Федеральное государственное образовательное бюджетное   
учреждение высшего образования

«Финансовый университет при Правительстве Российской Федерации»

Факультет информационных технологий и анализа больших данных

Департамент анализа данных и машинного обучения

**Направление подготовки:** «Прикладная информатика»

**Профиль:** «ИТ-сервисы и технологии обработки данных в экономике и финансах»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

**по дисциплине** «Современные технологии программирования»

**на тему:** «Приложение “Платная поликлиника”»

Выполнил студент 2-го курса,

группы ПИ21-2,

очной формы обучения,

Преснухин Дмитрий Михайлович

Проверяет преподаватель:

К.т.н., доцент, Гущин Сергей Иванович

# СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc133795872)

[Глава 1. Выбор стека технологий 5](#_Toc133795873)

[Глава 2. База данных 6](#_Toc133795874)

[2.1. Создание 6](#_Toc133795875)

[2.2. Заполнение 10](#_Toc133795876)

[Глава 3. Серверная часть 13](#_Toc133795877)

[3.1. Взаимодействие с БД 13](#_Toc133795878)

[3.2. Взаимодействие с клиентом 16](#_Toc133795879)

[3.3. Защита от несанкционированного доступа 20](#_Toc133795880)

[3.1.1. Реализация пользователей 21](#_Toc133795881)

[3.3.2. Аутентификация 24](#_Toc133795882)

[3.3.3. Работа с токенами 24](#_Toc133795883)

[3.3.4. Конфигурация 25](#_Toc133795884)

[3.4. Резюме 27](#_Toc133795885)

[Глава 4. Клиентская часть 29](#_Toc133795886)

[4.1. Структура 29](#_Toc133795887)

[4.2. Компоненты 30](#_Toc133795888)

[4.3. Аутентификация и взаимодействие с сервером 32](#_Toc133795889)

[4.4. Маршрутизация 35](#_Toc133795890)

[Глава 5. Возможности приложения 38](#_Toc133795891)

[5.1. Анонимный пользователь 38](#_Toc133795892)

[5.2. Пациент 38](#_Toc133795893)

[5.3. Администратор 38](#_Toc133795894)

[ЗАКЛЮЧЕНИЕ 40](#_Toc133795895)

[ИСТОЧНИКИ 43](#_Toc133795896)

[ПРИЛОЖЕНИЕ 44](#_Toc133795897)

[Руководство пользователя для пациента 44](#_Toc133795898)

[Руководство для администратора 52](#_Toc133795899)

# ВВЕДЕНИЕ

В настоящее время медицинская индустрия переживает значительный рост спроса на медицинские услуги, однако в некоторых регионах существуют проблемы с доступностью медицинской помощи для населения. Это становится особенно заметно в случае, если необходима своевременная консультация или обследование. Кроме того, часто в государственных медицинских учреждениях происходят задержки в обслуживании пациентов из-за большого количества посетителей, что может привести к ухудшению их здоровья.

Для решения данной проблемы было разработано веб-приложение "Платная поликлиника", реализующее клиент-серверную архитектуру, позволяющую позволяет пациентам записываться на прием к врачу в удобное для них время, не стоя в очереди, и получать быстрый доступ к медицинским услугам.

Для достижения данной цели были поставлены следующие задачи:

* Разработать клиент-серверную архитектуру приложения
* Создать сервер на Java Spring c использованием REST-контроллеров
* Разработать базу данных на PostgreSQL и обеспечить взаимодействие сервера с БД с помощью Spring, JPA и Hibernate
* Реализовать клиентскую часть приложения с использованием современных технологий. Сделать её максимально дружелюбной для пользователя и одновременно с этим функциональной
* Обеспечить взаимодействие между сервером и клиентской частью приложения

# Глава 1. Выбор стека технологий

Для разработки приложения "Платная поликлиника" был выбран стек технологий, который позволил обеспечить высокую производительность, безопасность и удобство использования приложения для пользователей.

Для серверной части приложения был выбран Java Spring, который является одним из самых популярных и широко используемых фреймворков для разработки серверных приложений на Java. Java Spring предоставляет широкий набор инструментов для реализации RESTful API, что позволяет создать легко масштабируемое и гибкое API для взаимодействия с клиентской частью.

Для реализации клиентской части приложения был выбран React, который является одним из наиболее популярных фреймворков для создания пользовательских интерфейсов. React обладает высокой производительностью и возможностью создавать компоненты для повторного использования, что ускоряет процесс разработки и упрощает поддержку приложения в дальнейшем.

Для взаимодействия с базой данных был выбран PostgreSQL, который является мощной и надежной системой управления реляционными базами данных. PostgreSQL позволяет обеспечить высокую производительность и защиту данных, а также предоставляет возможность горизонтального масштабирования для обработки большого количества запросов.

Выбранный стек технологий позволил создать удобное, быстрое и безопасное приложение "Платная поликлиника", которое обеспечивает быстрый доступ к медицинским услугам для пациентов и удобный способ работы для администраторов.

# Глава 2. База данных

## 2.1. Создание

В первую очередь необходимо разработать базу данных для приложения. После тщательного анализа и взвешивания трудности решения и его удобства, было решено использовать следующие сущности:

1. **Пациент**

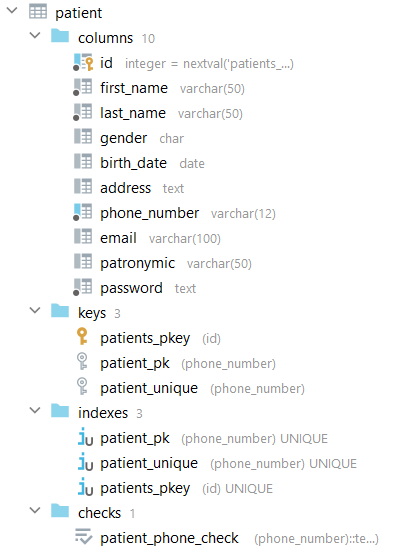


Рисунок 1

Номер телефона пациента может быть только в международном формате и начинаться только с +7. Такое решение было выбрано в целях упрощения реализации. Валидность номера телефона проверяется в самой базе при помощи patient\_phone\_check: (phone\_number)::text ~ similar\_to\_escape('\+7\d{10}'::text)

Стоит отметить, что в дополнение к вышеприведенным атрибутам пациента можно было добавить еще несколько, например номер СНИЛС или паспортные данные, но данное отношение является лишь демонстрацией и было решено не вдаваться в мелкие подробности.

Для номера телефона стоит ограничение уникальности – в базу нельзя занести несколько пациентов с одним номером.

Уникальный идентификатор пациента автоматически генерируется на основе последовательности , начиная с 1 и с интервалом 1.



1. **Врач**

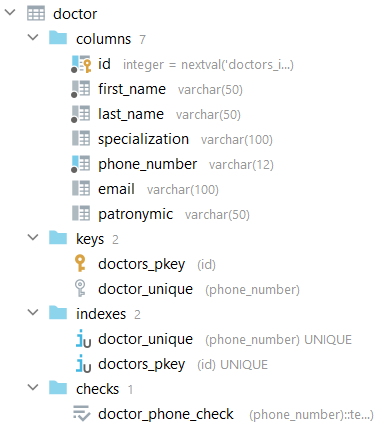


Рисунок 2

Отношение «doctor» обладает похожими свойствами, номер так же уникален и проверяется на валидность, id генерируется автоматически.

1. **Услуга**

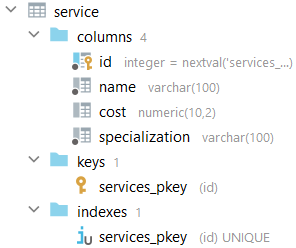


Рисунок 3

Отношение «service» хранит в себе записи о возможных платных услугах, оказываемых в клинике.

Идентификатор генерируется автоматически.

1. **Прием у врача**

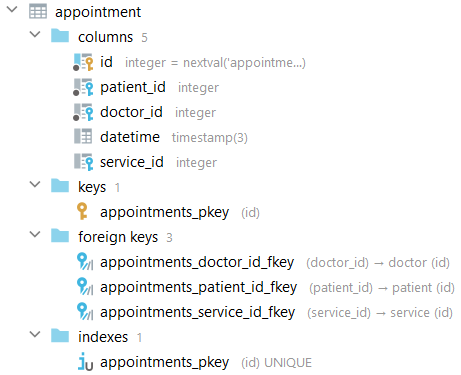


Рисунок 4

Отношение «appointment» в качестве внешних ключей использует уникальные идентификаторы таблиц «patient», «doctor» и «service». Таким образом можно связать информацию о пациенте, принимающем враче и оказанной услуге.

1. **Администратор**

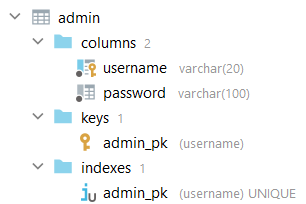


Рисунок 5

Отношение «admin» хранит логины и пароли всех администраторов, имеющих неограниченный доступ к REST-контроллерам (подробнее в разделе 3.3).

Итоговая модель базы данных выглядит так:

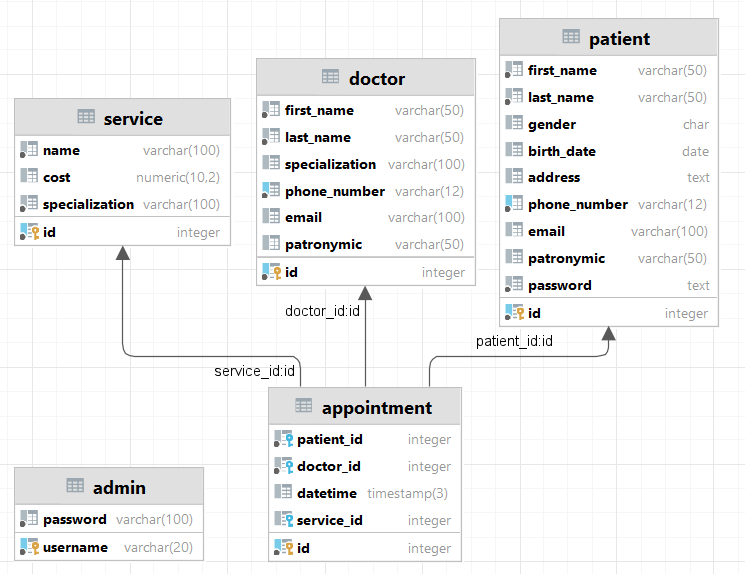


Рисунок 6

## 2.2. Заполнение

Для заполнения базы данных был написан скрипт на Python, генерирующий заданное количество случайных записей для всех таблиц, кроме «admin».

Код скрипта (в сокращении):

**import** **random**

**import** **string**

**import** **datetime**

**import** **requests**

**import** **json**

**import** **bcrypt**

# Списки русских имен, фамилий и отчеств

names\_male = ['Авдей', 'Авксентий'…]

names\_female = ['Агафья', 'Аглая'…]

surnames\_male = ['Абрамов'…]

surnames\_female = ['Абрамова'…]

patronymics\_male = ['Александрович'…]

patronymics\_female = ['Александровна'…]

specializations = {

"Кардиология": ['ЭКГ', 'Эхокардиография'…],

"Неврология": ['ЭЭГ', 'МРТ головного мозга'…]

…

}

# Дальше идут классы объектов сущностей

# Функция-генератор номеров телефонов

phone\_list = []

**def** **phone\_gen**():

**while** **True**:

phone\_number = "+79"

**for** i **in** range(**9**):

phone\_number += str(random.randint(**0**, **9**))

**if** phone\_number **not** **in** phone\_list:

phone\_list.append(phone\_number)

**return** phone\_number

# Функция-генератор почтовых адресов

email\_list = []

**def** **email\_gen**():

**while** **True**:

username = ''.join(random.choice(string.ascii\_lowercase) **for** \_ **in** range(random.randint(**5**, **10**)))

domain = random.choice(['gmail.com', 'ya.ru', 'mail.ru'])

email = f"{username}@{domain}"

**if** email **not** **in** email\_list:

email\_list.append(email)

**return** email

# Функция-генератор паролей, для теста пароль сделан одинковый для всех - "qwertyuiop"

# Пароль шифруется при помощи BCrypt

**def** **password\_gen**():

password = bcrypt.hashpw(b"qwertyuiop", bcrypt.gensalt())

**return** password

# Функция-генератор даты рождения

**def** **birthDate\_gen**():

start\_date = datetime.date(**1940**, **1**, **1**)

end\_date = datetime.date.today()

time\_between\_dates = end\_date - start\_date

days\_between\_dates = time\_between\_dates.days

random\_number\_of\_days = random.randrange(days\_between\_dates)

random\_date = start\_date + datetime.timedelta(days=random\_number\_of\_days)

**return** random\_date

# Функция-генератор даты-времени для приема у врача

**def** **datetime\_gen**():

start\_date = datetime.date(**2020**, **1**, **1**)

end\_date = datetime.date.today()

time = datetime.time(random.randint(**0**, **23**), random.randint(**0**, **59**), **0**)

random\_date = datetime.date.fromordinal(random.randint(start\_date.toordinal(), end\_date.toordinal()))

**return** datetime.datetime.combine(random\_date, time)

# Функция-генератор адресов Москвы

moscow\_streets = ['Арбат', 'Тверская'…]

**def** **address\_gen**():

**return** "Москва, " + random.choice(moscow\_streets) + ", д. " + str(random.randint(**1**, **20**)) + ", кв. " + str(random.randint(**1**, **50**))

# Далее происходит сама генерация

# Конечный SQL-запрос

**SQL** = f"""

DELETE FROM appointment;

DELETE FROM service;

DELETE FROM doctor;

DELETE FROM patient;

SELECT SETVAL('doctors\_id\_seq', 0);

SELECT SETVAL('patients\_id\_seq', 0);

SELECT SETVAL('appointments\_id\_seq', 0);

SELECT SETVAL('services\_id\_seq', 0);

INSERT INTO

patient (first\_name, last\_name, gender, birth\_date, address, phone\_number, email, patronymic, password)

VALUES {patient\_values[:-2]};

INSERT INTO

doctor (first\_name, last\_name, specialization, phone\_number, email, patronymic, password)

VALUES {doctor\_values[:-2]};

INSERT INTO service (name, cost, specialization)

VALUES {service\_values[:-2]};

INSERT INTO appointment (patient\_id, doctor\_id, datetime, service\_id)

VALUES {appointment\_values[:-2]};

"""

# После чего запрос записывается в файл

f = open("data.sql", mode="w", encoding="utf-8")

f.write(SQL)

f.close()

Таким образом в базу данных была внесена информация обо всех услугах, 50 врачах, 300 пациентов и о 3000 приемов у доктора (по 10 на одного пациента). *Любые совпадения с реальностью случайны*.

В таблицу «admin» добавлена лишь одна запись:



Рисунок 7

Пароль и логин администратора – admin.

# Глава 3. Серверная часть

## 3.1. Взаимодействие с БД

Так как для достижения цели курсовой работы предстояло организовать взаимодействие сервера с базой данных, было решено использовать популярный Java фреймворк Spring, позволяющий легко создавать приложения, взаимодействующие с базами данных. Spring предоставляет набор инструментов, таких как Spring Data JPA и Hibernate, которые значительно упрощают работу с базами данных и позволяют сократить время разработки.

Spring Data JPA — это библиотека, которая облегчает работу с БД и позволяет работать с данными, используя объектно-реляционную модель (ORM). Она предоставляет репозитории для каждой сущности и автоматически создает SQL-запросы на основе методов, которые мы определяем в репозитории. Это значительно упрощает работу с базами данных и уменьшает количество кода, который нужно написать.

Hibernate, в свою очередь, является ORM-фреймворком для работы с базами данных, который обеспечивает высокую производительность, надежность и безопасность при работе с данными. Hibernate позволяет работать с данными в виде объектов, что значительно упрощает написание кода и позволяет сократить время разработки.

Вот так выглядит структура классов, необходимых для работы ORM. Было решено отказаться от использования сервисов в дополнение к репозиториям, так как реализация бизнес-логики легко может быть описана далее в REST-контроллерах.

