch, Lagom, Scalatra);
* Генераторы документации API: HTML, Confluence Wiki;
* Файлы конфигурации: Apache2;
* Другие: JMeter.

Генерацию можно сделать командой:

java -jar swagger-codegen-cli.jar generate -i http://localhost:8080/v2/api-docs -l typescript-fetch -o api.tmp -D supportsES6=true

Опишем для каждой таблички основные методы взаимодействия с REST API:

export interface BaseApi<T> {  
 create(entity: T): Promise<T>;  
 edit(id: number, entity: T): Promise<T>  
 findAll(): Promise<T[]>  
 delete(id: number): Promise<Response>  
}

Далее, на подобии 3 лабораторной работы, создадим классы обертки:

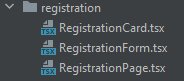


Рис. 4 Классы обертки

Получаем итоговый результат:

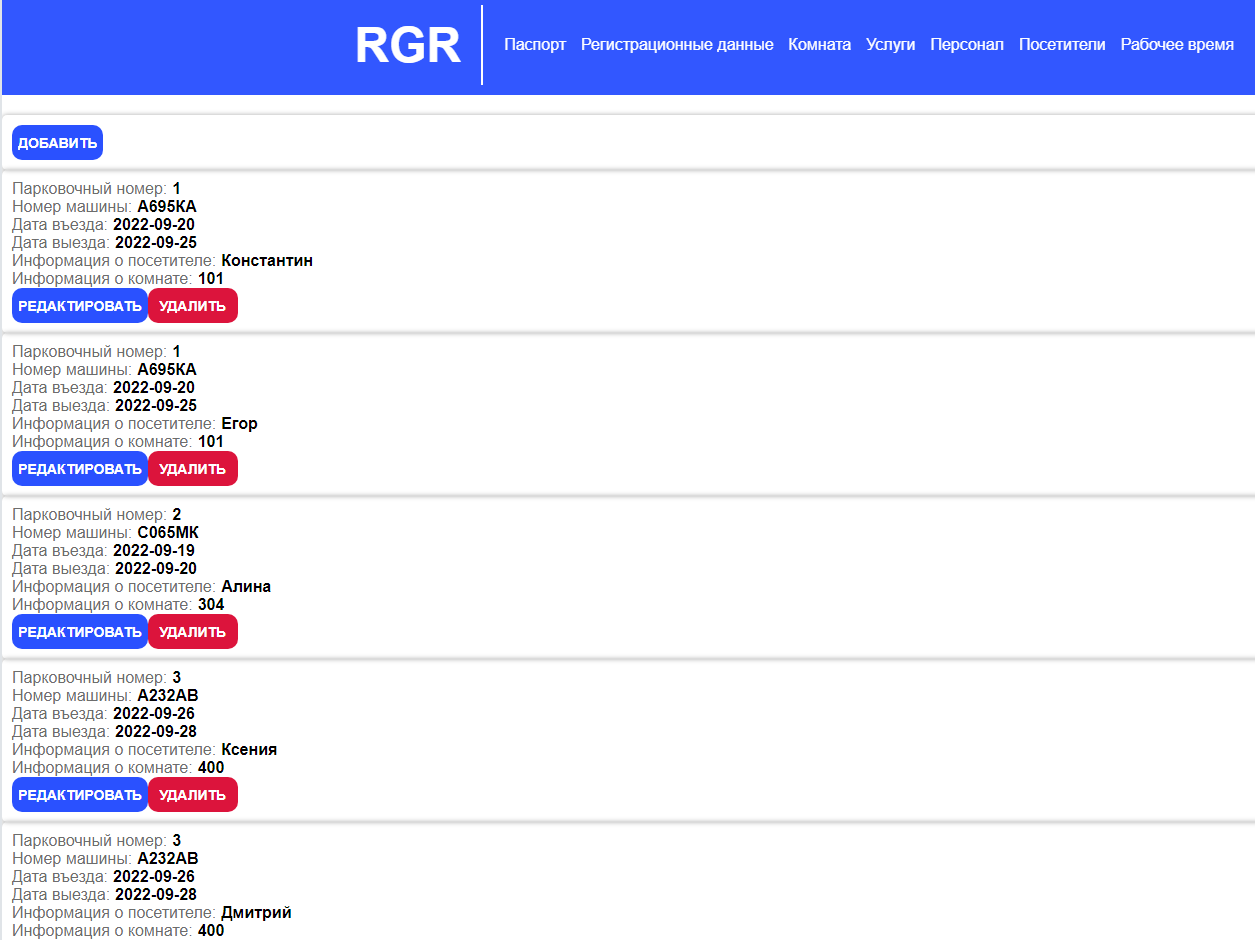


Рис. 5 Интерфейс

Вывод:

В ходе данной расчетно-графической работы было разработано клиентское приложение, использующее REST API для взаимодействия с сервером и управления сущностями в базе данных. Также было использовано приложение Swagger codegen, для генерации TypeScript классов для удобного взаимодействия с сервером.

Приложение:

Файл с расчетно-графической задачей прикреплен в файл Dispace

Основные моменты были приведены по ходу отчета, далее представим коды оберток:

***RegistrationCard.tsx:***

import {Registration} from "../../api/base";  
import {RegistrationForm} from "./RegistrationForm";  
import {useState} from "react";  
import {Property} from "../Property";  
  
interface Props {  
 registration: Registration  
 onEdit: (id?: number, visitor?: Registration) => void  
 onDelete: (id?: number) => void  
}  
  
export const RegistrationCard: ***React***.FC<Props> = ({ registration, onEdit, onDelete }) => {  
 const [isEdit, setIsEdit] = useState(false)  
  
  
 return (  
 <div className="card registration-card">  
 {isEdit ?  
 <RegistrationForm registration={registration} onSubmit={(newClient) => { onEdit(registration.id, newClient); setIsEdit(false) }} />  
