Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное   
образовательное учреждение высшего образования   
«Самарский национальный исследовательский университет   
имени академика С.П. Королева»

Институт информатики и кибернетики

Кафедра технической кибернетики

Отчет по лабораторным работам

Дисциплина: «Технологии сетевого программирования»

Тема «Проектирование приложения»

Выполнили: Новикова В.В.,   
Мартынов О.И.

Группа: 6302-010302D

Самара, 2025

Содержание

[Введение 3](#_Toc198934690)

[1 Архитектура приложения 4](#_Toc198934691)

[1.1 Общая структура 4](#_Toc198934692)

[1.2 Компоненты 4](#_Toc198934693)

[1.3 Интеграция через docker-compose 6](#_Toc198934694)

[2 Описание API 7](#_Toc198934695)

[3 Тестирование API 8](#_Toc198934696)

[4 Аутентификация и авторизация 12](#_Toc198934697)

[4.1 Аутентифиакция 12](#_Toc198934698)

[4.2 Авторизация 12](#_Toc198934699)

[4.3 Безопасность 13](#_Toc198934700)

[5 Структура пользовательского интерфейса 14](#_Toc198934701)

[6 Описание структуры контейнеризации и настройки окружения 15](#_Toc198934702)

[6.1 Общая архитектура контейнеров 15](#_Toc198934703)

[6.2 Docker-compose 15](#_Toc198934704)

[6.3 Конфигурация Spring Boot 16](#_Toc198934705)

[7 Демонстрация работы контейнеров и приложения 17](#_Toc198934706)

[7.1 Запуск приложения 17](#_Toc198934707)

[7.2 Проверка работы контейнера 17](#_Toc198934708)

[7.3 Взаимодействие с приложением 17](#_Toc198934709)

[Приложение А 20](#_Toc198934710)

Введение

Beauty Salon Booking — это полнофункциональное веб-приложение, предназначенное для онлайн-записи клиентов на услуги в салоне красоты. Приложение реализовано на базе стека технологий Java 21 + Spring Boot + PostgreSQL с использованием server-side рендеринга через Thymeleaf. Вся система контейнеризирована с помощью Docker, что обеспечивает удобство развёртывания, масштабируемость и изоляцию окружения.

Проект разрабатывался как учебно-практический, с целью демонстрации навыков в области серверной разработки, проектирования структуры приложения, настройки безопасной аутентификации, работы с базой данных, а также интеграции компонентов в контейнеризированное окружение.

1. Архитектура приложения
   1. Общая структура

Приложение реализовано по классической трёхуровневой архитектуре и разделено на следующие ключевые компоненты:

1. Backend — серверная часть, построенная с использованием Spring Boot 3.4.2. Отвечает за обработку запросов, бизнес-логику, взаимодействие с базой данных и безопасность.
2. Frontend — реализован средствами server-side рендеринга через Thymeleaf. HTML-страницы генерируются на стороне сервера с последующей отправкой готового контента пользователю.
3. База данных — PostgreSQL, развернутая в отдельном Docker-контейнере. Используется для хранения информации о клиентах, мастерах, услугах, записях и JWT-токенах, отозванных при выходе из системы.

Компоненты взаимодействуют между собой в рамках одной Docker-сети и управляются через единый docker-compose.yml, что упрощает запуск приложения, настройку переменных окружения и обеспечивает полную изоляцию окружения.

* 1. Компоненты

1. Backend (Spring Boot)

Серверная часть реализована в виде монолитного Spring Boot-приложения.

Основные особенности:

* Используется Java 21 и Spring Boot 3.4.2;
* Контроллеры делятся на REST и Page-контроллеры:
  + REST-контроллеры обрабатывают JSON-запросы;
  + Page-контроллеры возвращают HTML-страницы (через Thymeleaf).
* JWT-аутентификация: токен создаётся при логине и хранится в cookie; реализована система logout через RevokedToken;
* Используется Spring Security для разграничения доступа по ролям (CLIENT, MASTER);
* Реализованы DTO-классы и слой сервисов для бизнес-логики;
* Взаимодействие с БД — через Spring Data JPA.

Функции backend включают:

* Регистрацию и авторизацию пользователей;
* Просмотр и редактирование профиля;
* Запись на услуги;
* Управление расписанием и статусами записей мастерами;
* Отзыв JWT-токенов при logout;
* Полную интеграцию с Thymeleaf-страницами.

2. Frontend (Thymeleaf + HTML/CSS)

Пользовательский интерфейс реализован с помощью Thymeleaf и классических HTML-шаблонов:

* HTML-страницы:
  + homepage.html — главная страница;
  + login.html, register.html — авторизация и регистрация;
  + dashboardClient.html, dashboardMaster.html — личные кабинеты;
  + appointment.html — запись к мастеру.
* Стили CSS оформлены в отдельных файлах, таких как styleDashboard.css, styleAuth.css, styleAppointment.css и др.