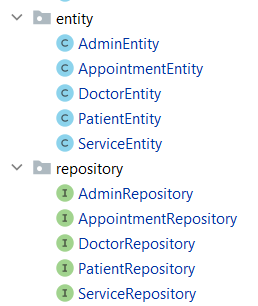


Рисунок 8

Классы сущностей базы данных были сгенерированы автоматически при помощи функции Intellij IDEA “Generate persistence mapping”. Вот фрагмент класса одной из сущностей:

*/\*\*  
 \* Класс, представляющий объект-сущность администратора в системе.  
 \* Наследуется от класса User.  
 \*/*@Entity  
@Table(name = **"admin"**, schema = **"public"**, catalog = **"PaidClinic"**)  
**public class** AdminEntity **extends** User {  
  
 */\*\*  
 \* Имя пользователя администратора.  
 \*/* @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 @Id  
 @Column(name = **"username"**, nullable = **false**, length = 20)  
 **private** String **username**;  
  
 */\*\*  
 \* Пароль администратора.  
 \*/* @Basic  
 @Column(name = **"password"**, nullable = **false**, length = 100)  
 **private** String **password**;  
  
 */\*\*  
 \* Получить имя пользователя администратора.  
 \*  
 \** ***@return*** *имя пользователя администратора  
 \*/* **public** String getUsername() {  
 **return username**;  
 }  
  
 */\*\*  
 \* Установить имя пользователя администратора.  
 \*  
 \** ***@param username*** *имя пользователя администратора  
 \*/* **public void** setUsername(String username) {  
 **this**.**username** = username;  
 }

Для чего Admin наследуется от класса User описано в главе 3.1.1.

Пример класса репозитория:

*/\*\*  
 \* Репозиторий {****@code*** *PatientRepository} для управления сущностями администратора {****@link*** *PatientEntity}.  
 \*/*@Repository  
**public interface** PatientRepository **extends** JpaRepository<PatientEntity, Integer> {  
  
 */\*\*  
 \* Метод для поиска всех пациентов, соответствующих указанному шаблону 'example',  
 \* и возврата их с учетом настройки постраничности 'pageable'.  
 \*  
 \** ***@param example*** *шаблон для поиска сущностей  
 \** ***@param pageable*** *настройка постраничности  
 \** ***@return*** *страница пациентов, соответствующих указанному шаблону  
 \*/* Page<PatientEntity> findAll(Example example, Pageable pageable);  
  
 */\*\*  
 \* Возвращает опциональный объект с сущностью пользователя, связанной с заданным номером телефона.  
 \*  
 \** ***@param phone*** *номер телефона, связанный с искомым пользователем  
 \** ***@return*** *опциональный объект с найденной сущностью пользователя  
 \*/* Optional<User> findByPhoneNumber(String phone);  
  
 */\*\*  
 \* Возвращает опциональный объект с сущностью пациента, связанной с заданным номером телефона.  
 \*  
 \** ***@param phone*** *номер телефона, связанный с искомым пациентом  
 \** ***@return*** *опциональный объект с найденной сущностью пациента  
 \*/* Optional<PatientEntity> findPatientEntityByPhoneNumber(String phone);  
  
 */\*\*  
 \* Возвращает сущность пациента, связанную с заданным номером телефона.  
 \*  
 \** ***@param phone*** *номер телефона, связанный с искомым пациентом  
 \** ***@return*** *найденная сущность пациента  
 \*/* PatientEntity getByPhoneNumber(String phone);  
}

Как видим, репозиторий расширяет класс JpaRepository, который как раз позволяет удобно взаимодействовать с БД без необходимости в ручной реализации запросов на языке SQL.

Таким образом, использование Java Spring, Spring Data JPA и Hibernate позволило легко организовать взаимодействие сервера с базой данных, упростить работу с данными и ускорить процесс разработки приложения.

## **3.2. Взаимодействие с клиентом**

Для взаимодействия сервера с клиентом в приложении была выбрана архитектура клиент-сервер, где сервер предоставляет API-интерфейсы, через которые клиент получает доступ к данным и функциональности приложения. В данном случае, для реализации API-интерфейсов на сервере были использованы REST-контроллеры, которые обеспечивают передачу данных между клиентом и сервером в формате JSON.

Сами контроллеры в приложении "Платная поликлиника" были реализованы с помощью фреймворка Spring. Они позволяют обрабатывать запросы клиента и возвращать ответы в формате JSON. REST-контроллеры обеспечивают обработку запросов на создание, чтение, обновление и удаление данных (CRUD), что позволяет клиенту получать доступ к функциональности приложения и управлять им.

Взаимодействие между клиентом и сервером осуществляется с помощью HTTP-запросов. Клиент отправляет HTTP-запрос на сервер, содержащий необходимые данные или параметры, а сервер в свою очередь обрабатывает запрос и возвращает клиенту результат в формате JSON.

Для удобства код контроллеров был разделен на 4 класса, каждый со своим адресом:

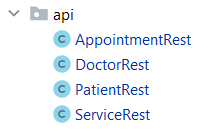
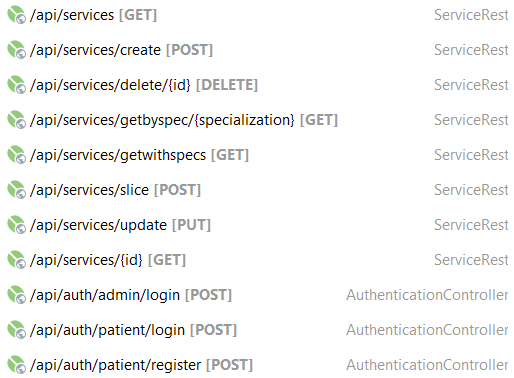
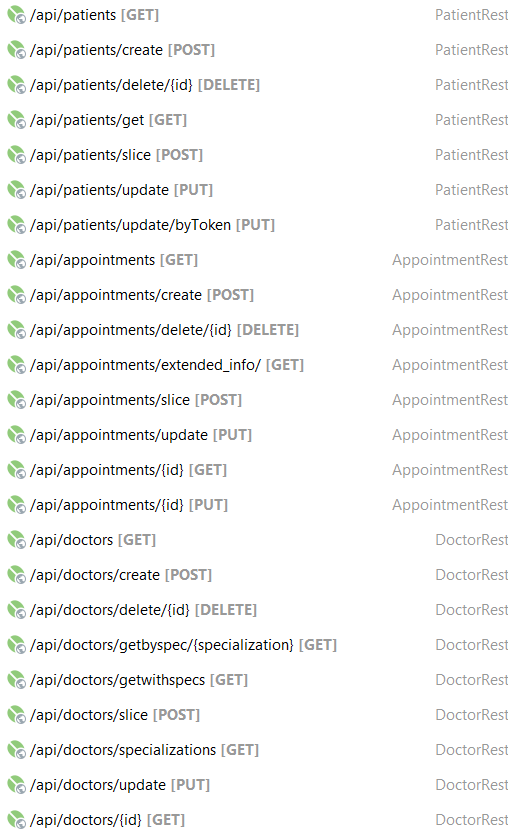


Рисунок 9

Пример реализации контроллера:

*/\*\*  
 \* Контроллер для работы с данными о пациентах.  
 \*/*@RestController  
@RequestMapping(**"/api/patients"**)  
**public class** PatientRest {  
  
 **private final** PatientRepository;  
  
 **private final** JwtService;  
  
 */\*\*  
 \* Конструктор, принимающий репозиторий пациентов и сервис для генерации JWT токенов.  
 \*  
 \** ***@param patientRepository*** *репозиторий пациентов  
 \** ***@param jwtService*** *сервис для генерации JWT токенов  
 \*/* **public** PatientRest(PatientRepository, JwtService jwtService) {  
 **this**.**patientRepository** = patientRepository;  
 **this**.**jwtService** = jwtService;  
 }  
  
 */\*\*  
 \* Получение списка всех пациентов.  
 \*  
 \** ***@return*** *список всех пациентов  
 \*/* @GetMapping  
 **public** List<PatientEntity> getAllPatients() {  
 **return patientRepository**.findAll();  
 }  
  
 */\*\*  
 \* Получение среза списка пациентов с сортировкой и пагинацией.  
 \*  
 \** ***@param pageIndex*** *номер страницы  
 \** ***@param pageSize*** *размер страницы  
 \** ***@param sortDirection*** *направление сортировки  
 \** ***@param sortBy*** *поле для сортировки  
 \** ***@param example*** *пример объекта пациента для фильтрации  
 \** ***@return*** *срез записей на прием с сортировкой и пагинацией  
 \*/* @PostMapping(**"/slice"**)  
 **public** Map<String, Object> getPatientsSlice(  
 @RequestParam **int** pageIndex,  
 @RequestParam **int** pageSize,  
 @RequestParam Sort.Direction sortDirection,  
 @RequestParam String sortBy,  
 @RequestBody(required = **false**) PatientEntity example  
 ) {  
 Page<PatientEntity> page;  
 **if** (example != **null**) {  
 ExampleMatcher matcher = ExampleMatcher.*matching*()  
 .withIgnoreCase()  
 .withIgnoreNullValues()  
 .withIgnorePaths(**"id"**)  
 .withStringMatcher(ExampleMatcher.StringMatcher.***CONTAINING***);  
 Example<Object> queryExample = Example.*of*(example, matcher);  
  
 page = **patientRepository**.findAll(queryExample, PageRequest.*of*(pageIndex, pageSize, sortDirection, sortBy));  
  
 } **else** {  
 page = **patientRepository**.findAll(PageRequest.*of*(pageIndex, pageSize, sortDirection, sortBy));  
 }  
  
 **int** totalPages = page.getTotalPages();  
  
 Map<String, Object> result = **new** HashMap<>();  
 result.put(**"data"**, page);  
 result.put(**"totalPages"**, totalPages);  
 **return** result;  
 }  
  
 */\*\*  
 \* Получение данных о пациенте по токену авторизации.  
 \*  
 \** ***@param token*** *токен авторизации  
 \** ***@return*** *объект пациента или ответ с статусом "не найдено"  
 \*/* @GetMapping(**"/get"**)  
 **public** ResponseEntity<PatientEntity> getPatientById(@RequestHeader(**"Authorization"**) String token) {  
  
 token = token.substring(7);  
 Optional<PatientEntity> patient = **patientRepository**.findPatientEntityByPhoneNumber(**jwtService**.extractUsername(token));  
  
 **return** patient.map(ResponseEntity::*ok*).orElseGet(() -> ResponseEntity.*notFound*().build());  
 }

Вот итоговый список всех конечных точек:



Рисунок

## 3.3. Защита от несанкционированного доступа

Одной из важнейших задач в разработке приложения "Платная поликлиника" было обеспечение безопасности пользовательских данных. Для этого были использованы механизмы защиты, авторизации и аутентификации, реализованные в Spring Security.

Spring Security — это фреймворк для обеспечения безопасности приложений на основе Spring. Он предоставляет механизмы защиты, которые позволяют ограничивать доступ к определенным ресурсам и действиям в приложении. В приложении "Платная поликлиника" Spring Security был использован для защиты API-интерфейсов, доступных только авторизованным пользователям, а также настроить разделение доступа для пациентов и администраторов.

Для реализации защиты были использованы механизмы Spring Security, которые позволяют проверять подлинность пользователей и ограничивать доступ к ресурсам приложения только авторизованным пользователям. Для этого в приложении была создана система ролей, которые определяют уровень доступа пользователя к определенным ресурсам. Все запросы на доступ к защищенным ресурсам проходят через механизм Spring Security, который проверяет, имеет ли пользователь необходимые права доступа к ресурсу.

### 3.1.1. Реализация пользователей

Так как приложение предполагает наличие двух видов пользователей (пациентов и администраторов), было решено создать абстрактный суперкласс User, реализующий интерфейс UserDetails пакета org.springframework.security.core.userdetails.

Код класса User:

*/\*\*  
 \* Абстрактный класс, представляющий собой пользователя системы, реализующий интерфейс UserDetails.  
 \* Содержит поля логина и пароля.  
 \*/***public abstract class** User **implements** UserDetails {  
  
 */\*\*  
 \* Логин пользователя.  
 \*/* **private** String **username**;  
 */\*\*  
 \* Пароль пользователя.  
 \*/* **private** String **password**;  
}

В свою очередь, от класса User наследуются классы PatientEntity и AdminEntity. В оба класса была добавлена реализация необходимых для работы SpringSecurity методов. Фрагмент кода класса PatientEntity:

@Override  
**public** Collection<? **extends** GrantedAuthority> getAuthorities() {  
 **return** List.*of*(**new** SimpleGrantedAuthority(Role.***USER***.name()));  
}  
  
**public** String getPassword() {  
 **return password**;  
}  
  
**public void** setPassword(String password) {  
 **this**.**password** = password;  
}  
  
@Override  
**public** String getUsername() {  
 **return phoneNumber**;  
}  
  
@Override  
**public boolean** isAccountNonExpired() {  
 **return true**;  
}  
  
@Override  
**public boolean** isAccountNonLocked() {  
 **return true**;  
}  
  
@Override  
**public boolean** isCredentialsNonExpired() {  
 **return true**;  
}  
  
@Override  
**public boolean** isEnabled() {  
 **return true**;  
}

В качестве логина пациента используется его номер телефона. В качестве роли для всех пациентов была установлена роль «USER». Было решено не добавлять поле «роль» в базу данных, так как предполагается, что все пациенты и администраторы будут иметь одинаковые роли «USER» и «ADMIN» соответственно.

Остальные методы возвращают значение «true», так как для упрощения и ускорения разработки было решено не использовать механизмы блокировки и отключения учетных записей пользователей.

Возможные роли описаны в перечислении «Role»:

*/\*\*  
 \* Перечисление, содержащее возможные роли пользователей системы.  
 \* Может быть присвоена каждому пользователю системы.  
 \* Возможные роли: USER, ADMIN.  
 \*/***public enum** Role {  
 ***USER***,  
 ***ADMIN***}

Реализация поиска пользователя по логину прописана в репозитории «UserRepository»:

*/\*\*  
 \* Репозиторий, предоставляющий метод для поиска пользователя по его имени пользователя.  
 \*/*@Repository  
**public class** UserRepository {  
  
 */\*\*  
 \* Репозиторий пациентов, используемый для поиска пациентов по имени пользователя.  
 \*/* **private final** PatientRepository;  
  
 */\*\*  
 \* Репозиторий администраторов, используемый для поиска администраторов по имени пользователя.  
 \*/* **private final** AdminRepository;  
  
 */\*\*  
 \* Конструктор, инициализирующий репозитории пациентов и администраторов.  
 \*  
 \** ***@param patientRepository*** *Репозиторий пациентов.  
 \** ***@param doctorRepository*** *Репозиторий докторов (не используется).  
 \** ***@param adminRepository*** *Репозиторий администраторов.  
 \*/* **public** UserRepository(PatientRepository patientRepository, DoctorRepository, AdminRepository adminRepository) {  
 **this**.**patientRepository** = patientRepository;  
 **this**.**adminRepository** = adminRepository;  
 }  
  
 */\*\*  
 \* Метод для поиска пользователя по его имени пользователя.  
 \* Если пользователь с таким именем найден в репозитории пациентов, возвращается соответствующий пациент,  
 \* если в репозитории администраторов - возвращается соответствующий администратор.  
 \* Если пользователь не найден в обоих репозиториях, выбрасывается исключение UsernameNotFoundException.  
 \*  
 \** ***@param username*** *Имя пользователя.  
 \** ***@return*** *Найденный пользователь.  
 \** ***@throws*** *UsernameNotFoundException Исключение, выбрасываемое, если пользователь не найден.  
 \*/* **public** Optional<User> findByUsername(String username) {  
 Optional<User> admin = **adminRepository**.findByUsername(username);  
 **if** (admin.isPresent()) {  
 **return** admin;  
 }  
 Optional<User> patient = **patientRepository**.findByPhoneNumber(username);  
 **if** (patient.isPresent()) {  
 **return** patient;  
 }  
 **throw new** UsernameNotFoundException(**"User not found"**);  
 }  
}

### 3.3.2. Аутентификация

Для аутентификации и регистрации пользователей в системе был написан отдельный REST-контроллер AuthenticationController, реализующий соответствующие методы.

Данный контроллер использует сервис AuthenticationService, реализующий логику аутентификации и регистрации.

