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

Финальный отчет

Дисциплина: «Технологии сетевого программирования»

Завершение проекта «Где находится картина?»

Выполнил:
Пчелинцев М.Д.
Группа: 6303-010302D

Самара 2025

Идея проекта «Где находится картина»

Где находится картина -- это веб-приложение, позволяющее отслеживать, в каком музее находится та или иная известная картина, а также выставляется ли она прямо сейчас.

Описание сущностей и ER-модель

1) **Картина.** Основной объект во всей структуре, так как цель данного проекта по сути накапливать данные о картинах. Необходимо создать гибкую систему описания картин, поэтому картины имеют уникальный идентификатор, а также могут иметь название, одного или нескольких авторов, год создания и прикрепленную оцифрованную копию. Картины могут создавать и редактировать только сотрудники институций (музеев), поэтому, чтобы редактировать информацию о той или иной картине необходимо либо быть сотрудником музея и иметь права на редактирование или самому создать эту картину (в рамках какого-то музея). Таким образом, картина непосредственно связана с институцией, которой она принадлежит. Кроме того, в случае, если картина в данный момент экспонируется, об этом указывается информация в отдельном поле.

2) Автор. Автор картины. Необходимо хранить информацию об авторе, поскольку это может облегчить поиск для пользователя.

3) Пользователь. Представляет собой простой аккаунт с логином и паролем. Для расширения этой функциональности добавляется возможность отмечать понравившиеся картины, чтобы следить за тем, экспонируются ли они где-либо. Пользователь системы может создать музей, тогда его аккаунт пользователя привяжется к этому музею правами редактировать картины в этом музее, а также давать другим пользователям права от имени этого музея.

3) Права. Необходимы для хранения данных о том, что может делать пользователь в рамках того или иного музея. У одного пользователя могут быть права в рамках нескольких музеев.

4) Музей. Записи о музеях создаются пользователями и содержат базовую информацию о них. Музей может быть подтверждённым. Это означает, что информация, предоставленная музеем веб-приложению скорее всего достоверна. Подтверждённость музея задается вручную администратором базы данных, поскольку полагается, что подтверждение музея требует тщательной проверки, выходящей за рамки онлайн-взаимодействия.