3. База данных (PostgreSQL)

В качестве основного хранилища данных используется PostgreSQL. Таблицы соответствуют основным сущностям системы:

* client, master — пользователи;
* beauty\_service — услуги;
* appointment — записи;
* revoked\_token — отозванные JWT-токены.

Миграции и создание схемы происходят автоматически благодаря spring.jpa.hibernate.ddl-auto=update.

* 1. Интеграция через docker-compose

Компоненты приложения объединены и управляются через docker-compose.yml.

Основные особенности:

* Контейнеры:
  + postgres — база данных;
  + backend — Spring Boot-приложение.
* Переменные окружения пробрасываются из docker-compose.yml в приложение;
* Используется depends\_on для правильного порядка запуска (сначала БД, затем backend);
* Общая Docker-сеть обеспечивает безопасное взаимодействие между компонентами;
* Файл Dockerfile-backend обеспечивает сборку backend-контейнера из jar-файла.

Преимущества такого подхода:

* Быстрый и воспроизводимый запуск на любой машине;
* Простое масштабирование и удобство сопровождения.

1. Описание API

В таблице 1 представлены основные эндпоинты API, которые соответствуют реализованному функционалу и могли бы использоваться во внешнем клиентском интерфейсе при наличии отдельного frontend-приложения.

Таблица 1 – Описание API

|  |  |  |
| --- | --- | --- |
| Метод | URL | Описание |
| POST | /api/auth/register/ master | Регистрация мастера |
| POST | /api/auth/register/client | Регистрация клиента |
| POST | /api/auth/login | Аутентификация пользователя |
| POST | /api/auth/logout | Выход пользователя из системы |
| POST | /api/masters/{masterId}/beauty-services/{beautyServiceId} | Присваивание услуги мастеру |
| POST | /api/masters/{masterId}/change-password | Изменение пароля у мастера |
| POST | /api/clients/{clientId}/change-password | Изменение пароля у клиента |
| POST | /api/appointments/create | Создание записи |
| GET | /api/masters/{masterId}/beauty-services | Получения списка всех услуг мастера |
| GET | /api/masters/{masterId}/available-time-slots | Получения свободных окон у мастера |
| GET | /api/masters/{masterId}/appointments | Получение всех записей мастера |
| GET | /api/clients/{clientId}/appointments | Получение всех записей клиента |
| GET | /api/beauty-services | Получение всех услуг |
| GET | /api/beauty-services/{beautyServiceId}/masters | Получение всех мастеров, оказывающих услугу |
| PATCH | /api/appointments/{appointmentId} | Изменение статуса у записи |
| PATCH | /api/masters/{masterId} | Обновление информации о мастере |
| PATCH | /api/clients/{clientId} | Обновление информации о клиенте |
| DELETE | /api/masters/{masterId}/beauty-services/{beautyServiceId} | Удаление услуги у мастера |

1. Тестирование API

Для проверки корректности работы серверной части веб-приложения проводилось ручное тестирование API с использованием инструмента Postman. Несмотря на то, что клиентский интерфейс реализован с помощью серверного рендеринга через Thymeleaf, каждый основной функционал был дополнительно проверен через вызовы соответствующих HTTP-запросов.

Процесс тестирования включал следующие аспекты:

* Проверка регистрации и авторизации пользователей (мастеров и клиентов);
* Создание, изменение и удаление записей на услуги;
* Получение списков доступных услуг, мастеров и записей;
* Обновление личной информации и смена пароля;
* Обработка ошибок (например, неправильный формат данных, отсутствие прав доступа и др.).

На каждом этапе проверялись:

* Корректность ответа (HTTP-статус, тело ответа, сообщения об ошибках);
* Логика обработки ролей (мастер/клиент);
* Защита маршрутов с использованием JWT-токенов.

Ниже приведены примеры запросов, выполненных в Postman:

* Регистрация клиента: POST /api/auth/register/client;
* Аутентификация: POST /api/auth/login с получением токена доступа;
* Регистрация клиента с занятым логином: POST /api/auth/register/client;
* Получение информации о записях клиента по защищенному эндпоинту: GET /api/clients/{clientId}/appointments с передачей токена в заголовке Authorization;
* Изменение пароля мастера по защищенному эндпоинту: POST /api/masters/{masterId}/change-password с передачей токена в заголовке Authorization;

Скриншоты успешных и неуспешных запросов приведены на рис. 1–5.