Пример кода аутентификации администратора:

*/\*\*  
 \* Метод для аутентификации администратора и генерации JWT-токена.  
 \*  
 \** ***@param request*** *Запрос аутентификации администратора  
 \** ***@return*** *ResponseEntity со статусом ОК и объектом AuthenticationResponse, содержащим JWT-токен  
 \*/***public** AuthenticationResponse authenticateAdmin(AuthenticationRequest request) {  
 **authenticationManager**.authenticate(  
 **new** UsernamePasswordAuthenticationToken(  
 request.getUsername(),  
 request.getPassword()  
 )  
 );  
 AdminEntity admin = (AdminEntity) **adminRepository**.findByUsername(request.getUsername())  
 .orElseThrow();  
 **var** jwtToken = **jwtService**.generateToken(admin);  
 **return** AuthenticationResponse.*builder*()  
 .token(jwtToken)  
 .build();  
}

Процесс аутентификации происходит при помощи менеджера AuthenticationManager пакета org.springframework.security.authentication и сервиса JwtService, который будет описан ниже.

Для регистрации в системе используется также шифровальщик паролей PasswordEncoder из пакета org.springframework.security.crypto.password.

### 3.3.3. Работа с токенами

Для работы с токенами авторизации был написан сервис JwtService, использующий библиотеку jsonwebtoken.

Для подписи Jwt-токена используется 256-битный секретный ключ:

**private static final** String ***SECRET\_KEY*** = **"753778214125442A472D4B6150645367566B59703373367638792F423F452848"**;

Сервис реализует методы для генерации токена, проверки его валидности, и извлечения данных из токена.

### 3.3.4. Конфигурация

В классе ApplicationConfig – конфигурации всего приложения, реализовано создание сервиса пользовательских данных, провайдера и менеджера аутентификации и кодировщика паролей.

Конфигурация защиты описана в классе SecurityConfiguration. Для фильтрации запросов к серверу используется бин securityFilterChain, реализующий цепочку проверок прав доступа к конечным точкам.

Код цепочки фильтров:

*/\*\*  
 \* Создание цепочки фильтров для обработки запросов с учетом правил доступа. В частности, включает  
 \* установку правил доступа на основе ролей пользователя и настройку фильтров аутентификации.  
 \*  
 \** ***@param http*** *- объект, представляющий настройки безопасности HTTP-запросов.  
 \** ***@return*** *объект типа SecurityFilterChain - цепочка фильтров, обрабатывающая запросы с учетом правил доступа.  
 \** ***@throws*** *Exception если возникает ошибка при настройке цепочки фильтров безопасности.  
 \*/*@Bean  
**public** SecurityFilterChain securityFilterChain(HttpSecurity http) **throws** Exception {  
 http  
 .csrf().disable()  
 .cors().and()  
 .authorizeHttpRequests()  
 .requestMatchers(**"/api/auth/\*\*"**,  
 **"/api/doctors/getwithspecs"**,  
 **"/api/services/getwithspecs"**).permitAll()  
 .requestMatchers(**"/api/patients/get/\*\*"**,  
 **"/api/patients/update/\*\*"**,  
 **"/api/appointments/extended\_info/\*\*"**,  
 **"/api/doctors/specializations"**,  
 **"/api/services/getbyspec/\*\*"**,  
 **"/api/doctors/getbyspec/\*\*"**,  
 **"/api/appointments/create"**).hasAnyAuthority(**"USER"**, **"ADMIN"**)  
 .anyRequest().hasAuthority(**"ADMIN"**)  
 .and()  
 .sessionManagement()  
 .sessionCreationPolicy(SessionCreationPolicy.***STATELESS***)  
 .and()  
 .authenticationProvider(**authenticationProvider**)  
 .addFilterBefore(**jwtAuthFilter**, UsernamePasswordAuthenticationFilter.**class**);  
  
 **return** http.build();  
}

Здесь определены адреса, доступные для всех, в том числе анонимных пользователей. Адрес “/api/auth/\*\*” ведет к контроллерам авторизации и регистрации. Другие два адреса используются при на клиентской стороне для отображения информации, доступной всем.

Роли USER доступны лишь некоторые методы, связанные с просмотром и изменением своих персональных данных.

Все остальные конечные точки доступны только администраторам.

Дополнительно был написан класс конфигурации CORS (Cross-origin resource sharing), реализующий интерфейс WebMvcConfigurer.

Для тестирования в нем установлена конфигурация, разрешающая доступ из любого источника, но при развертывании приложения на постоянном сервере следовало бы установить только адрес сервера, на котором находится представление веб-приложения.

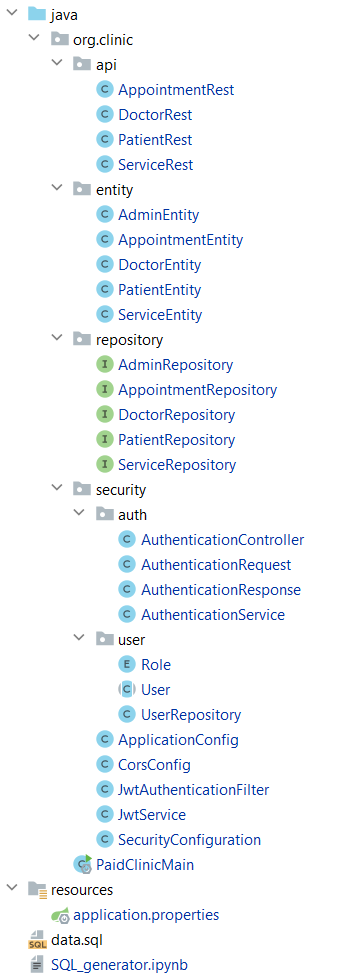
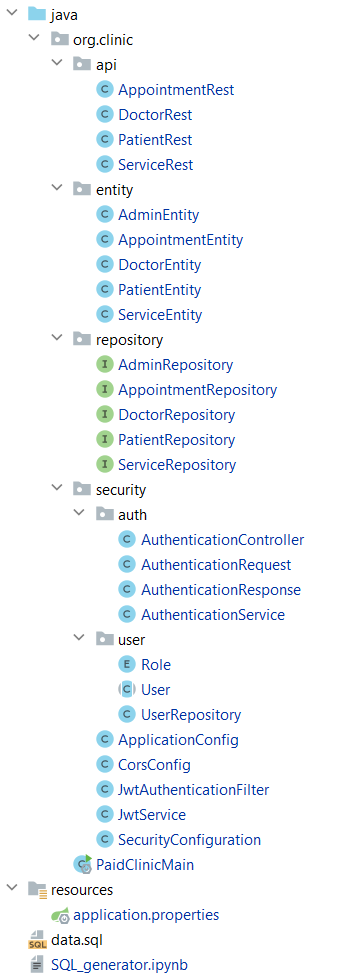
*/\*\*  
 \* Класс конфигурации для Cross-Origin Resource Sharing (CORS).  
 \* Позволяет настроить доступ к ресурсам с других источников.  
 \*/*@Configuration  
**public class** CorsConfig **implements** WebMvcConfigurer {  
  
 */\*\*  
 \* Регистрирует отображения CORS-запросов.  
 \* Разрешены любые источники.  
 \* Разрешены методы "GET", "POST", "PUT", "DELETE".  
 \* Разрешены любые заголовки.  
 \*  
 \** ***@param registry*** *Реестр CORS-отображений.  
 \*/* @Override  
 **public void** addCorsMappings(CorsRegistry registry) {  
 registry.addMapping(**"/\*\*"**)  
 .allowedOrigins(**"\*"**)  
 .allowedMethods(**"GET"**, **"POST"**, **"PUT"**, **"DELETE"**)  
 .allowedHeaders(**"\*"**)  
 .maxAge(3600);  
 }  
}

Таким образом, использование механизмов защиты, авторизации и аутентификации в приложении "Платная поликлиника" позволяет обеспечить безопасность пользовательских данных и защитить приложение от несанкционированного доступа.

## 3.4. Резюме

Разработка серверной части веб-приложения включает в себя реализацию взаимодействия с базой данных, которое осуществляется при помощи Java фреймворка Spring и JPA + Hibernate, настройку взаимодействия сервера с клиентской частью, при помощи RestController. Для защиты от несанкционированного доступа используется Spring Security, предоставляющий возможность использовать различных пользователей с ролями, аутентификацию, работу с токенами и фильтрацию доступа.

Конечная структура кода сервера выглядит так:



Рисунок

Подключенные в файле pom.xml зависимости:

* spring-boot-starter-data-jpa
* spring-boot-starter-security
* spring-security-test
* spring-boot-starter-freemarker
* spring-boot-devtools
* postgresql
* jjwt-api
* jjwt-impl
* jjwt-jackson
* javax.servlet-api
* jjwt
* jakarta.xml.bind-api
* jakarta.servlet-api
* lombok
* commons-beanutils

# Глава 4. Клиентская часть

Клиентская часть приложения "Платная поликлиника" является важным компонентом, предоставляющим интерфейс для взаимодействия с серверной частью. Для разработки клиентской части был выбран React - популярный JavaScript-фреймворк для разработки пользовательских интерфейсов. Он предоставляет широкие возможности по созданию многокомпонентных приложений и упрощает разработку.

Архитектура клиентской части приложения включает в себя множество компонентов, отвечающих за отображение страниц и интерфейса пользователя.

Особенности реализации клиентской части заключаются в том, что она была разработана с учетом принципов дизайна, обеспечивающих удобство и простоту использования для конечного пользователя.

## 4.1. Структура

Структура директории src приложения:

* components – директория, содержащая React-компоненты
* hoc – компоненты высшего порядка (Higher-Order Component)
* hook – пользовательские хуки, упрощающие взаимодействие с hoc
* images – изображения

Структура директории public:

* images – публичные изображения
* favicon.ico – иконка сайта, отображающаяся на панели вкладок и в поисковой системе
* index.html – документ, представляющий собой каркас для React - приложения

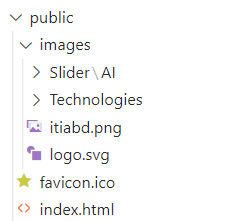
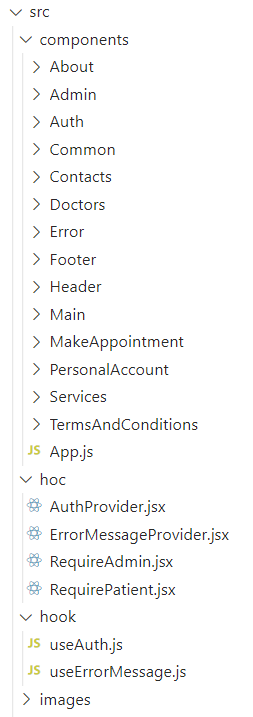


Рисунок Рисунок 13

## 4.2. Компоненты

Для написания компонентов использовалось расширение языка JavaScript – JSX, которое позволяет удобно описывать пользовательский интерфейс, используя HTML-разметку прямо в JS коде. При этом разметка автоматически транслируется в обычный JS код. В основном использовались функциональные объекты React, так как они значительно удобнее классов и отлично поддерживаются в последних версиях React. Для стилизации компонентов использовался код SCSS и, в качестве эксперимента, module.css (для компонента Header), но такой формат мне показался значительно менее удобным, чем SCSS. Файлы стилей хранятся в одних директориях с компонентами, к которым они относятся. Для реализации логики приложения по возможности использовались последние нововведения React, и различные хуки вроде useState, useEffect, useRef, useContext и т. п. Вот пример кода одного из компонентов:

import axios from 'axios';

import React, { useContext, useState } from 'react'

import { useForm } from "react-hook-form";

import { useLocation, useNavigate } from 'react-router-dom'

import { useAuth } from '../../hook/useAuth';

import { RootContext } from '../..';

export const AdminAuth = () => {

    const { register, handleSubmit } = useForm();

    const [error, setError] = useState(false)

    const navigate = useNavigate();

    const location = useLocation();

    const { signAdminIn } = useAuth();

    const { server } = useContext(RootContext);

    const fromPage = location.state?.from?.pathname || "/admin/";

    const onSubmit = (data) => {

        axios.post(`${server}/api/auth/admin/login`, data)

            .then(response => {

                const admin = response.data.token;

                signAdminIn(admin, () => navigate(fromPage, { replace: true }));

            })

            .catch(error => {

                console.error(error);

                setError(true);

                setTimeout(() => setError(false), 300)

            });

    }

    return (

        <div className="container">

            <div className='big\_title'>{fromPage !== "/admin/auth" ? "Необходимо авторизоваться" : "Авторизация"}</div>

            <form className='auth\_form admin\_form' onSubmit={handleSubmit(onSubmit)}>

                <div className='title'>Вход</div>

                <input

                    title='Username'

                    minLength={1}

                    maxLength={20}

                    type='text'

                    placeholder='Username'

                    required

                    autoComplete='username'

                    {...register('username')} />

                <input

                    title='Password'

                    pattern='.{1,100}'

                    type='password'

                    placeholder='Password'

                    required

                    autoComplete="current-password"

                    {...register('password')} />

                <button className={`button ${error ? "error" : ""}`} type='submit'>Войти</button>

            </form>

        </div>

    )

}

Использование React позволяет удобно и быстро реализовывать логику приложения, в отличие от нативного HTML + JS, а богатая документация, отличное сообщество и множество библиотек, написанных специально для React, позволяют без особых затруднений решить почти любую задачу.

Так как для админ-панели предстояла задача реализовать манипулирование большими объемами данных, таблицы, отображающее информацию из базы данных, используют пагинацию. Данный прием позволяет загружать данные из базы данных постранично, что улучшает общую производительность.

Для удобной работы с данными таблицы так же оснащены фильтрами и возможностью сортировки, так что найти интересующую запись в базе данных не составит труда.

## 4.3. Аутентификация и взаимодействие с сервером

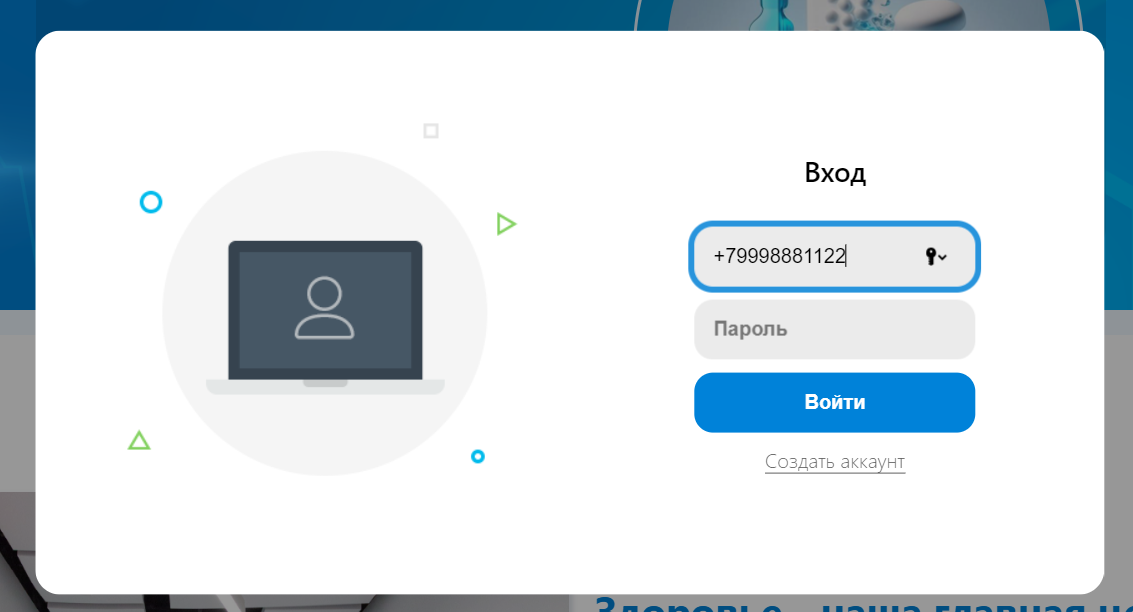
Для предоставления пользователям возможности войти в систему или зарегистрироваться были написаны компоненты Auth (для простых клиентов), AdminAuth (для администраторов) и hoc AuthProvider, сохраняющий текущее состояние аутентификации и предоставляющий контекст для всех остальных компонентов.



Рисунок

Алгоритм входа в свой аккаунт.

1. Пользователь вводит свои данные в форму и нажимает кнопку «Войти».