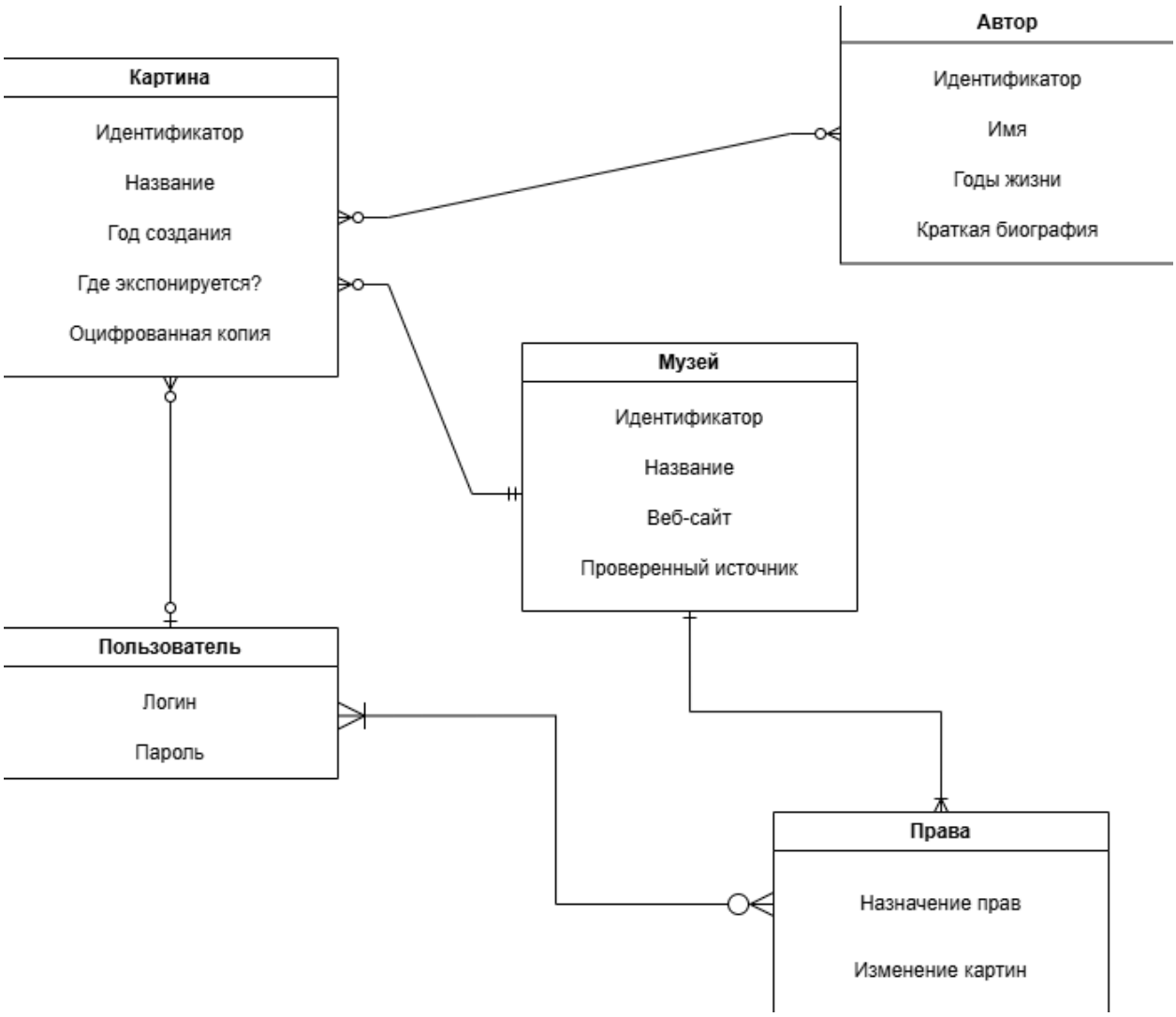

Рисунок 1 – ER-модель веб-приложения

Замечание: В ходе разработки проекта были внесены изменения в общую структуру сущностей проекта, поэтому в финальной версии некоторые объекты функционируют не так, как предполагалось изначально. Далее будут приведены данные изменения и их причины.

– связь автора с картиной стала 1 к 1, это было сделано, поскольку такое упрощение предметной области в целом может быть оправдано, в то время как использования связи 1ко многим в этом случае сильно усложнило бы множество запросов к базе данных, существенно замедлив их обработку.

– Права пользователей стали содержать лишь одно поле isAdmin. Это упростило обработку а также сделала систему более понятной. Админ может добавлять и удалять пользователей, менять им права, а пользователь без адинских прав может лишь менять картины. В реальном проекте это было бы сделано более проработано и существование прав администратора (а также возможно и других прав) было бы более оправдано. В учебном проекте это было сделано скорее в качестве демонстрации возможной реализации ролей.

Стек технологий:

Для реализации проекта был выбран следующий стек технологий:

Front-end: html + css + js (без библиотек)
Back-end: golang + gin + gin (router) + orm (gorm)
База данных: postgresql
Контейнеризация: docker

Структура REST API:

Основная структура REST API на период начала проекта доступна в репозитории проекта в файле README.md. Следует отметить, однако, что в дальнейшем эта структура изменилась (подробнее в разделе о второй контрольной точке).

Контрольная точка 1: структура БД и ORM.