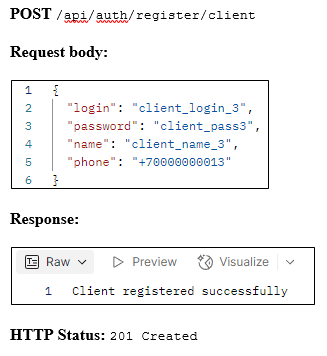


Рисунок 1 – Успешная регистрация клиента



Рисунок 2 – Успешная аутентификация клиента



Рисунок 3 – Ошибка при регистрации: логин уже занят

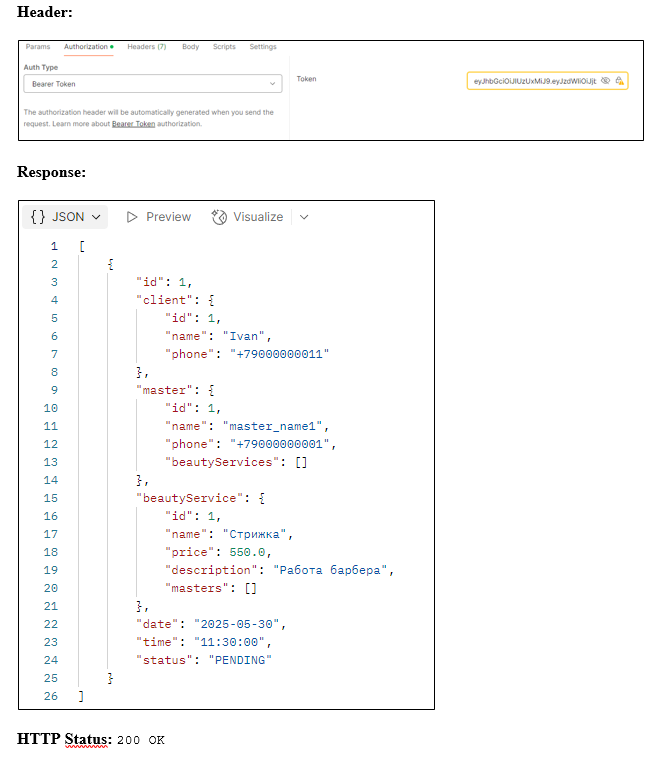


Рисунок 4 – Успешный доступ к защищённому эндпоинту

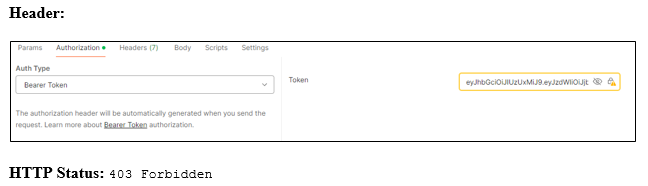


Рисунок 5 – Ошибка при изменении данных по защищенному эндпоинту

1. Аутентификация и авторизация

Веб-приложение использует механизм JWT-аутентификации (JSON Web Token) для управления доступом пользователей. Аутентификация и авторизация реализованы на стороне backend-приложения с использованием Spring Security.

* 1. Аутентифиакция

При входе в систему пользователь (клиент или мастер) отправляет запрос POST /api/auth/login, передавая свои учётные данные (логин и пароль). В случае успешной аутентификации сервер формирует JWT-токен, который возвращается в теле ответа. Этот токен представляет собой зашифрованную строку, содержащую информацию о пользователе (ID, роль и срок действия), и служит ключом доступа к защищённым ресурсам.

При последующих запросах клиент должен указывать токен в заголовке Authorization: Bearer <токен> или в cookies. Без валидного токена доступ к защищённым маршрутам будет запрещён.

* 1. Авторизация

Каждому пользователю при регистрации присваивается роль:

* CLIENT — клиент, имеющий доступ к просмотру услуг, созданию и редактированию своих записей;
* MASTER — мастер, который может управлять своими услугами, просматривать и редактировать записи клиентов.

На основе роли пользователя backend фильтрует доступ к маршрутам, используя аннотации и конфигурации Spring Security. Основная настройка осуществляется в классе SecurityConfig, где определяются правила доступа к API-эндпоинтам с использованием механизма URL-фильтрации и ограничения по ролям.

Пример:

* Только MASTER имеет доступ к маршруту обновления услуг мастера;
* Только CLIENT может создавать записи к мастерам;
* Любой пользователь может регистрироваться и аутентифицироваться.
  1. Безопасность

Для хранения паролей используется хэширование с BCrypt, что исключает хранение паролей в открытом виде в базе данных. Также реализованы следующие меры:

* Проверка уникальности login при регистрации;
* Валидация входных данных;
* Ограничение доступа к маршрутам с помощью ролей и токенов.