Рисунок

1. Значения из форм собираются при помощи функций, предоставляемых библиотекой react-hook-form.
2. Полученные данные отправляются на сервер при помощи HTTP библиотеки axios, позволяющей отправлять запросы к заданным конечным точкам:

    const onSubmit = (data) => {

        axios.post(`${server}/api/auth/patient/login`, data)

            .then(response => {

                let patient = response.data;

                signPatientIn(patient, () => { });

                handleSuccess();

                navigate("/lk", { replace: true });

            })

            .catch(error => {

                console.error(error);

                setError(true);

                setTimeout(() => setError(false), 300)

            });

    };

1. При успешной проверке учетных данных сервер возвращает токен авторизации, а React вызывает функцию signPatientIn(), прописанную в хуке AuthProvider:

    const signPatientIn = (newPatient, cb) => {

        localStorage.patientToken = newPatient.token;

        setPatient(newPatient)

        cb();

    }

1. Для хранения токена авторизации было решено использовать localStorage, так как это проще, чем реализовывать хранение данных в куки (cookie).

До получения токена авторизации пациенту недоступны страницы личного кабинета и записи на прием к врачу. Ограничение доступа реализовано при помощи другого hoc – RequirePatient, который перенаправляет любые запросы к защищенным страницам обратно на главную:

import { Navigate } from 'react-router-dom'

import { useAuth } from '../hook/useAuth';

export const RequirePatient = ({ children }) => {

    const { patient } = useAuth();

    if (!patient) {

        return <Navigate to='/' state={{ showAuthForm: true }} />

    }

    return children;

}

При наличии токена пользователь получает доступ к личному кабинету, и все запросы на получение / отправку защищенных данных снабжаются заголовком Authorization, содержащим токен. Пример функции, позволяющей клиенту получить информацию о себе:

    const getPatient = async () => {

        axios.get(url + `/get`, {

            headers: {

                Authorization: `Bearer ${token}`,

            }

        })

            .then(res => {

                const patient = res.data;

                setPatient(patient);

            })

    }

Аутентификация администратора происходит схожим образом, аналогично, доступ к страницам администратора ограничен при помощи hoc RequireAmin, а все HTML-запросы сопровождаются уже токеном администратора.

## 4.4. Маршрутизация

Для навигации по сайту использовалась библиотека react-router-dom, которая предоставляет объекты BrowserRouter, Routes, Route – для маршрутизации страниц сайта; Link, NavLink – ссылки на маршрут, на замену <a></a> и функцию navigate(), которая позволяет перенаправить пользователя на другую страницу без использования ссылки.

Список маршрутов сайта:

class App extends Component {

  render() {

    return (

      <ErrorMessageProvider>

        <AuthProvider>

          <BrowserRouter basename='/'>

            <Routes>

              <Route path='/' element={<PageWrapper />}>

                <Route index element={<Main />} />

                <Route path='lk' element={

                  <RequirePatient>

                    <PersonalAccount />

                  </RequirePatient>

                } />

                <Route path='make\_appointment' element={

                  <RequirePatient>

                    <MakeAppointment />

                  </RequirePatient>

                } />

                <Route path='services' element={<Services />} />

                <Route path='doctors' element={<Doctors />} />

                <Route path='contacts' element={<Contacts />} />

                <Route path='about' element={<About />} />

                <Route path='agreement' element={<TermsAndConditions />} />

                <Route path='\*' element={<Error error='404' errorInfo='Запрашиваемая страница не найдена' />} />

              </Route>

              <Route path='/admin' element={<AdminPage />}>

                <Route path='auth' element={<AdminAuth />} />

                <Route index element={

                  <RequireAdmin>

                    <AdminGreetings />

                  </RequireAdmin>

                } />

                <Route path='patients' element={

                  <RequireAdmin>

                    <AdminPatients />

                  </RequireAdmin>

                } />

                <Route path='doctors' element={

                  <RequireAdmin>

                    <AdminDoctors />

                  </RequireAdmin>

                } />

                <Route path='services' element={

                  <RequireAdmin>

                    <AdminServices />

                  </RequireAdmin>

                } />

                <Route path='appointments' element={

                  <RequireAdmin>

                    <AdminAppointments />

                  </RequireAdmin>

                } />

                <Route path='statistics' element={

                  <RequireAdmin>

                    <AdminStatistics />

                  </RequireAdmin>

                } />

                <Route path='\*' element={

                  <RequireAdmin>

                    <Error error='404' errorInfo='Запрашиваемая страница не найдена' />

                  </RequireAdmin>

                } />

              </Route>

            </Routes>

          </BrowserRouter>

        </AuthProvider>

      </ErrorMessageProvider>

    );

  }

}

В этом коде как раз можно заметить два глобальных контекста, используемые на всех страницах: ErrorMessageProvider – позволяет вывести на экран сообщение об успехе или ошибке и AuthProvider – провайдер аутентификации. Далее описаны все пути маршрутизации. Некоторые страницы обернуты в теги RequirePatient или RequireAdmin – ограничители доступа к странице незарегистрированным пользователям.

# Глава 5. Возможности приложения

## 5.1. Анонимный пользователь

Возможности, доступные неавторизированным пользователям:

* Просматривать главную страницу сайта, содержащую рекламную информацию о клинике
* Просматривать страницу, описывающую врачей и их специализации
* Просматривать страницу со всеми услугами
* Просматривать страницу с контактами (на этой странице расположены номера телефонов-заглушки и адрес Финансового Университета с интегрированной Яндекс-картой
* Просматривать страницу «Об авторе», содержащую информацию обо мне
* Входить в систему в качестве пациента или администратора, регистрироваться

## 5.2. Пациент

Возможности пациента:

* Все возможности анонимного пользователя
* Просматривать и редактировать информацию о себе
* Просматривать свою историю приемов у врача
* Записываться на новый прием у врача

## 5.3. Администратор

Возможности администратора:

* Все возможности анонимного пользователя
* Просматривать, изменять, создавать, удалять записи базы данных с врачами, пациентами, услугами и приемами
* Просматривать статистику приемов – количество и общую стоимость за каждый день заданного месяца в виде графиков

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы было разработано приложение "Платная поликлиника", предоставляющее пользователям возможность записаться на прием к врачу и просмотреть свою историю приемов. Также приложение включает в себя админ-панель, позволяющую работать с базой данных, производить операции CRUD и просматривать необходимую статистику.

Основными достижениями проекта являются успешная реализация клиент-серверной архитектуры приложения с использованием популярных технологий, таких как Java Spring, React и PostgreSQL. При этом приложение создавалось защищенным от несанкционированного доступа при помощи Spring Security.

Кроме того, были разработаны удобный интерфейс и функциональность приложения, позволяющие пользователям удобно пользоваться услугами клиники.

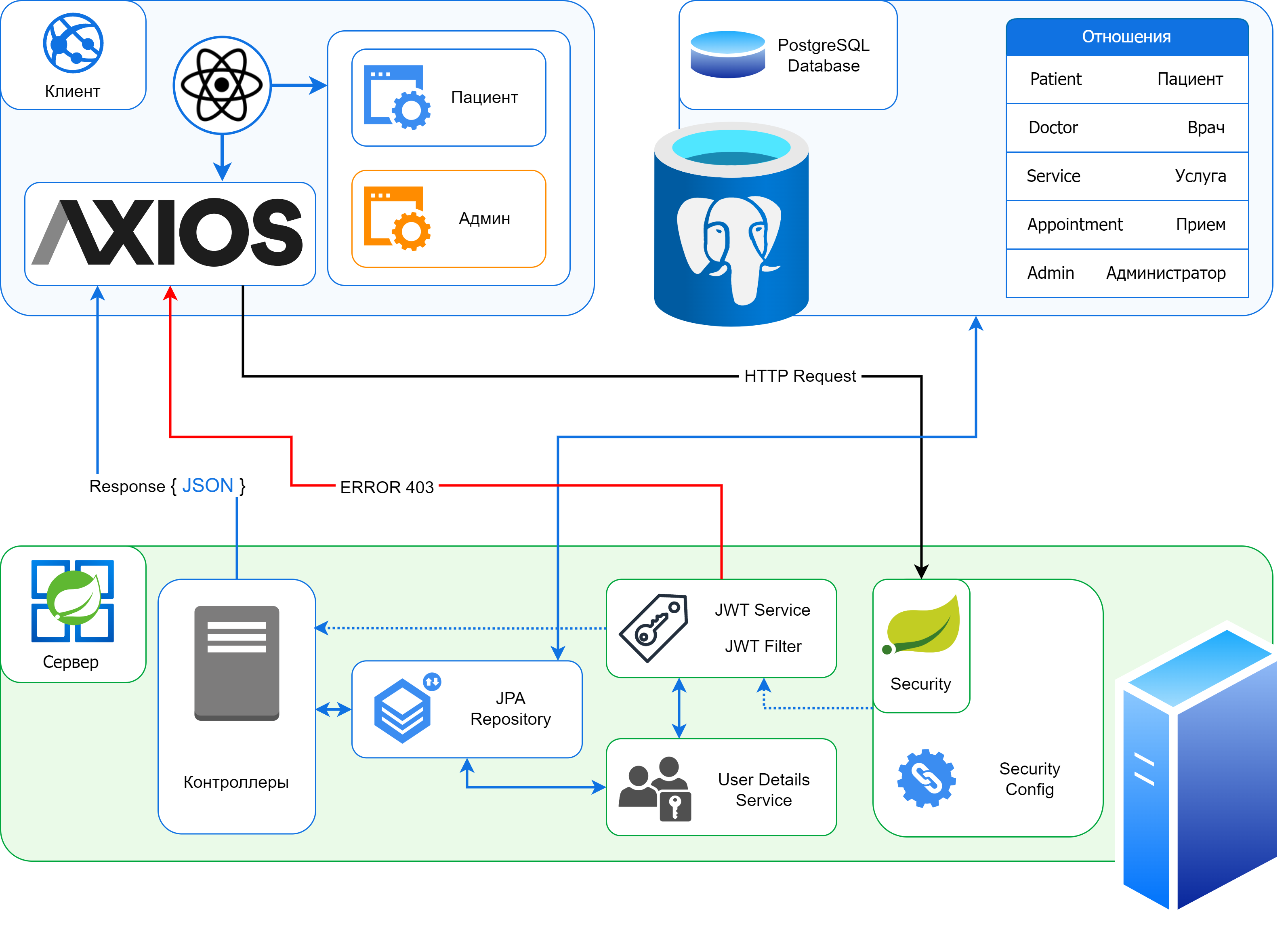
Основной трудностью была необходимость обеспечения безопасности и защиты данных пользователей, что потребовало значительных усилий для реализации. Задача усложнялась недостаточным количеством информации о новейших версиях Spring Security в Интернете. Фреймворк постоянно обновляется и некоторые классы и методы либо помечаются устаревшими (deprecated), либо вообще удаляются. К тому же сопутствующие библиотеки тоже постоянно обновляются и заменяются. Например, старые версии Spring Security использовали пакет Javax, но новые уже перешли на Jacarta. При этом возникала ситуация, когда для реализации одного и того же метода необходимо было использовать обе библиотеки, которые не совместимы друг с другом, вследствие чего было невозможно реализовать некоторые аспекты защиты.

Однако, при переходе на Spring Security 6 и новейшие версии сопутствующих библиотек, такой проблемы не возникало, так что в приложении получилось реализовать достаточно высокий уровень защиты и разграничить доступ для различных ролей пользователей.

Для дальнейшего улучшения приложения, возможно, потребуется более глубокое исследование в области оптимизации производительности и улучшения пользовательского интерфейса. Также можно рассмотреть возможность добавления новых функций и сервисов, которые могут сделать приложение более удобным и полезным для пользователей. Хорошим дополнением к графической части сайта было бы использование каркасных экранов – заглушек в тех местах, где необходимо ожидать информацию с сервера. Таким образом можно одновременно и сделать проект визуально более привлекательным, и показать пользователю, что сайт функционирует, и ему надо лишь немного подождать, пока загрузятся необходимые данные.

В целом приложение требует множество доработок помимо оптимизации и улучшения интерфейса. Полезным будет использование более конкретных ответов от контроллеров в случае ошибки, чтобы можно было реализовать уведомления для пользователей, что пошло не так, помимо уже реализованных сообщений «Сервер не отвечает» и «Ошибка».

Конечная архитектура приложения выглядит так:



Рисунок

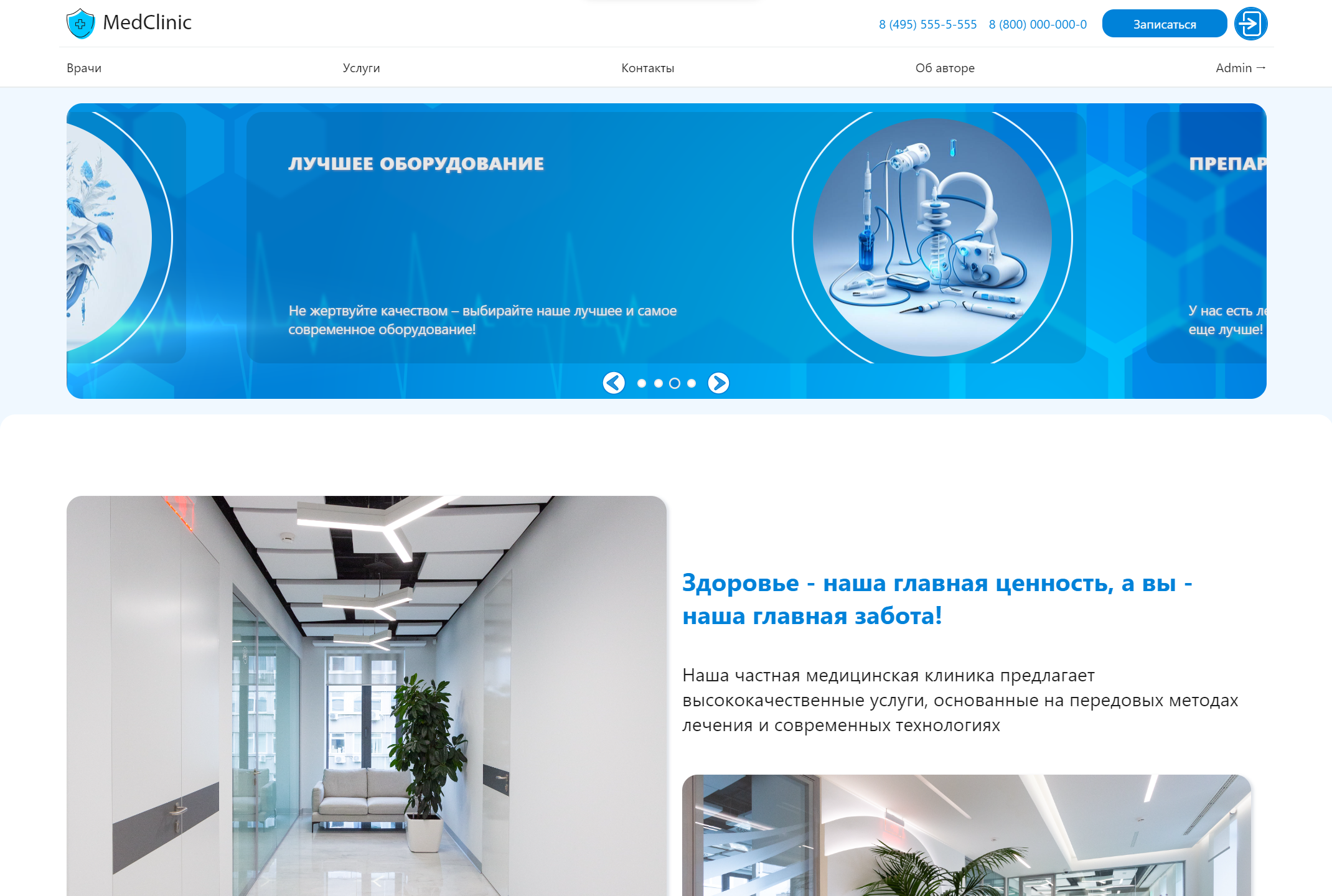
# ИСТОЧНИКИ

1. Spring Framework Documentation: Официальная документация, 2023 [Электронный ресурс]. URL: https://docs.spring.io/spring-framework/docs/current/reference/html/.
2. What’s New in SpringSecurity 6.0: Официальная документация, 2023 [Электронный ресурс]. URL: https://docs.spring.io/spring-security/reference/whats-new.html.
3. Bouali Ali. Spring Boot 3.0 Security with JWT Implementation: Демонстрационный проект, 2023. [Электронный ресурс]. URL: https://github.com/ali-bouali/spring-boot-3-jwt-security.
4. Блинов, И.Н. Java методы программирования / И.Н. Блинов, В.С. Романчик. – Минск: Изд-во «Четыре четверти», 2013.
5. Getting started – React: Официальная документация, 2023 [Электронный ресурс]. URL: https://legacy.reactjs.org/docs/getting-started.html.
6. React-table Docs: Официальная документация, 2023. [Электронный ресурс]. URL: https://tanstack.com/table/v8/docs/guide/introduction.
7. Recharts - A composable charting library built on React components: Официальная документация, 2023. [Электронный ресурс]. URL: https://recharts.org/en-US/.
8. React Router: Официальная документация, 2023. [Электронный ресурс]. URL: https://reactrouter.com/en/main.
9. Saas: Syntactically Awesome Style Sheets Официальная документация, 2023. [Электронный ресурс]. URL: https://sass-lang.com/.