Ключевым решением для меня при выполнении этой работы было начать разработку базы данных без ORM. Это решение отражено в репозитории проекта, а результат этой работы храниться в файле /no_orm/main.go. В тот момент постравил драйвер postgresql, настроил базу данных изнутри обычными sql запросами и запустил код, который должен был подключаться к базе данных. Преимуществом этого была простота и легковесность данного решения. Недостатки – отсутствие миграций, неопытность и пренебрежение к стандартному подходу привели меня к ошибкам. Одной из таких ошибок оказалось, например, использование sprint для формирования запросов в бд, что не безопасно.

Далее я выбрал ORM для дальнейшей работе. Выбор пал на GORM. Использование GORM дало мне миграции практически бесплатно, кроме того основные функции реализованные в GORM оказались довольно простыми в использовании. Также в GORM довольно интуитивно реализованы транзакции. Однако дальнейшая работа с GORM была не настолько удобной, как можно предположить. Одной из проблем является невозможность установки средствами GORM некоторых ограничений на отношения, полная неспособность GORM работать со сложными запросами. Кроме того, для получения большинства данных из БД не смотря на заданные изначально структуры, необходимо было создавать новые структуры внутри handler-ов. В результате, наиболее удобными и полезными функциями в GORM оказались .Raw(…) и .Exec(…), способные запрашивать и исполнять произвольный sql-запрос. Вывод, который я смог сделать из этого: в дальнейшем я не буду полагаться на ORM в структурировании своей программы и формирование запросов, однако использование подобных, возможно более простых инструментов, позволяющих удобно работать с транзакциями, корректно вставлять данные в SQL-запросы – очень продуктивно. Недостатки с которыми я при этом сталкивался: невозможность протестировать sql запрос непосредственно в бд, сложная отладка sql запросов – по сути снимаются. На финальном этапе работы с БД мне пришлось создавать довольно сложные sql-запросы, содержащие в себе несколько вспомогательных view. Работая над проектом, по сути все sql-запросы я писал inline в handler-ы. Ввиду большой изменчивости структуры этих handler-ов это было удачное решение.

Кроме этого была создана система для генерации картин и авторов. Основной целью этого было быстро предоставить тестовые данные в любом количестве, поэтому эта система не особенно производительна и устойчива к ошибкам, но такой они и планировалась, т.к. её не должно, по сути, быть в финальном проекте.

Было принято решение хранить изображения в обычной директории. Существуют, однако, и лучшие альтернативы – хранение в базе данных в виде сырых данных или хранение в объектной системе хранения. В случае моего проекта из двух альтернатив 1-я мне кажется лучше, однако, поскольку

проект учебный я решил не волноваться об этом. Эти тестовые скрипты доступны в main.go (addPaintingsIntoDB, addAuthorsIntoDB).

Контрольная точка 2: REST.

Для обработки REST запросов я воспользовался роутером gin. Структура задания типичного запроса в нем следующая:
Router.[GET]|[POST]|[PUT]|[DELETE]("/route:param", [middleware_func,] handler)

Это оказалось необычайно удобно. Gin позволял извлекать данные из json, uri, хранить переменные и проч.

В процессе разработки данного проекта я столкнулся с необходимостью обрабатывать больше запросов, чем планировал изначально. В этой связи main.go пришлось декомпозировать на несколько файлов. В основном (main.go) я продолжал хранить функциональные запросы, возвращающие json. Кроме этого я создал отдельные файлы для запросов в страницам (как правило, возвращают html) и тестовых запросов (/debug). Функции обрабатывающие запросы к страницами имеют постфикс _PAGE.