Такой подход обеспечивает надежную защиту данных пользователей и изоляцию функциональности между ролями в приложении.

1. Структура пользовательского интерфейса

Пользовательский интерфейс веб-приложения реализован с использованием Thymeleaf в рамках шаблонов Spring MVC. Интерфейс построен по классической MVC-модели и рендерится на стороне сервера, что обеспечивает простоту интеграции с Spring Boot и улучшает контроль над безопасностью.

Приложение разделено на логически обособленные пользовательские интерфейсы в зависимости от роли пользователя (гость, клиент, мастер). Основные страницы и компоненты:

* Регистрация и авторизация – отдельные формы для клиентов с соответствующей валидацией и отображением ошибок.
* Главная страница – доступна для всех аутентифицированных пользователей, содержит информацию о салоне и кнопку записи на услуги.
* Панель клиента – отображает список его записей, предоставляет возможность изменить пароль, редактировать личные данные, записаться к мастеру.
* Панель мастера – включает управление своими услугами, просмотр расписания, управление статусом записей, редактирование профиля.
* Страница записи на услугу – позволяет клиенту выбрать услугу и просмотреть доступных мастеров с возможностью записи.

Навигация между страницами обеспечивается через обычные переходы между маршрутами. Интерфейс адаптирован для базовой мобильной отзывчивости с использованием CSS (без использования фреймворков).

Такой подход обеспечивает простой, но функциональный пользовательский опыт, минимизируя зависимость от внешних библиотек и облегчая отладку в рамках серверного рендеринга.

1. Описание структуры контейнеризации и настройки окружения
   1. Общая архитектура контейнеров

Приложение полностью контейнеризировано с использованием Docker, что обеспечивает изоляцию компонентов, удобство развертывания и масштабируемость.

Архитектура включает два основных контейнера:

1. PostgreSQL — контейнер с официальным образом PostgreSQL 15, в котором развёрнута база данных приложения. Использует том ./data:/var/lib/postgresql/data для постоянного хранения данных между перезапусками.
2. Backend — контейнер с Spring Boot приложением, построенный на базе образа eclipse-temurin:21-jdk-alpine. Содержит собранный .jar файл, который запускается при старте контейнера.

Контейнеры объединены в одну виртуальную сеть beauty-network с типом bridge, что позволяет им безопасно взаимодействовать друг с другом внутри одной среды исполнения.

* 1. Docker-compose

Для управления контейнерами используется docker-compose.yml (рисунок 6), который описывает конфигурацию всех сервисов и их связи. Ключевые особенности:

* Переменные окружения используются для настройки доступа к базе данных (имя БД, пользователь, пароль), передаваемые как в контейнер PostgreSQL, так и в backend.
* Зависимости (depends\_on) гарантируют корректный порядок запуска: сначала база данных, затем backend.
* Проброс портов обеспечивает доступ к backend через порт 8080, а к PostgreSQL — через 5432.
* Сборка backend выполняется из локальной директории ./backend с использованием собственного Dockerfile-backend.

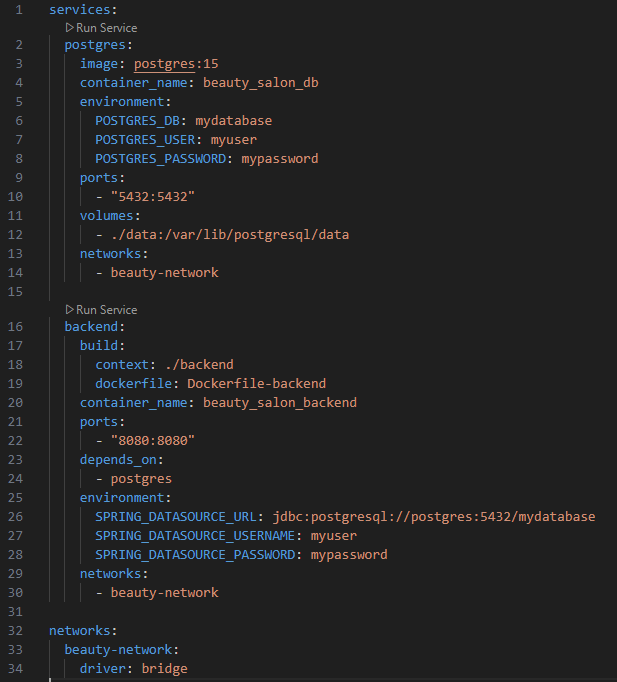


Рисунок 6 – Содержимое docker-compose.yml

* 1. Конфигурация Spring Boot

Файл application.properties (рисунок 7) настроен на чтение переменных окружения, передаваемых из Docker, что позволяет гибко управлять конфигурацией в зависимости от среды.