# ПРИЛОЖЕНИЕ

## Руководство пользователя для пациента

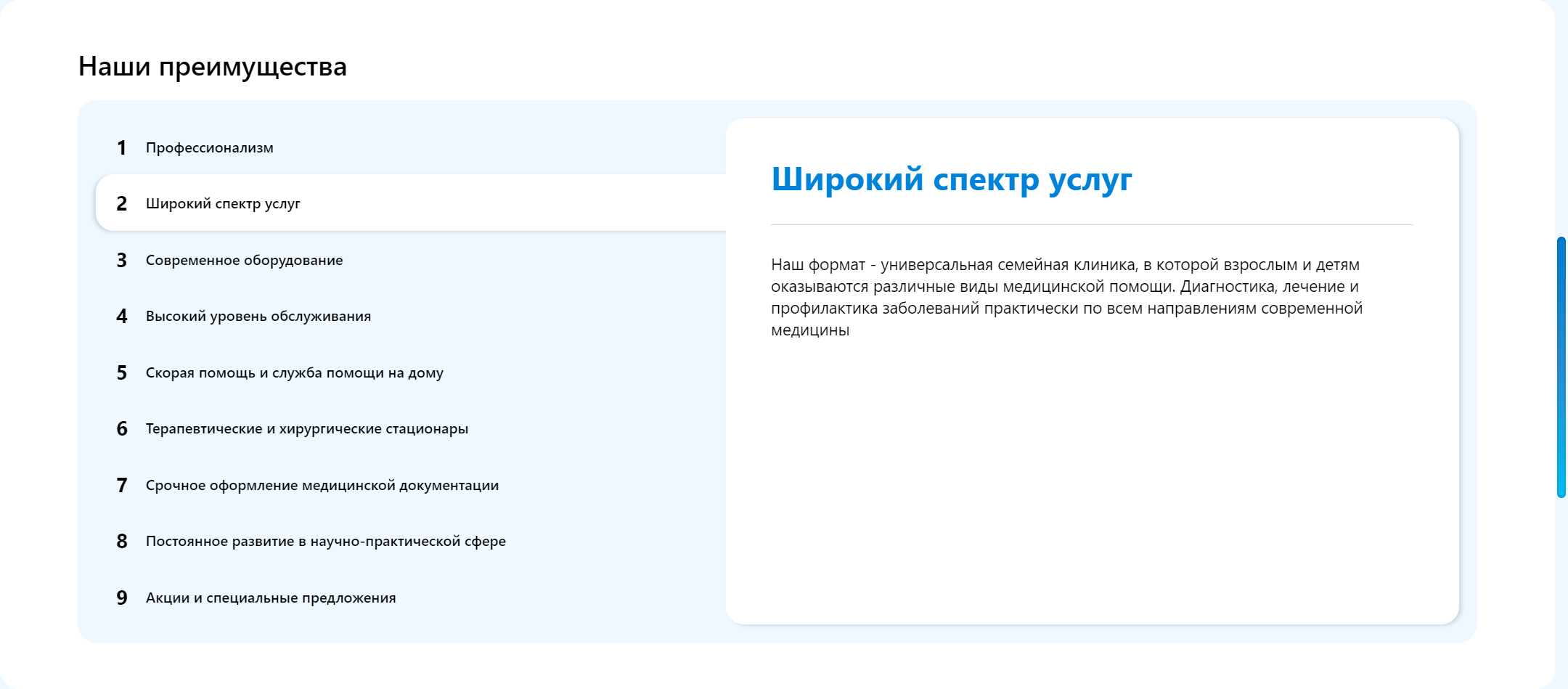
Главная страница сайта.



Приложение

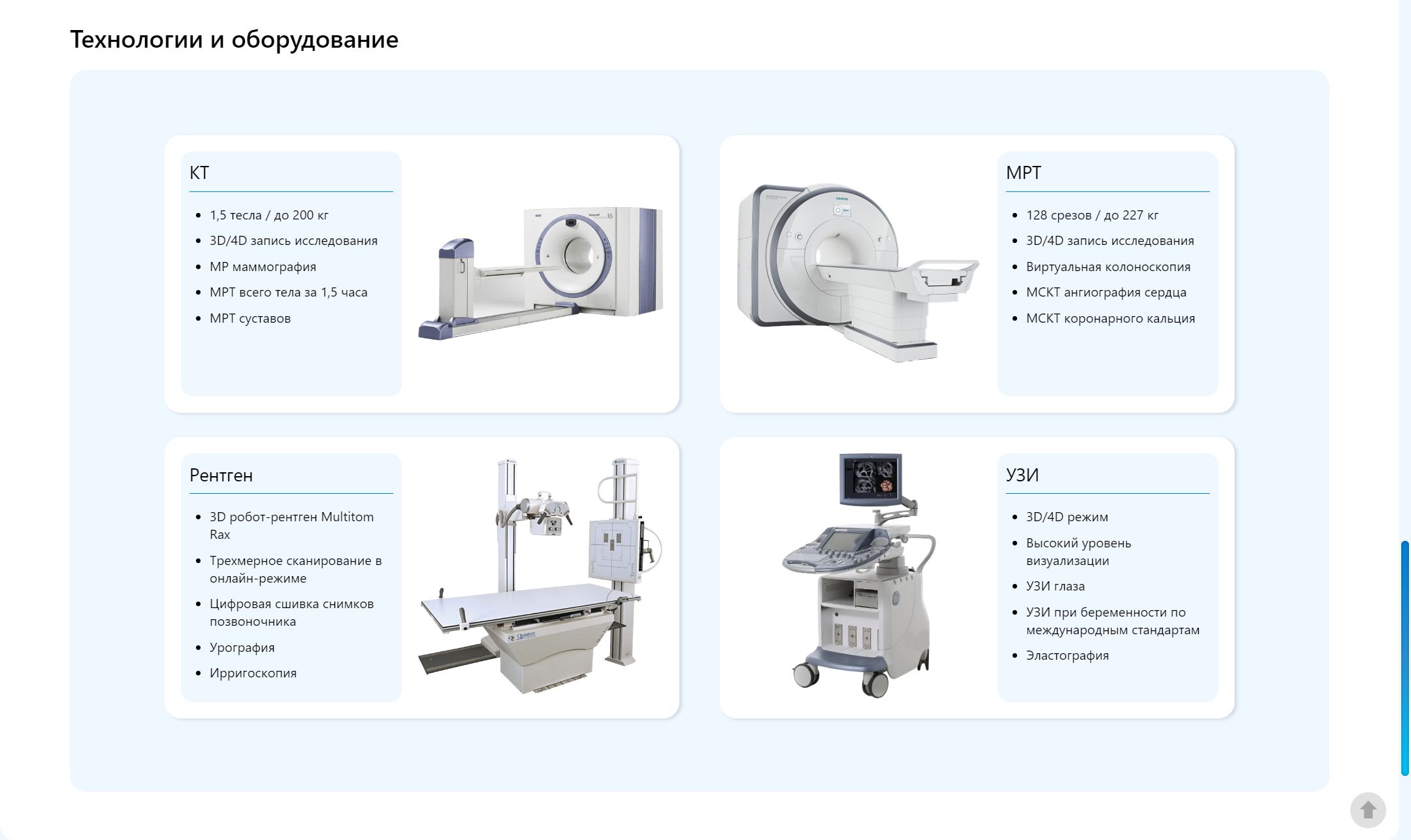
Здесь можно почитать демонстрационную информацию о клинике, разделенную на блоки.

Например, страница содержит информацию о преимуществах компании.



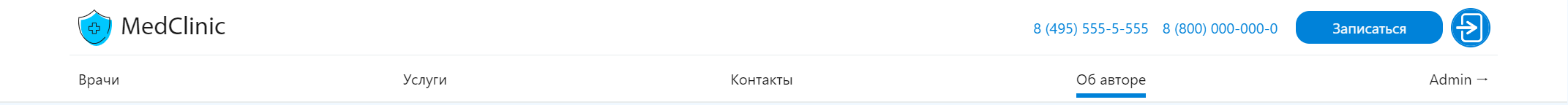
Приложение

И техническом оснащении.



Приложение

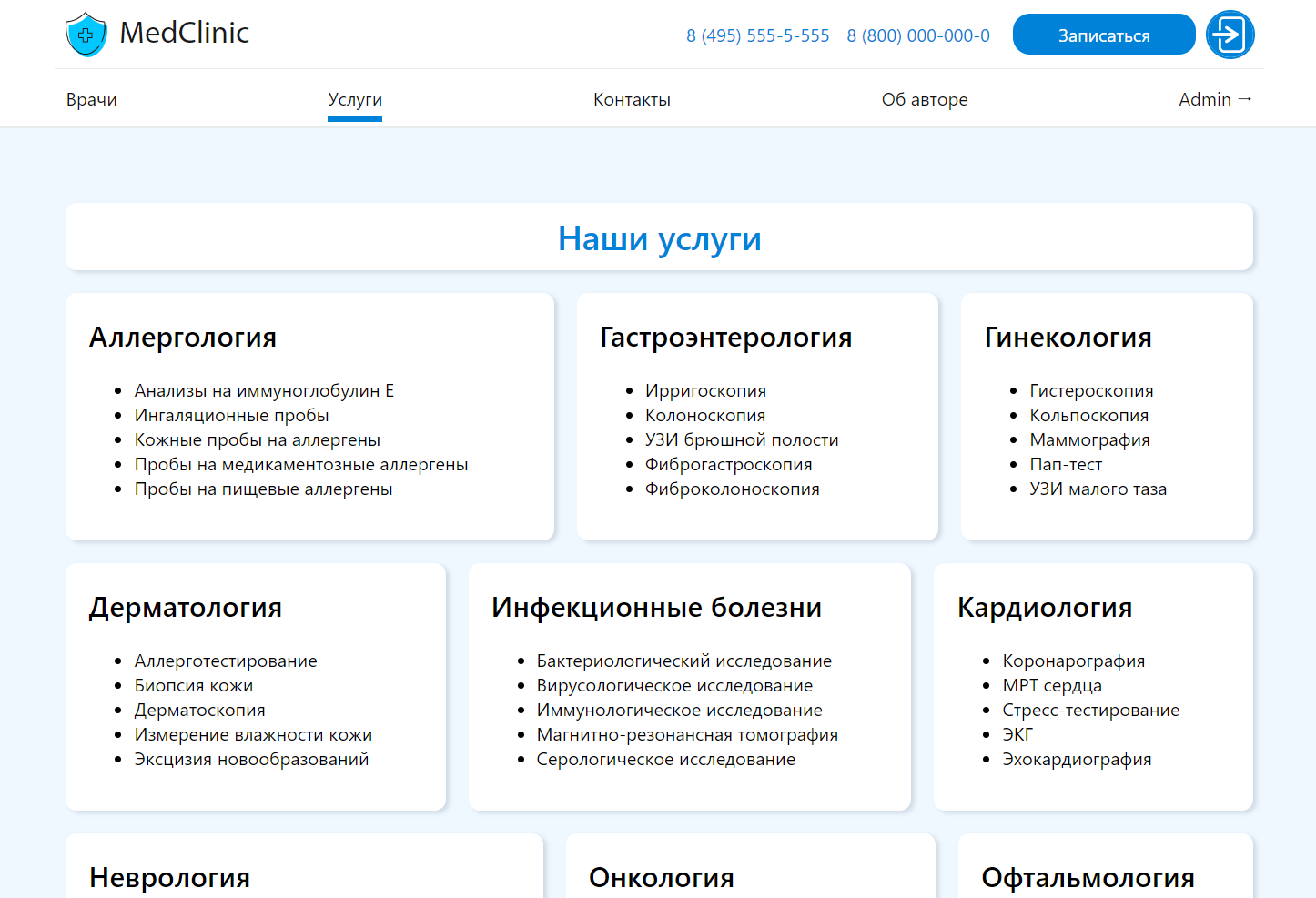
Шапка сайта содержит панель навигации и кнопки входа / регистрации и записи на прием .



Приложение

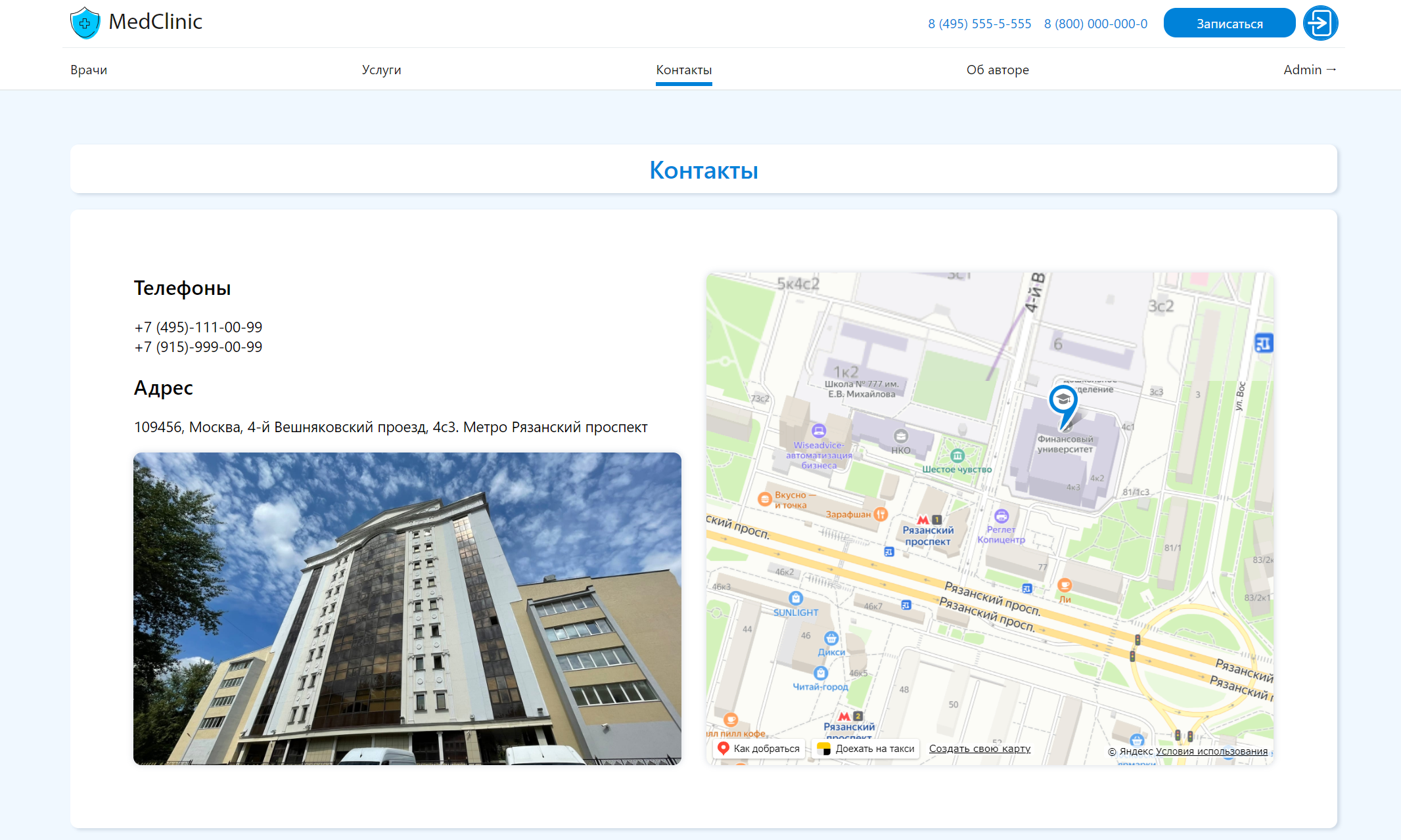
Неавторизированному пользователю доступны все страницы, указанные в навигационном меню.

Вот внешний вид страницы услуг, информация для неё подгружается с сервера на Java.