Далее приведены основные endpoint-ы, сопровождённые комментариями.

router.GET("/", getMain_PAGE) – получение главной страницы.
router.GET("/debug", get_DEBUG_PAGE) – получение страницы на которой тестировался front-end.
router.POST("/debug", post_DEBUG) – тестовый POST
router.GET("/success", success_PAGE) – получение страницы информирующей об успехе операции (на эту страницу выполняется redirect в js)
router.GET("/museum:museum_id/search_users_:request", searchNewUsers) – выдает список пользователей которые ещё не зарегистрированы в текущем музее. Список ограничен небольшим числом. Этого достаточно, поскольку пользователи имеют уникальные login-ы.
router.GET("/paintings_:request", searchPainting)
router.GET("/login_paintings_:request", requireAuth, searchPaintingsLoggedIn)
router.GET("/paintings_by_:request", searchByAuthor)
router.GET("/login_paintings_by_:request", requireAuth, searchByAuthorLoggedIn)
router.GET("/authors_:request", searchAuthor)

Пять запросов представленных выше позволяют пользователю искать картины (по названию и по автору) в двух состояниях: login и logout. Разница заключается в том, что при получении списка картин при login необходимо также отображать нравится ли данная картина пользователю или нет.

router.POST("/login", login)
router.POST("/register", register) –

Два запроса предназначенные для регистрации и аутентификации пользователя в системе. Выход пользователя из системы реализован на уровне front-end-а.

router.GET("/register_author", requireAuthPAGE, getRegisterAuthor_PAGE)
router.POST("/register_author", requireAuth, postRegisterAuthor)
router.GET("/register_museum", requireAuthPAGE, getRegisterMuseum_PAGE)
router.POST("/register_museum", requireAuth, postRegisterMuseum)
router.GET("/museum:museum_id/register_painting", requireAuthPAGE, registerPainting_PAGE)
router.POST("/museum:museum_id/register_painting", requireAuth, postPainting)

Представленные выше 6 запросов выдают страницы с формами для регистрации музеев, авторов, картин, а также принимают данные с этих форм.

router.GET("/museum:museum_id", requireAuthPAGE, getMuseum_PAGE) – выдает страницу музея. Интересно, что страница музея рендерится на back-end при помощи щаблонизатора go – страница рендерится по разному для пользователей с разными правами. В целом этот подход лучше, чем тот, что был избран с главной страницей, где она «дорисовывается» в результате запросов со стороны front-end.

router.GET("/museum:museum_id/paintings_:request/page:page_id", getMuseumPaintings) – возвращает картины по определенному запросу. Важно отметить, что есть картин много на сайте появляется кнопка «больше», нажимая на которую загружается следующая страница со списком картин. В данном запросе это отмечено параметром :page_id.

router.POST("/museum:museum_id/rights", requireAuth, postUserRights)
router.PUT("/museum:museum_id/rights", requireAuth, changeUserRights)
router.DELETE("/museum:museum_id/rights", requireAuth, deleteUserRights)

Три представленных выше запроса позволяют давать, изменять и удалять права пользователей в музее. Выдавая права пользователю они по умолчанию делают его обычным членом музея (НЕ админом), права на администрирование можно выдать или забрать через PUT. DELETE полностью удаляет пользователя из списка сотрудников музея.

router.PUT("/painting:painting_id/change_painting", requireAuth, changePainting)
router.DELETE("/painting:painting_id/delete_painting", requireAuth, deletePainting)

Из представленных выше запросов в финальном веб-приложении используется только DELETE (удаляющий картину). Было решено, что изменение картины нецелесообразно усложнит функциональность front-end-части (отмечу в скобках, что это происходит потому, что для изменения той или иной картины необходимо создавать отдельную страницу (не форму), где возможно было бы динамически отслеживать и применять изменения (по крайней мере это то, что я хотел реализовать изначально). Скорее всего для

этого потребовалось бы несколько вспомогательных handler-ов. Я решил, что проще на данном этапе обойтись без этой функциональности и пользоваться комбинацией создания и удаления. Отмечу также, что в реальном музее подобная функция была бы основной, а добавление и удаление картин нужно было бы реализовать пакетами, а не по одной картине, возможно через загрузку файла определенного формата. Данный комментарий я написал, чтобы показать некоторую перспективу, которую мог бы взять проект при настоящей реализации).