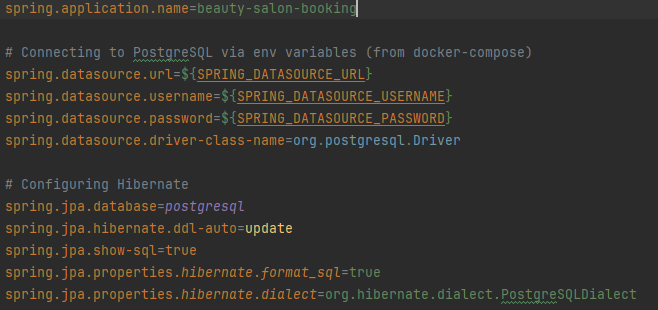


Рисунок 6 – Содержимое application.properties

Такой подход позволяет запускать всё приложение одной командой docker-compose up, без необходимости ручной настройки окружения.

1. Демонстрация работы контейнеров и приложения
   1. Запуск приложения

Для запуска всего приложения достаточно выполнить одну команду:

docker-compose up --build

После этого:

* Контейнер PostgreSQL разворачивается с заданными параметрами базы данных;
* Контейнер Backend автоматически подключается к базе и запускает Spring Boot приложение;
* Приложение становится доступно по адресу http://localhost:8080 (в случае использования frontend через Thymeleaf — интерфейс отображается сразу в браузере).
  1. Проверка работы контейнера

Состояние работающих контейнеров можно проверить командой:

docker ps

В результате отображаются оба запущенных сервиса:

* beauty\_salon\_db — база данных PostgreSQL;
* beauty\_salon\_backend — Spring Boot backend.
  1. Взаимодействие с приложением

Пользователь может перейти по адресу http://localhost:8080, где доступен web-интерфейс приложения, реализованный с использованием Thymeleaf. Через него можно:

* Зарегистрироваться как клиент;
* Просматривать услуги и доступных мастеров;
* Создавать и управлять записями;
* Изменять личную информацию и пароли;
* Входить и выходить из системы с различными ролями.

Также все доступные API-запросы могут быть протестированы через Postman, как было продемонстрировано в предыдущем разделе.

На рисунках 7-11 показана работа контейнеров и приложения.