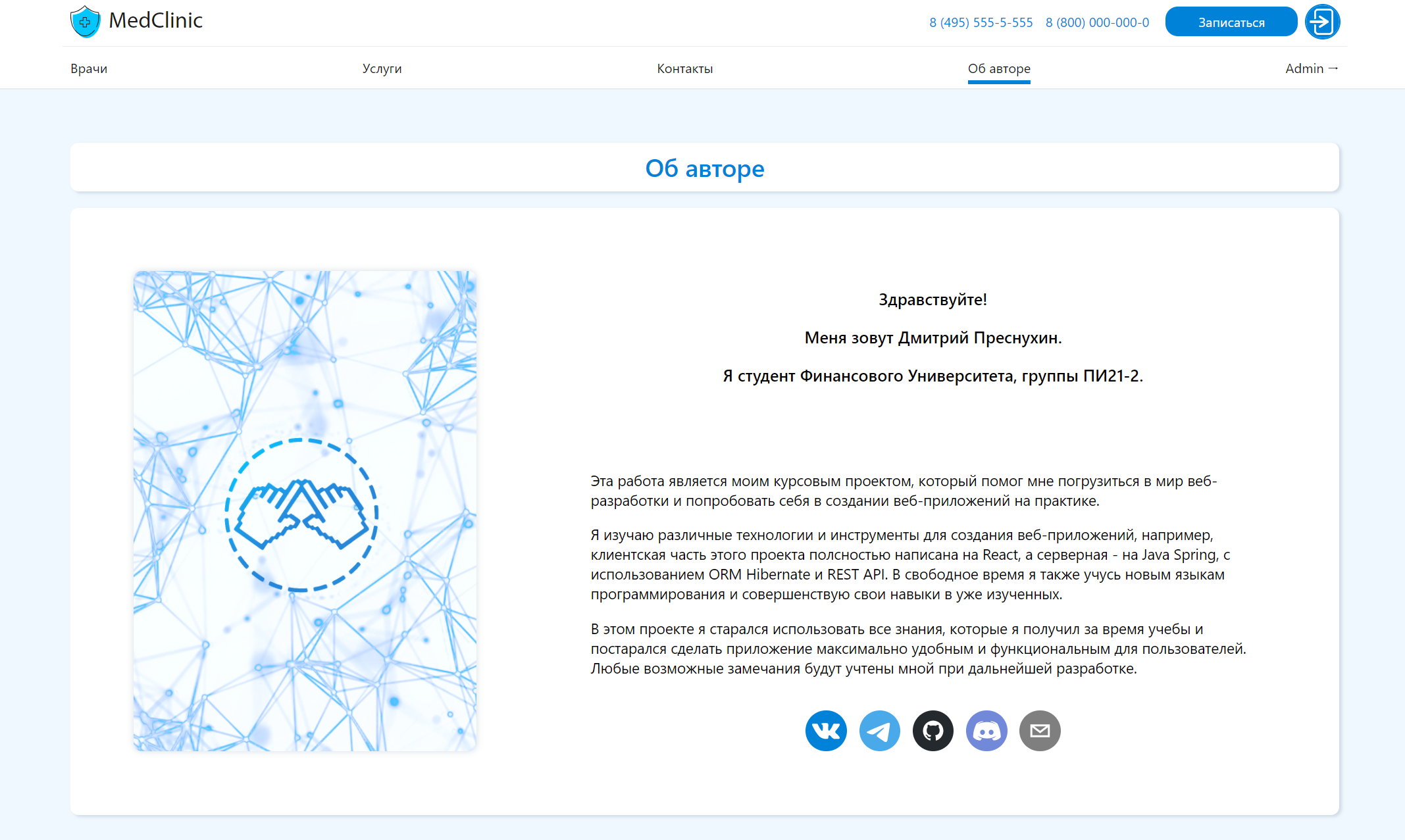
Приложение

На странице «Контакты» можно посмотреть информацию-заглушку о компании. Сейчас там указан адрес Финансового университета и несуществующие номера телефонов.



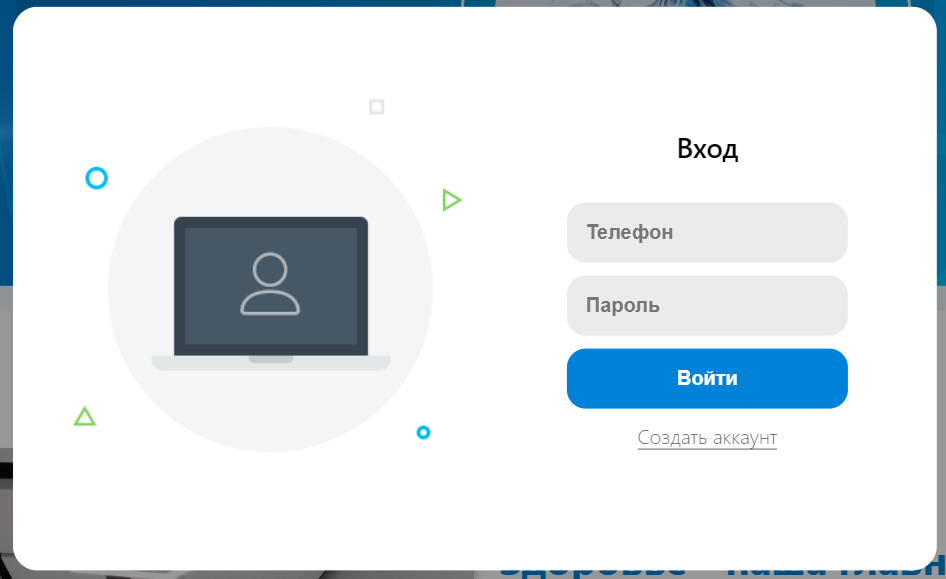
Приложение

Страница «Об авторе» содержит информацию обо мне и ссылки на социальные сети.

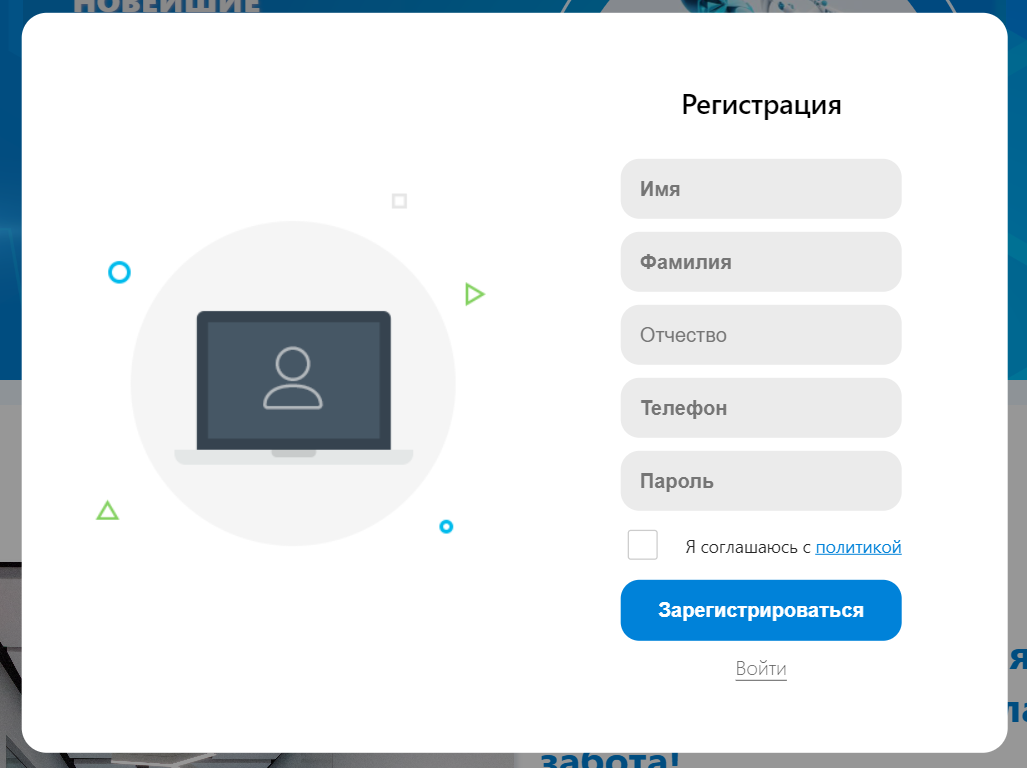


Приложение

При нажатии на кнопку входа или записи на прием пользователю будет предложено войти или зарегистрироваться.



Приложение

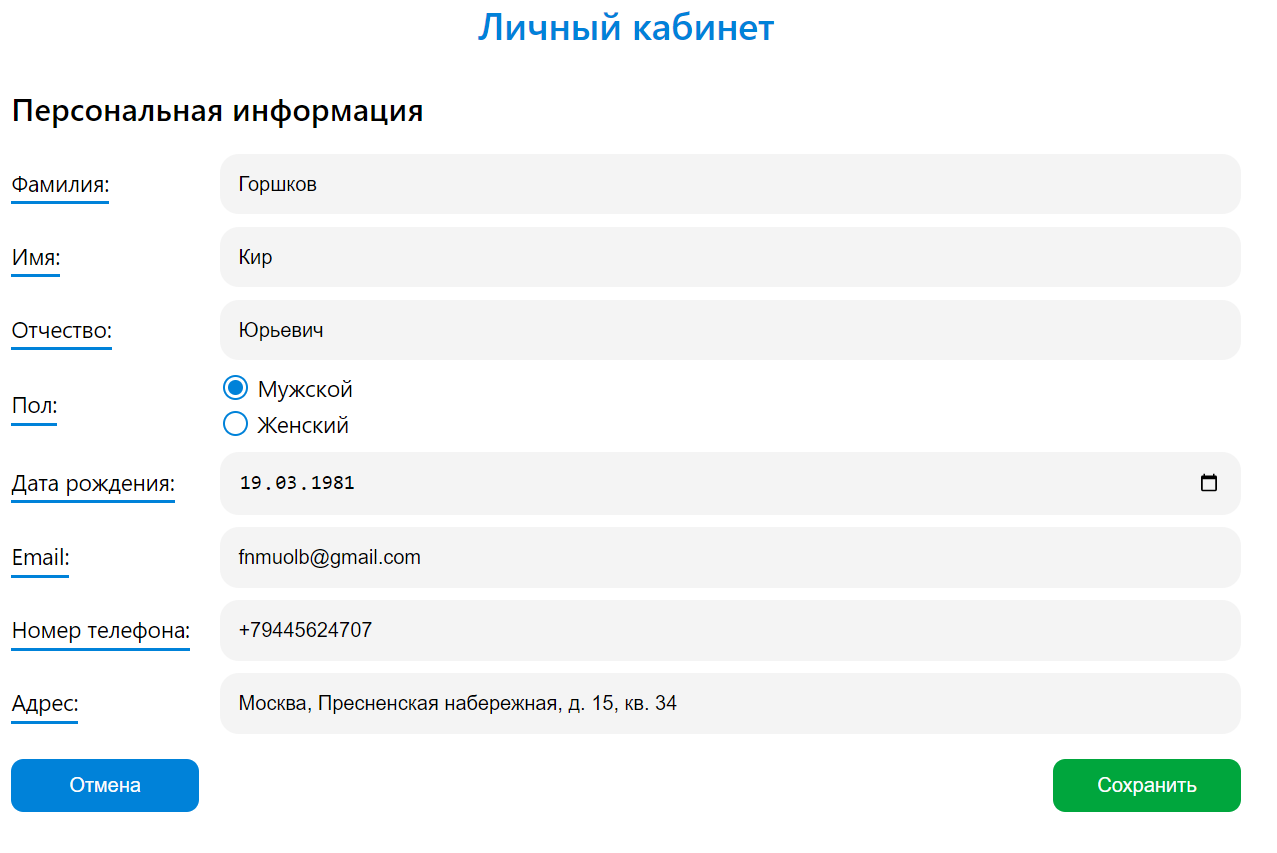


Приложение

Для демонстрации можно использовать номер телефона +79445624707 и пароль qwertyuiop.

После ввода учетных данных пользователя перенаправляет на страницу личного кабинета.

Здесь можно отредактировать информацию о себе.



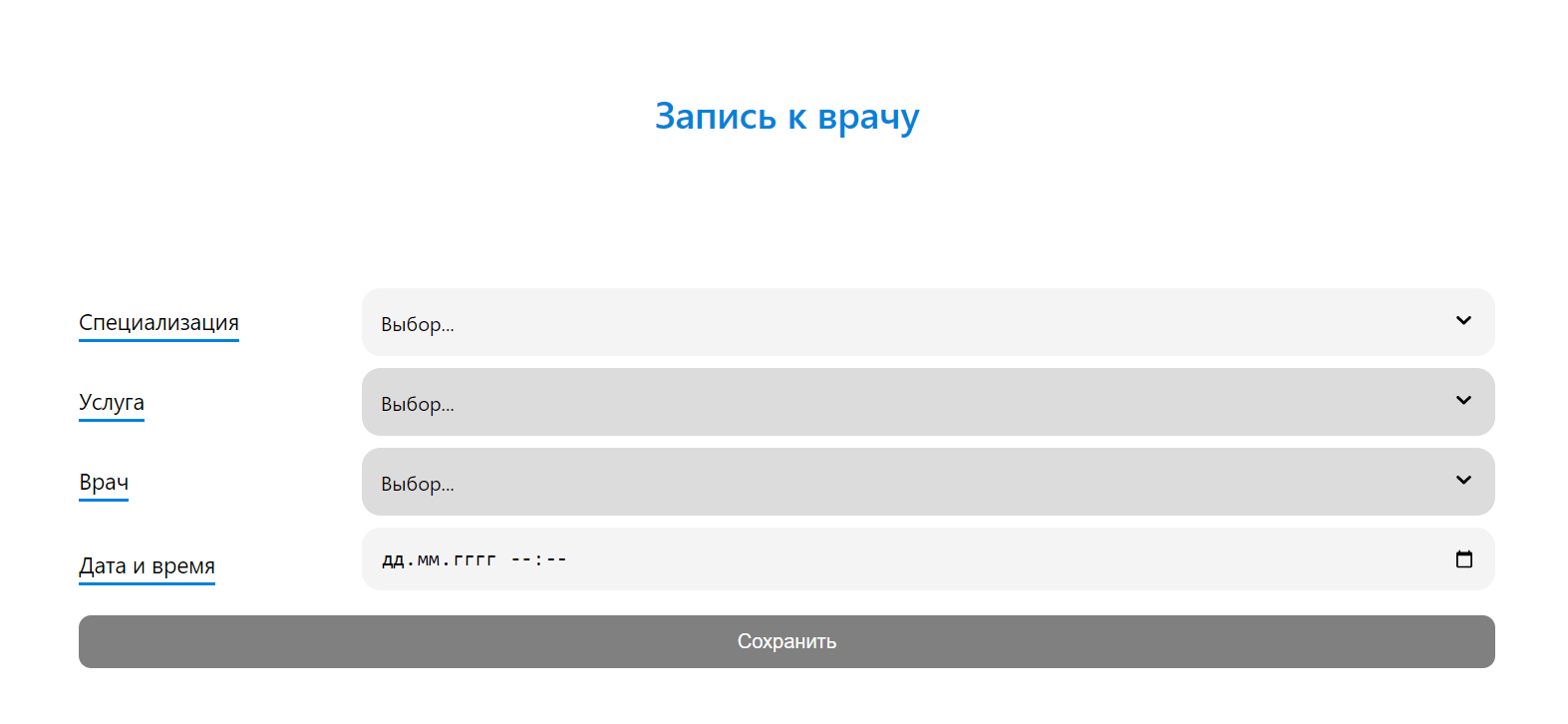
Приложение

Или посмотреть историю посещений.