router.GET("/favorite", requireAuth, getFavorites)
router.POST("/favorite", requireAuth, postFavorite)
router.DELETE("/favorite", requireAuth, deleteFavorite)

Эти три запроса позволяют пользователям добавлять и удалять любимые картины. GET запрос возвращает список любимых картин пользователя, который необходимо отображать в его личном кабинете.

router.GET("/login_info", requireAuth, getLoginInfo) – данный запрос используется для заполнения главной страницей информацией авторизованного пользователя. По сути результатом запроса является список всего, что нужно отобразать в кабинете в первую очередь. Такой запрос призван объединить в один пакет информацию, которая как правило приходит на front-end вместе. Вынужден признать, что в этом смысле архитектура front-end-а страницы музея (с использованием шаблонизатора) намного удачнее, но я решил оставить это решение тоже и в учебных целях довести его до конца, чтобы проанализировать преимущества и недостатки обоих подходов.

Для тестирования end-point-ов я использовал как просто браузер, так (в случае с json-ответами) и Postman. На рисунке 2 представлен список запросов в Postman.

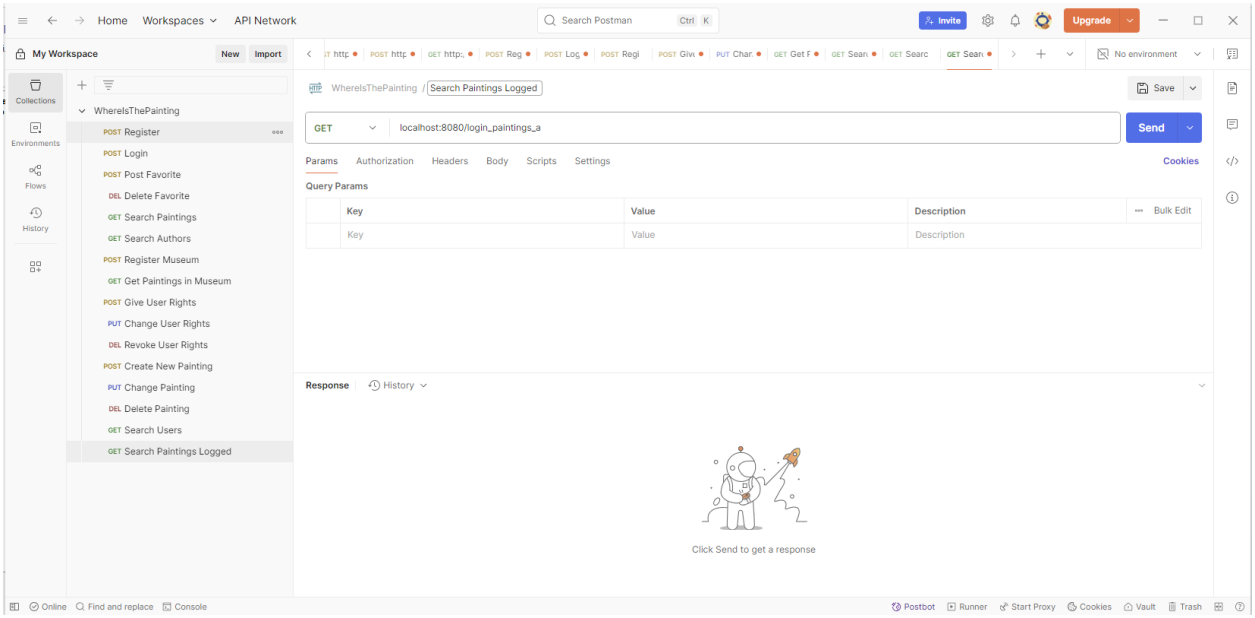

Рисунок 2 – Список запросов в Postman

Контрольная точка 3: Авторизация

В работе используется авторизация через JWT-токены, для этого был подключена библиотека golang-jwt (классическая для golang web-разработки). В целом выдача токенов и их проверка были сделаны максимально классическим образом: при аутентификации пользователю выдается jwt-токен, который передается для записи в cookie. Доступ к токену через js на front-end-е