 :  
 <div className="visitor-card\_\_main">  
  
 <Property title="Парковочный номер:" value={registration.parkingNumber} />  
 <Property title="Номер машины:" value={registration.carRegistrationNumber} />  
 <Property title="Дата въезда:" value={registration.dateOfEntry} />  
 <Property title="Дата выезда:" value={registration.dateOfDeparture} />  
  
 <Property title="Информация о посетителе:" value={registration.visitorByVisitor?.name} />  
 <Property title="Информация о комнате:" value={registration.roomByRoom?.roomNumber} />  
 </div>  
 }  
 <div className="visitor-card\_\_controls">  
 <button className="button"  
 onClick={() => setIsEdit(!isEdit)}>{isEdit ? 'Закрыть' : 'Редактировать'}</button>  
 <button className="button button\_red" onClick={() => onDelete(registration.id)}>Удалить</button>  
 </div>  
 </div>  
 )  
}

***RegistrationForm.tsx:***

import {Registration} from "../../api/base";  
import {useState} from "react";  
import {***registrationApi***} from "../../api/RegistrationApi";  
import {useEntities} from "../../util/useEntities";  
import {Property} from "../Property";  
import {***visitorApi***} from "../../api/VisitorApi";  
import {***roomApi***} from "../../api/RoomApi";  
  
interface Props {  
 registration?: Registration  
 onSubmit: (registration: Registration) => void  
}  
  
export const RegistrationForm: ***React***.FC<Props> = ({ registration, onSubmit }) => {  
  
 const [visitors] = useEntities(***visitorApi***)  
 const [rooms] = useEntities(***roomApi***)  
  
 const [parkingNumber, setParkingNumber] = useState(Number(registration?.parkingNumber ?? ''))  
 const [carRegistrationNumber, setCarRegistrationNumber] = useState(registration?.carRegistrationNumber ?? '')  
 const [dateOfEntry, setDateOfEntry] = useState(registration?.dateOfEntry ?? '')  
 const [dateOfDeparture, setDateOfDeparture] = useState(registration?.dateOfDeparture ?? '')  
  