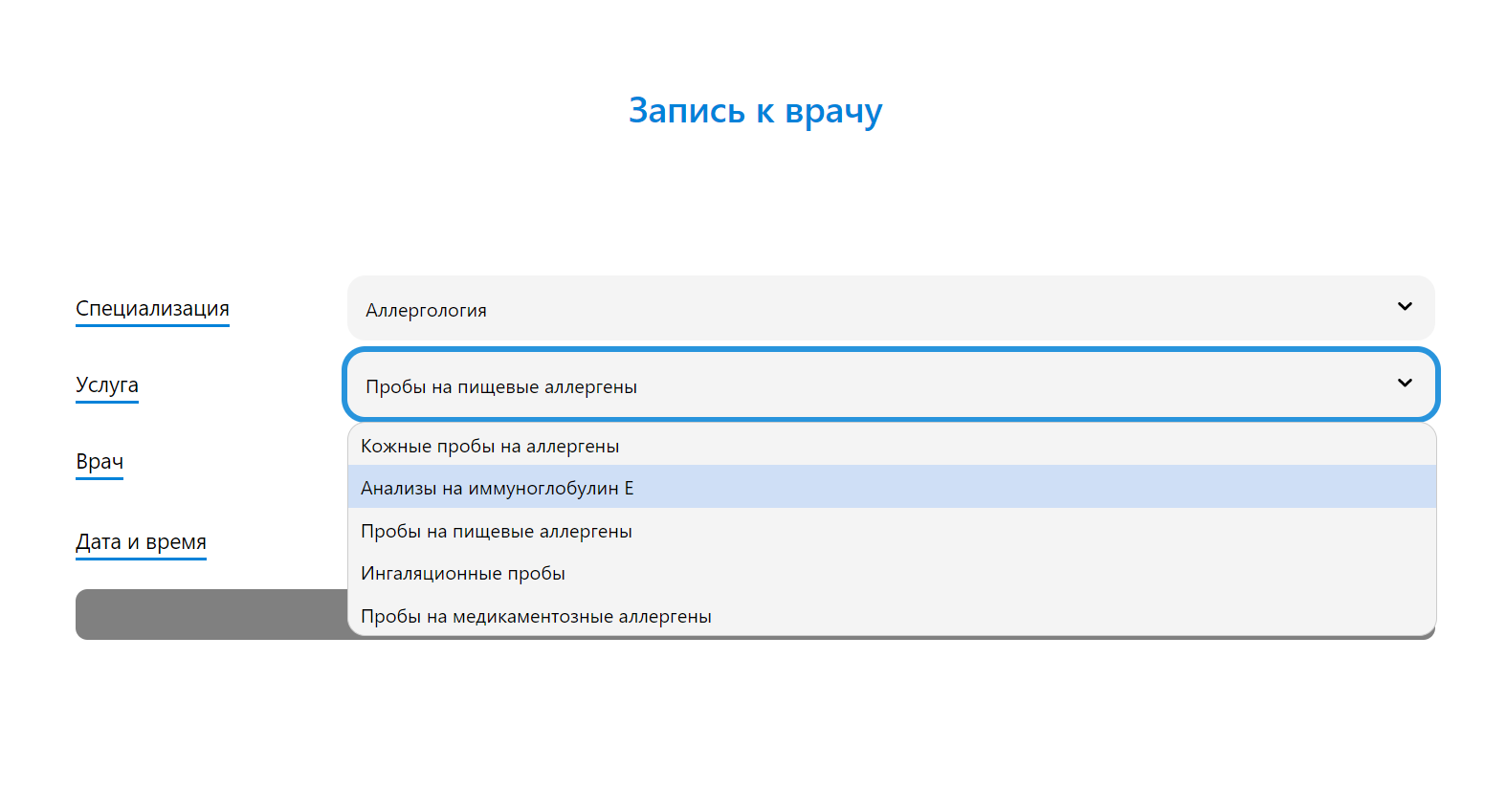
Приложение

При нажатии на кнопку «Записаться» пользователя отправит на страницу С формой для записи.

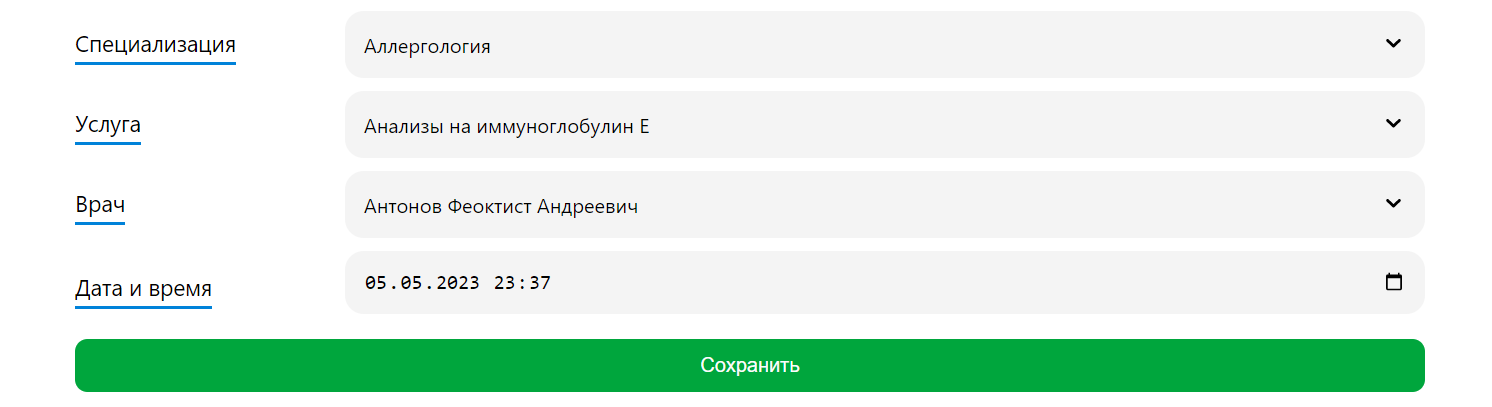


Приложение

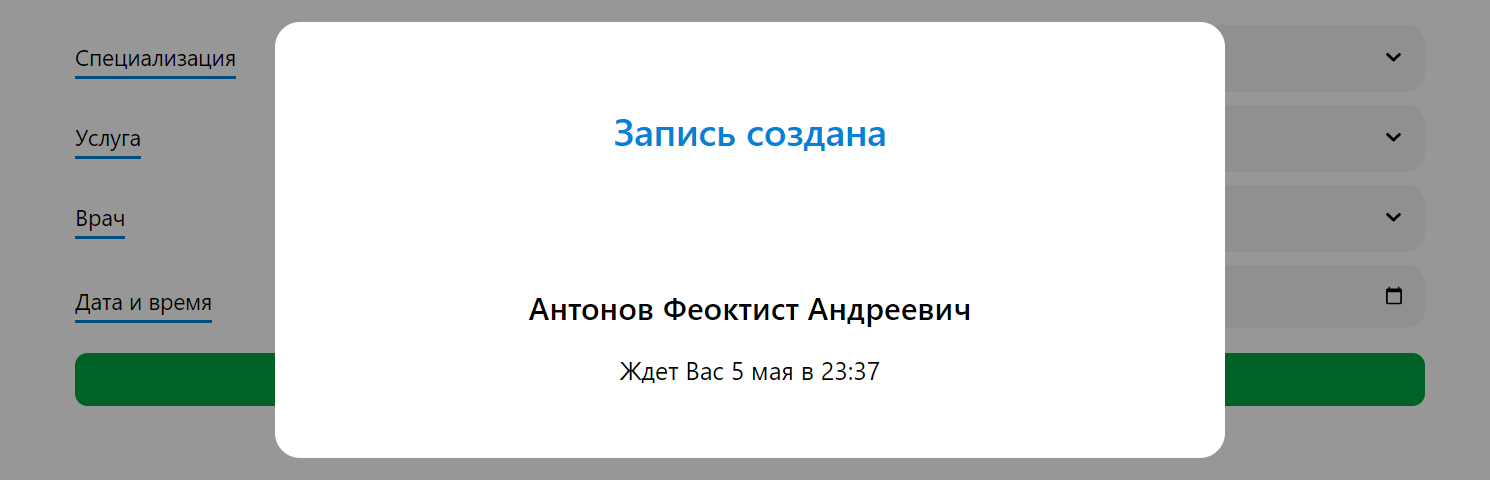
Выпадающие списки с услугами и врачами изначально недоступны, так как нужно сперва выбрать медицинское направление.



Приложение



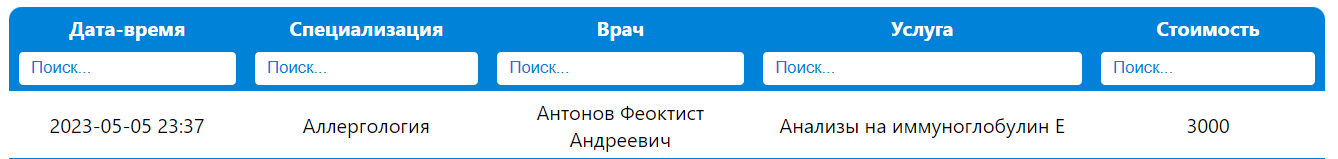
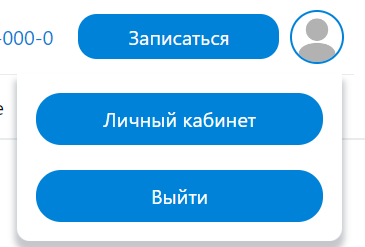
Приложение



Приложение

Далее можно перейти в личный кабинет и увидеть созданную запись в таблице.

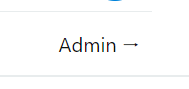
Приложение



Приложение 17

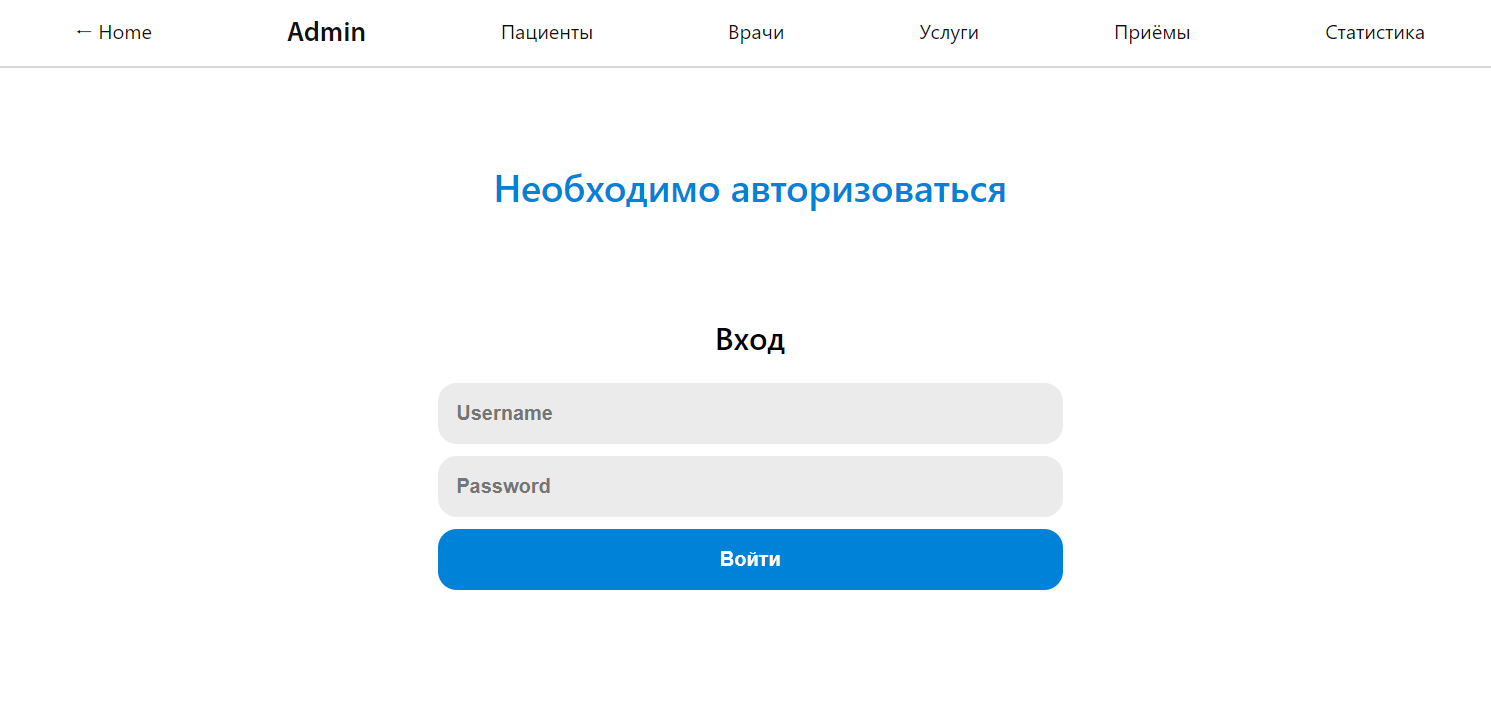
## Руководство для администратора

Для перехода в админ-панель надо нажать на кнопку Admin в шапке сайта.



Приложение

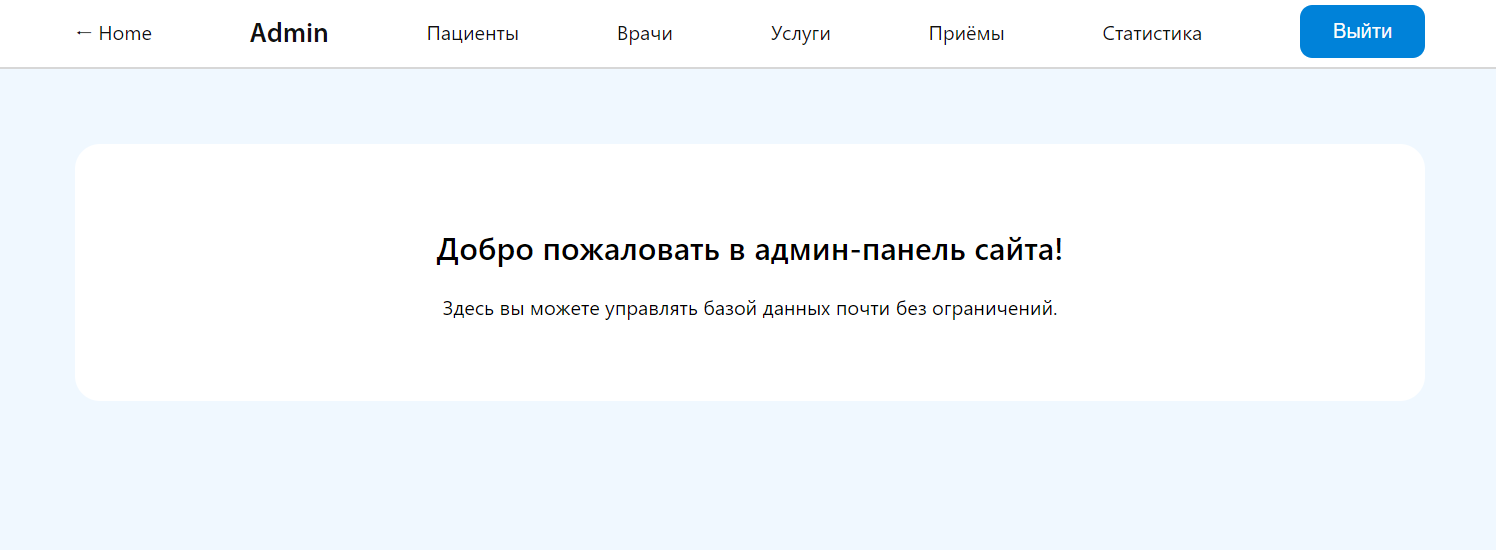
После этого откроется страница авторизации.



Приложение

Так как в базе данных есть только одна учетная запись администратора, вводим данные admin – admin.