ограничен. Далее все запросы передаются на сервер вместе с токеном. Midddleware проверяет токены и в случае отказа доступа возвращает 401 unauthorized. Важно отметить, что реализовано 2 версии middleware, одна – отвечающая на запросы, которые должны выдавать json и другая – для html-страниц. Html – версия кроме прочего возвращает пользователю страницу неавторизован. Json-версия требует проверки со стороны front-end-а. Время жизни одного jwt-токена – один час. По истечению времени пользователю необходимо перезайти.
Механизм выхода из сессии (logout) реализован на стороне front-end-а: при успешном получении jwt-токена front-end регистрирует этот факт в local storage. При нажатии на кнопку выйти front-end чистит страницы от данных предыдущего пользователя и запоминает, что он не авторизован, при этом jwt-токен не стирается. При следующей успешной аутентификации jwt токен перезаписывается и система снова понимает, что пользователь авторизован. Данный вариант реализации авторизации не идеален. Во-первых, существование одного access-токена – не самый безопасный вариант для реализации. Можно было бы ввести и access и refresh токены. Второе улучшение, которое можно было бы предоставить – использование отдельного отношения, хранящего действительные на данный момент токены.

Далее на рисунке 3 продемонстрировано поведение системы при попытке получить доступ к защищенному end-point-у без авторизации. На рисунке 4 показан результат успешной аутентификации.

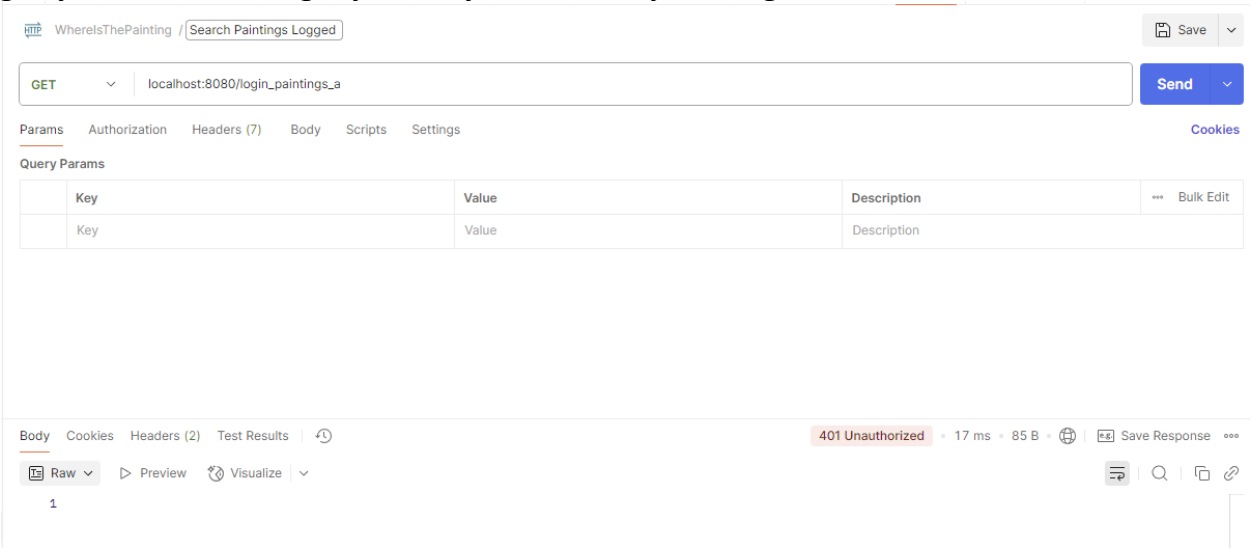

Рисунок 3 – Попытке неавторизованного пользователя получить доступ к защищенному end-point-у