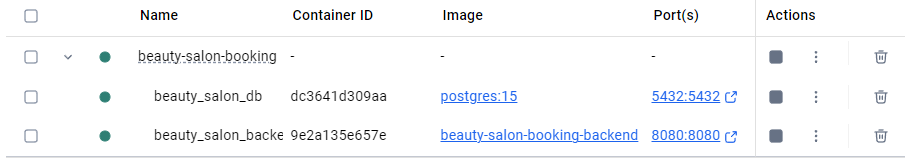


Рисунок 7 – Запущенные контейнеры

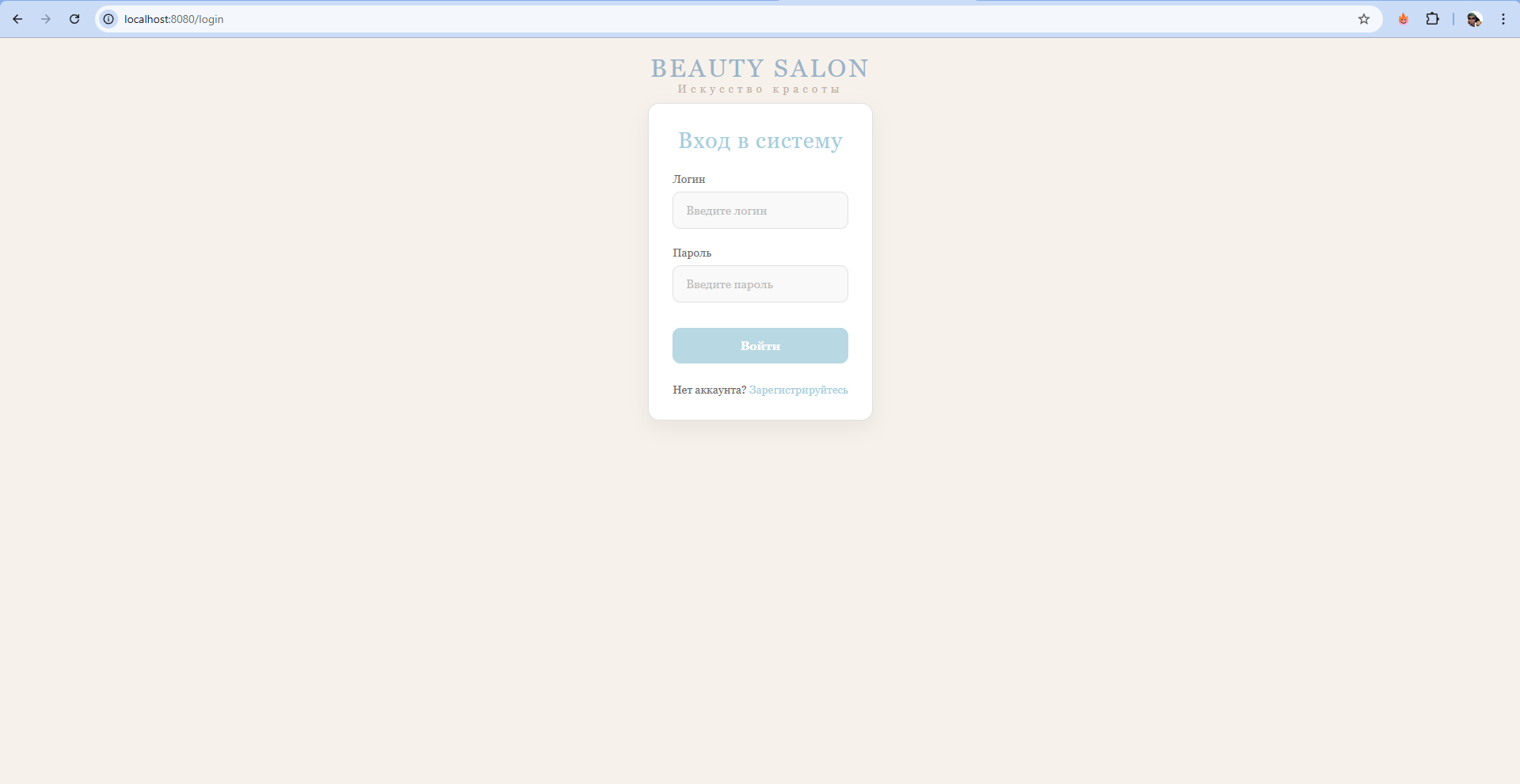


Рисунок 8 – Страница входа