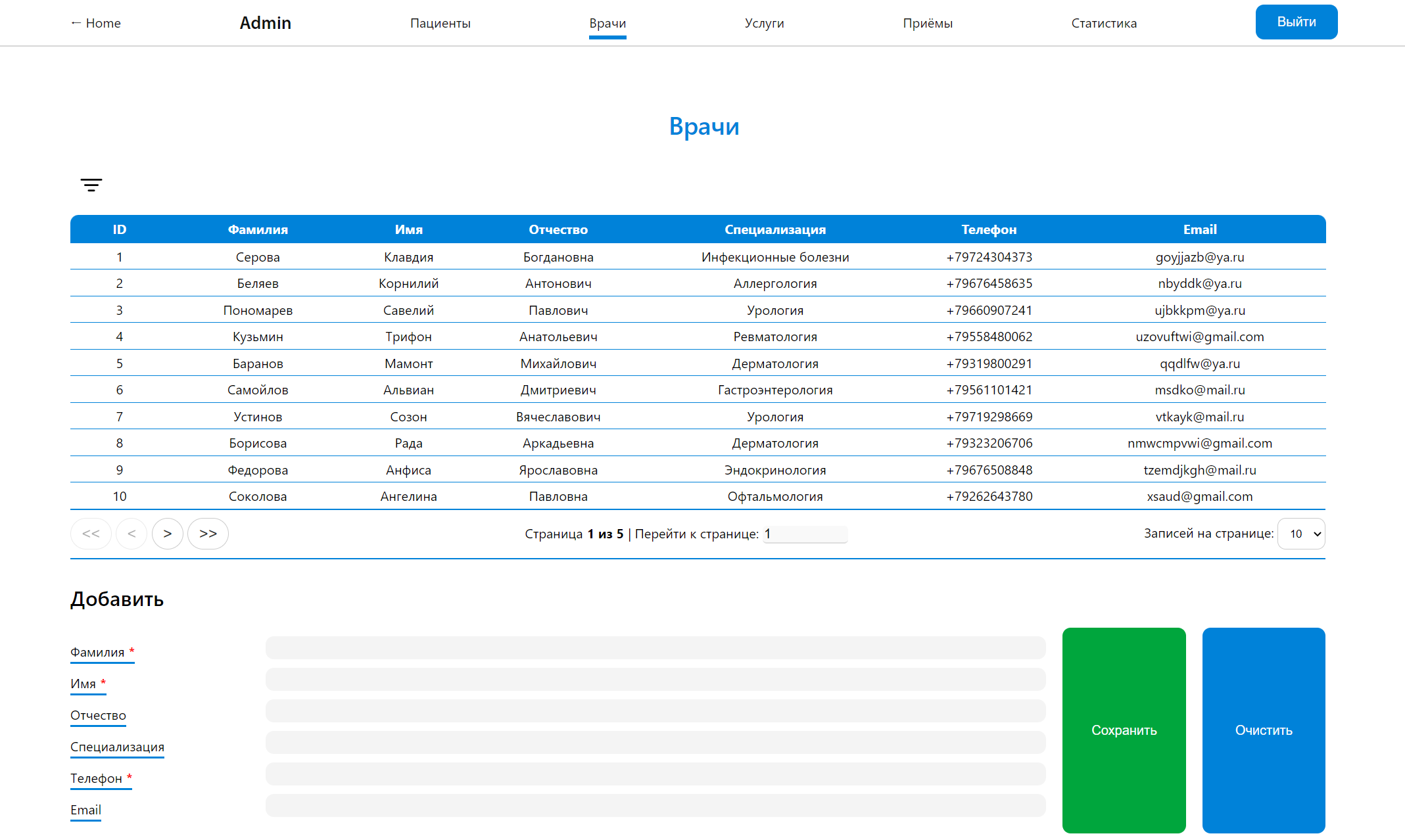
После входа пользователя перебросит либо на страницу приветствия, либо на другую страницу администратора, защищенную от доступа неавторизированным пользователям, с которой было выполнено перенаправление на страницу входа.



Приложение

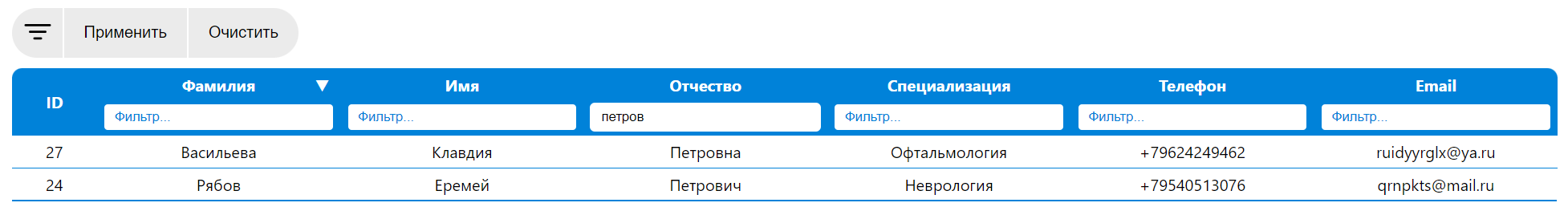
В шапке можно видеть все страницы, доступные администратору.

Страницы Пациенты, Врачи, Услуги, Приемы содержат в себе таблицы с соответствующими сущностями.



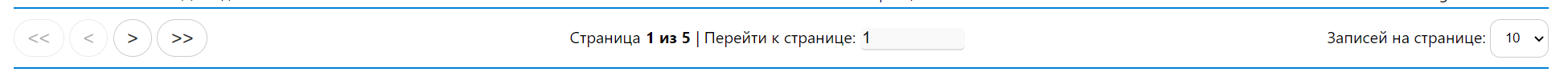
Приложение

Здесь можно посмотреть информацию об интересующей сущности, при этом пользователю доступны фильтры и сортировка. Для изменения параметров сортировки необходимо кликнуть по заголовку столбца таблицы, а для применения фильтров необходимо сперва их открыть, нажав на кнопку с фильтрами в верхней левой части таблицы, ввести необходимые данные и нажать Enter.



Приложение

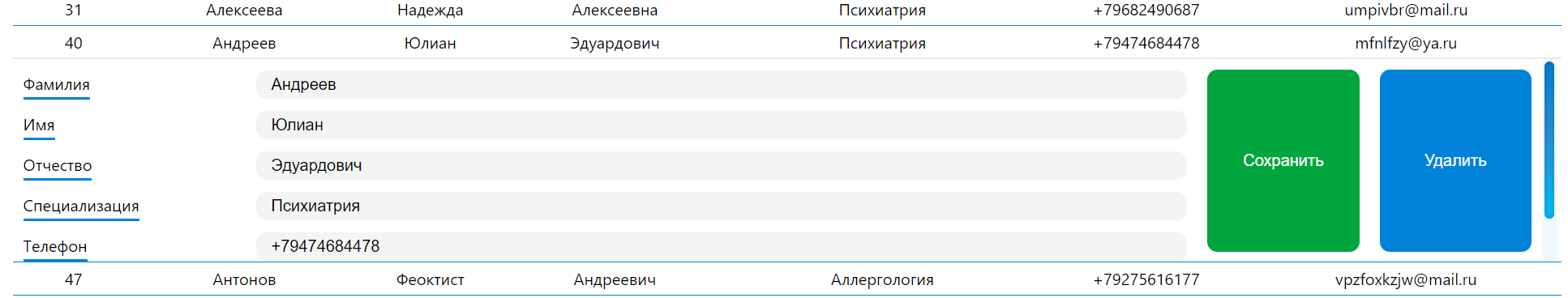
В нижней части таблицы есть элементы управления пагинацией.



Приложение

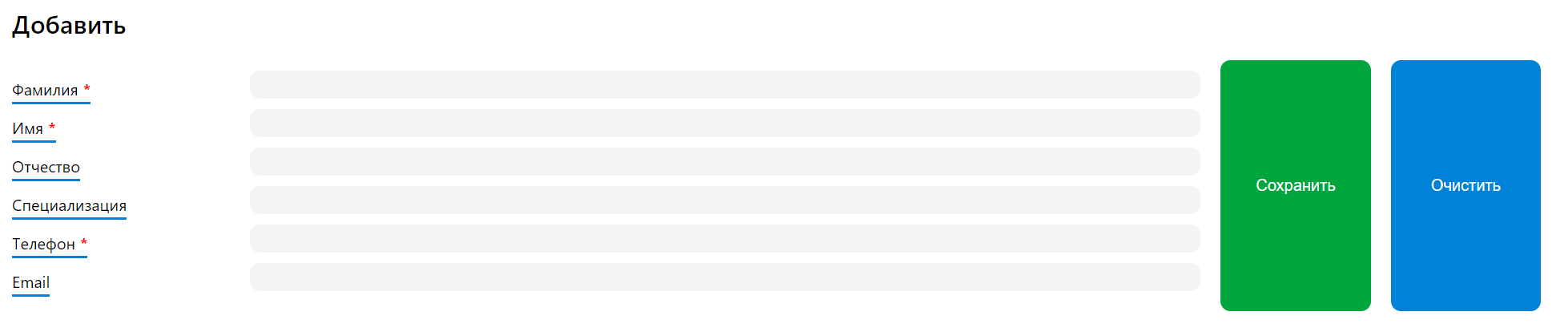
После каждого изменения в настройках пагинации, сортировки или фильтрации клиент отправляет запрос на сервер с новой конфигурацией и получает соответствующие данные. Это значит, что таблица не загружается на сайт целиком, что позволяет ускорить работу и не ждать долгую загрузку больших объемов данных.

Для изменения данных в записи необходимо кликнуть на соответствующую строку таблицы, после чего появится форма, где можно скорректировать данные, либо вовсе удалить запись.



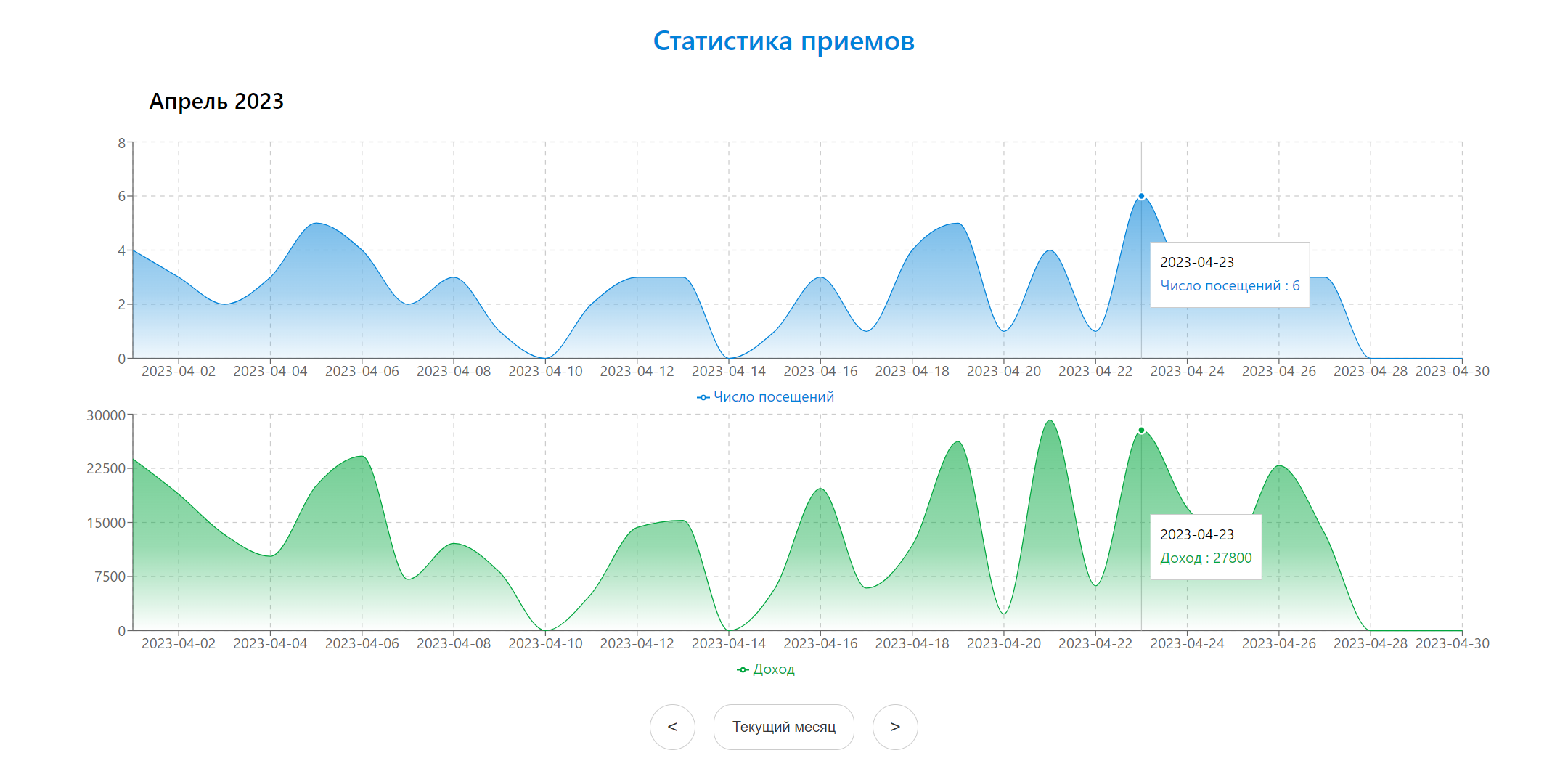
Приложение

Для создания новой записи необходимо воспользоваться формой, расположенной под таблицей.



Приложение

На странице Статистика администратору доступна информация о приемах врачей за определенный месяц.

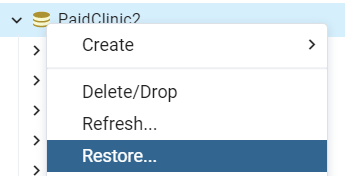


Приложение

Здесь можно увидеть графики посещаемости и суммарный доход от оказанных услуг. Для изменения периода просматриваемых данных можно воспользоваться элементами управления под графиками, после чего клиент отправит на сервер новый запрос с соответствующим сдвигом по месяцам относительно текущего и получит новые данные для отображения.

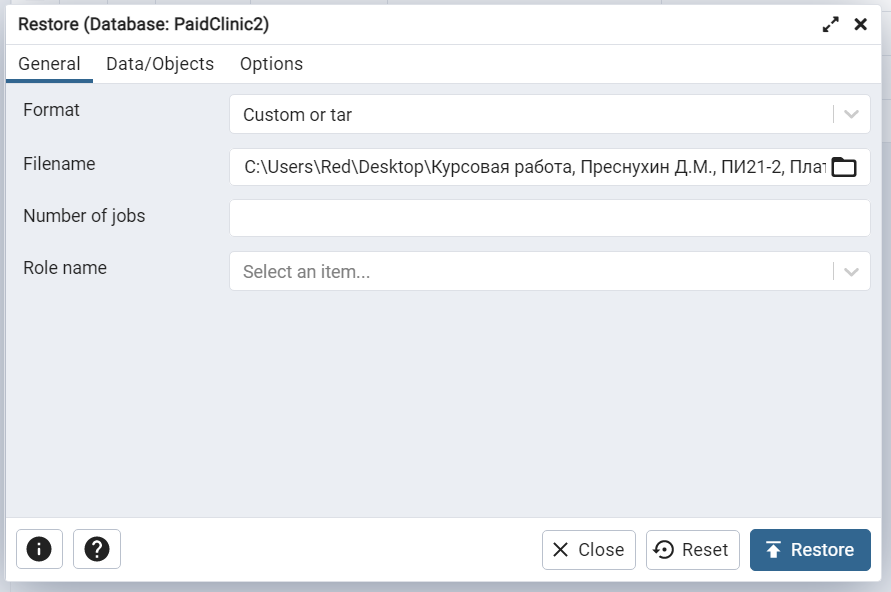
## Инструкция по запуску приложения

1. Создание базы данных
   1. Открыть pgAdmin
   2. Создать новую базу данных
   3. ПКМ по базе данных -> Restore... (Восстановть...)



Приложение

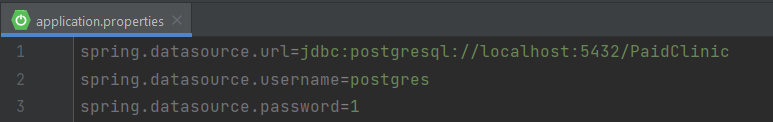
* 1. В поле Filename выбрать файл database.backup и нажать Restore



Приложение

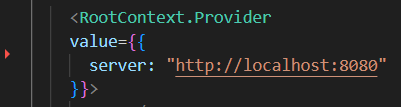
(аналогичные действия можно проделать при помощи любого другого менеджера баз данных Postgres)

1. Запуск сервера Spring
   1. Открыть папку проекта PaidClinic в Intellij IDEA или другой среде разработки
   2. В конфигурации aplication.properties прописать путь к базе данных и учетные данные



Приложение

1. Запуск сервера клиента
   1. Открыть файл clinic\_react\src\index.js и в значение контекста прописать адрес запущенного сервера на Spring



Приложение

* 1. В консоли (командной строке) перейти в папку с проектом clinic\_react
  2. Выполнить команду `npm install` для установки всех необходимых локальных библиотек (необходим установленный node.js)
  3. Дождаться установки зависимостей и выполнить команду `npm start`
  4. Если сайт не открылся автоматически, перейти по пути, указанному в консоли