 const [selectVisitor, setSelectVisitor] = useState<string[]>(visitors?.map(o => o.id?.toString() ?? '') ?? [])  
 const [selectRoom, setSelectRoom] = useState<string[]>(rooms?.map(o => o.id?.toString() ?? '') ?? [])  
  
  
 const onClick = () => {  
 if (parkingNumber === 0) return  
 onSubmit({  
 parkingNumber,  
 carRegistrationNumber,  
 dateOfEntry,  
 dateOfDeparture,  
  
 visitorByVisitor: visitors?.find(st => !!selectVisitor.find(ts => ts === st.id?.toString() ?? '-1')),  
 roomByRoom: rooms?.find(st => !!selectRoom.find(ts => ts === st.id?.toString() ?? '-1'))  
 })  
 setParkingNumber(0)  
 setCarRegistrationNumber('')  
 setDateOfEntry('')  
 setDateOfDeparture('')  
  
 setSelectVisitor([])  
 setSelectRoom([])  
 }  
  
 return (  
 <div className="registration-form">  
 <Property title="Парковочный номер:" value={<input type="text" value={parkingNumber} onChange={e => setParkingNumber(Number(e.target.value))} />} />  
 <Property title="Номер машины:" value={<input type="text" value={carRegistrationNumber} onChange={e => setCarRegistrationNumber(e.target.value)} />} />  
 <Property title="Дата въезда:" value={<input type="text" value={dateOfEntry} onChange={e => setDateOfEntry(e.target.value)} />} />  
 <Property title="Дата выезда:" value={<input type="text" value={dateOfDeparture} onChange={e => setDateOfDeparture(e.target.value)} />} />  
  
 <Property title="Информация о посетителе:">  
 <select multiple value={selectVisitor}  
 onChange={e => setSelectVisitor(***Array***.from(e.target.selectedOptions, option => option.value))}>  
 {visitors?.map(m => <option key={m.id}  
 value={m.id}>{m.name}</option>)}  
 </select>  
 </Property>  
  
 <Property title="Информация о комнате:">  
 <select multiple value={selectRoom}  
 onChange={e => setSelectRoom(***Array***.from(e.target.selectedOptions, option => option.value))}>  
 {rooms?.map(m => <option key={m.id}  
 value={m.id}>{m.roomNumber}</option>)}  
 </select>  
  
 </Property>  
 <button className="button button\_green" onClick={onClick}>Ок</button>  
 </div>  
 )  
}

***StaffPage.tsx:***

import {***registrationApi***} from "../../api/RegistrationApi";  
import {Registration} from "../../api/base";  
import {useEntities} from "../../util/useEntities";  
import {RegistrationForm} from "./RegistrationForm";  
import {RegistrationCard} from "./RegistrationCard";  
import {useState} from "react";  
  
  
export const RegistrationPage: ***React***.FC = () => {  
 const [registrations, \_, refresh] = useEntities(***registrationApi***)  
 const [addFormShow, setAddFormShow] = useState(false)  
  
 const onAddSubmit = (client: Registration) => {  
 ***registrationApi***.create(client).finally(refresh)  
 setAddFormShow(false)  
 }  
  
 const onEdit = (id?: number, client?: Registration) => {  
 if (!id || !client) return  
 ***registrationApi***.edit(id, client).finally(refresh)  
 }  
  
 const onDelete = (id?: number) => {  
 if (!id) return  
 ***registrationApi***.delete(id).finally(refresh)  
 }  
  
 return (  
 <div className="registration-page">  
 <div className="card">  
 <button className="button" onClick={() => setAddFormShow(!addFormShow)}>{`${addFormShow ? 'Закрыть' : 'Добавить'}`}</button>  
 {addFormShow &&  
 <RegistrationForm onSubmit={onAddSubmit} />  
 }  
 </div>  
 <div>  
 {registrations?.map(c => <RegistrationCard key={c.id} registration={c} onEdit={onEdit} onDelete={onDelete} />)}  
 </div>  
 </div>  
 )  
}