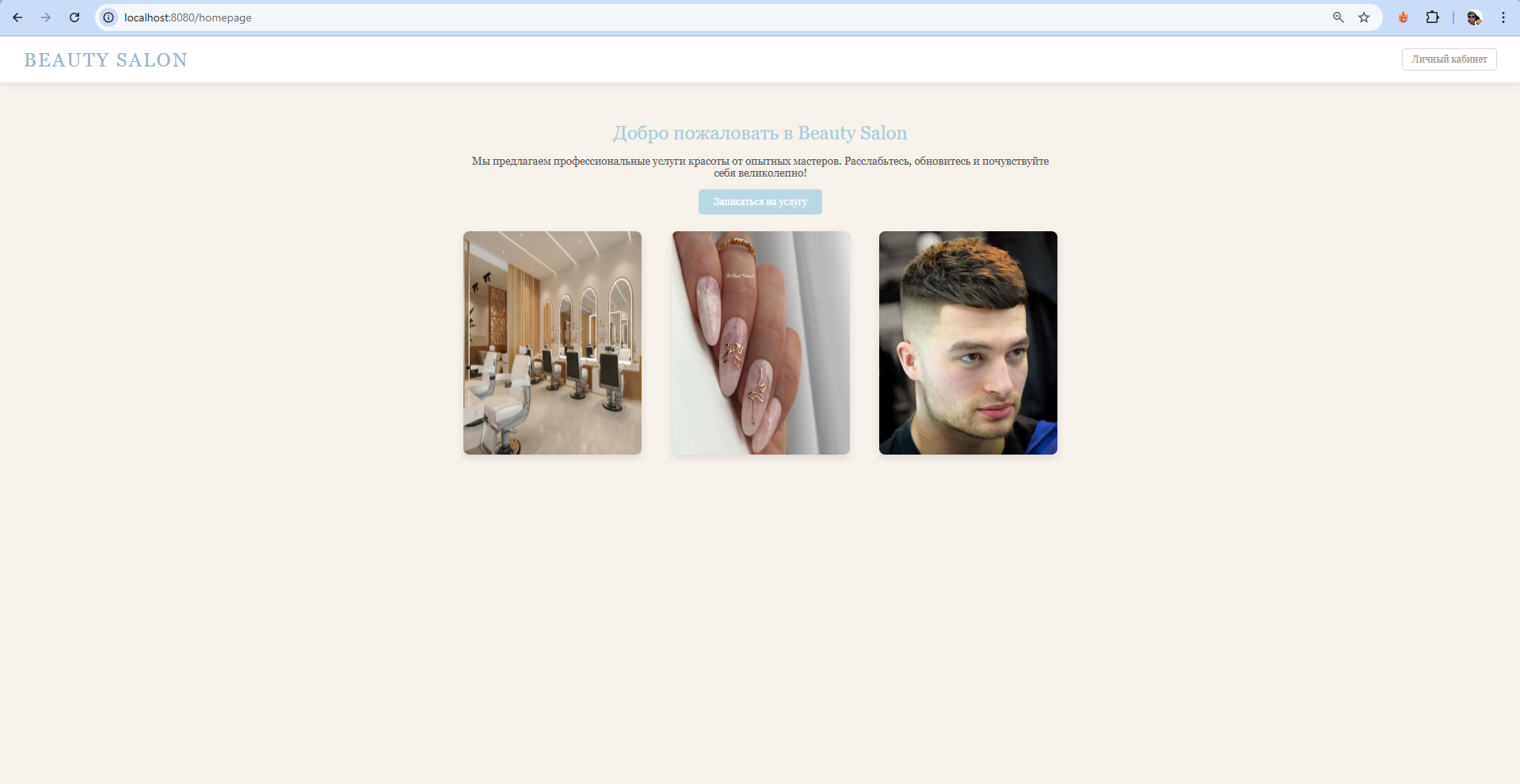


Рисунок 9 – Главная страница

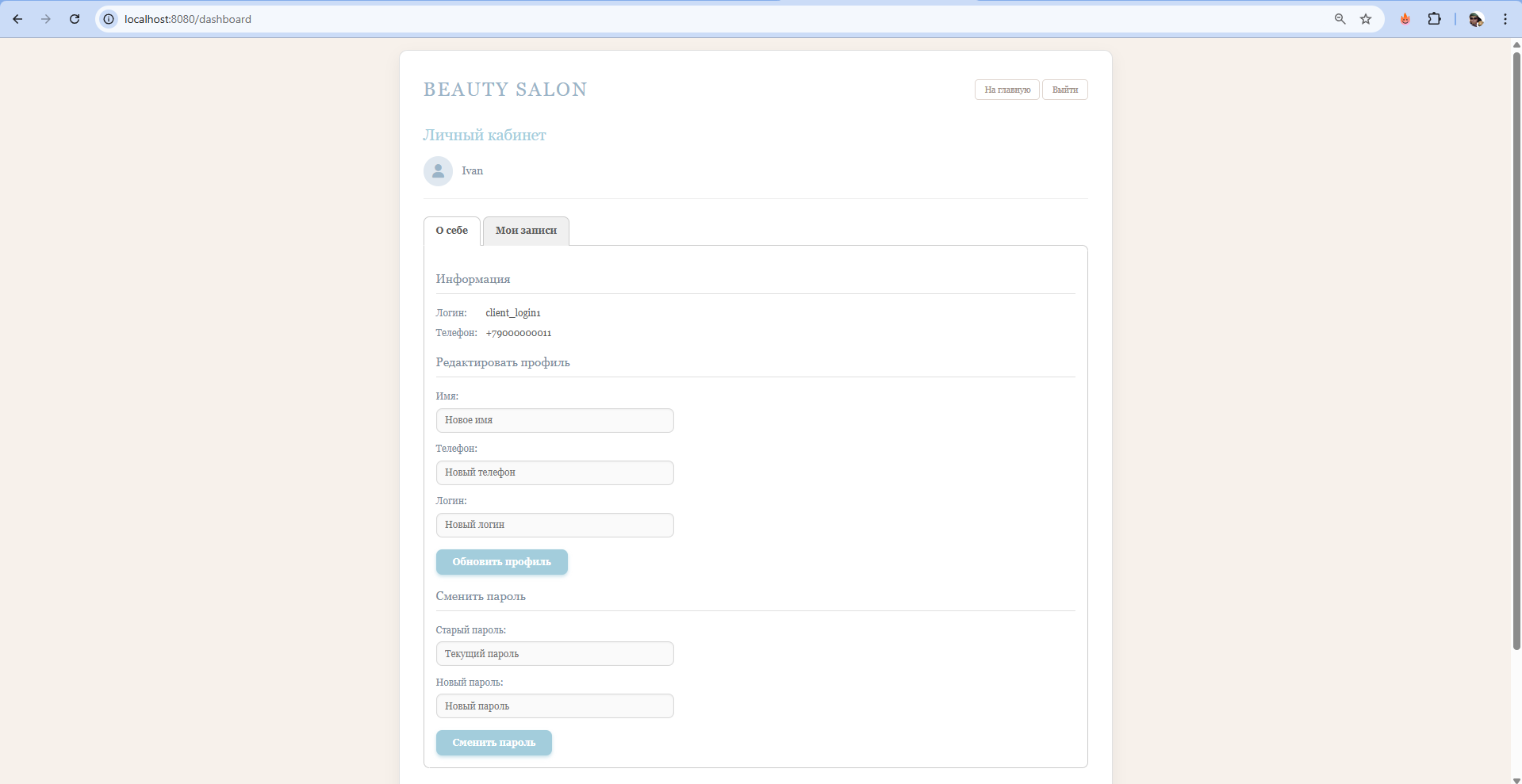


Рисунок 10 – Личный кабинет

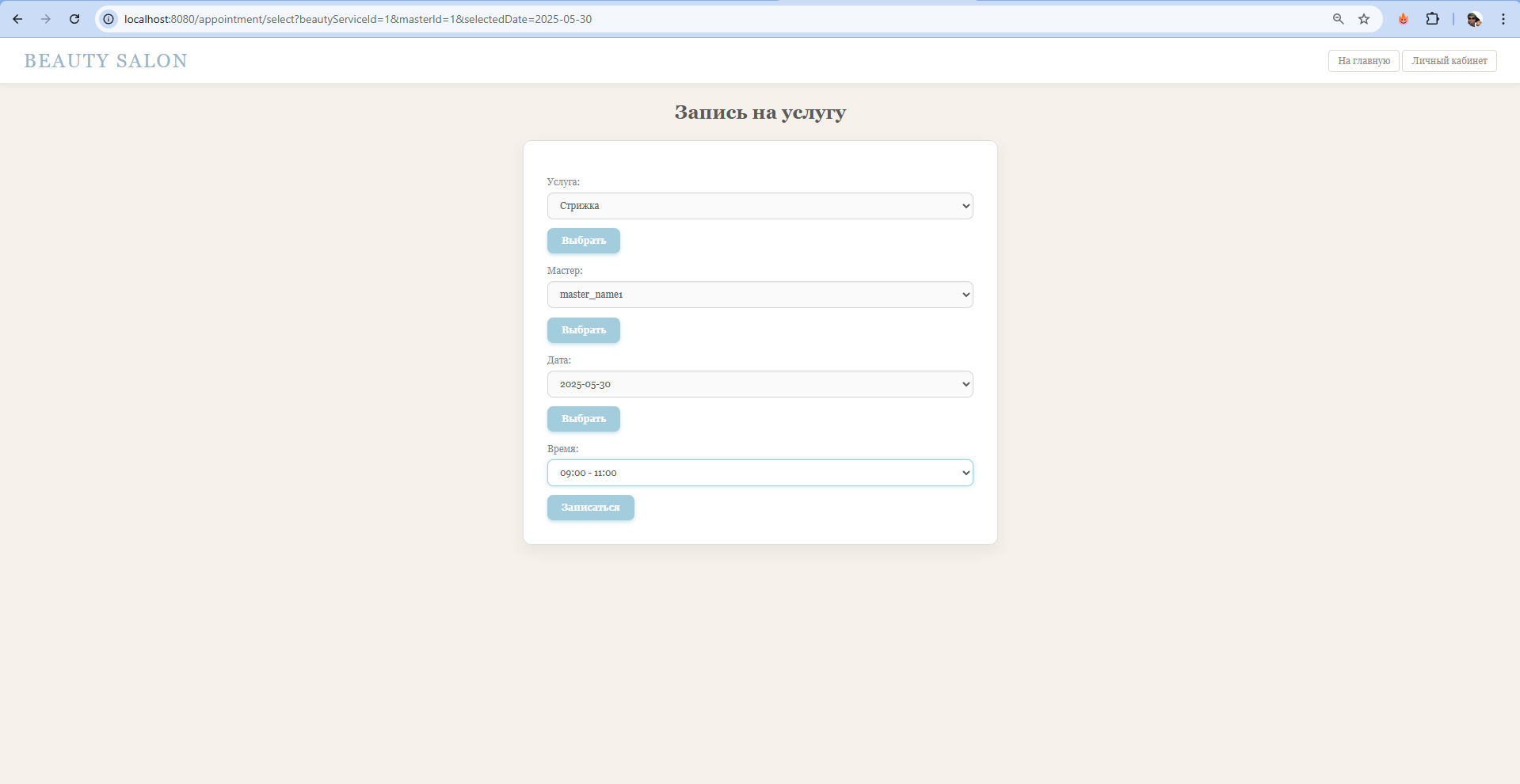


Рисунок 11 – Страница записи на услугу

Приложение А

Ссылка на GitHub со всем проектом:

<https://github.com/Olejjjka/beauty-salon-booking.git>