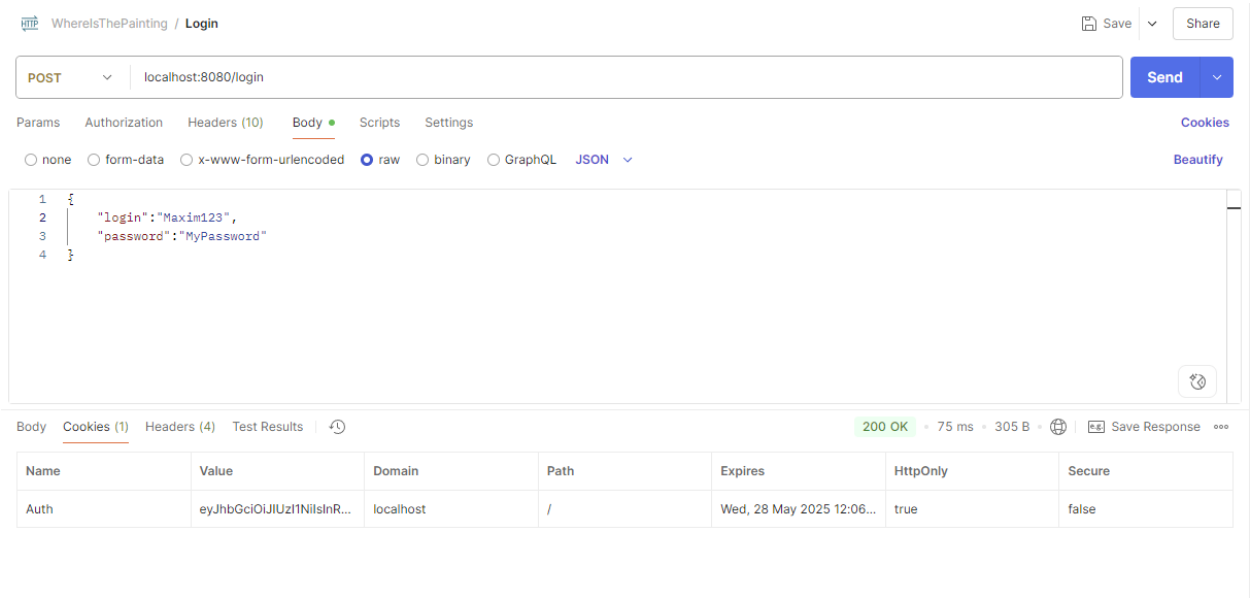

Рисунок 4 – Результат успешной аутентификации пользователя

Контрольная точка 4: front-end.

Front-end реализован на чистом js. Запросы отправляются на сервер при помощи fetch. В нескольких местах на сайте реализованы выпадающие списки. Они обновляют значения, посылая fetch запросы на сервер при изменении поля input. Многие обработчики в js изменяют данные в нескольких местах на сайте, например, при нажатии кнопки «мне нравится» картинка отмечается как понравившаяся, а также появляется в списке сбоку (в личном кабинете). Для удобства на сайте используется monospace-шрифт, поскольку это позволяет точно отслеживать длину поисковых запросов. Во всех результатах на запросы выделяется запрашиваемая подстрока. Ещё на back-end, на уровне бд запросы сортируются в порядке вхождения данной подстроки.

Напомню, что часть страниц рендерится ещё на сервере (страница музея), в то время как часть – рендерится при загрузке при помощи дополнительных fetch запросов из js (главная страница). В процессе рендеринга из шаблона страницы музея проверяется является ли запросивший страницу пользователь администратором и в зависимости от этого рендеринг осуществляется по разному. Многие динамические изменения страницы реализованы через изменение innerHTML/outerHTML некоторых компонентов. Также отсылаю к контрольной точке 3, где описывается реализация поведения системы при logout, реализованного на front-end.

Далее на рисунках 5 – 23 представлена визуальная часть реализации front-end с демонстрацией функционала.

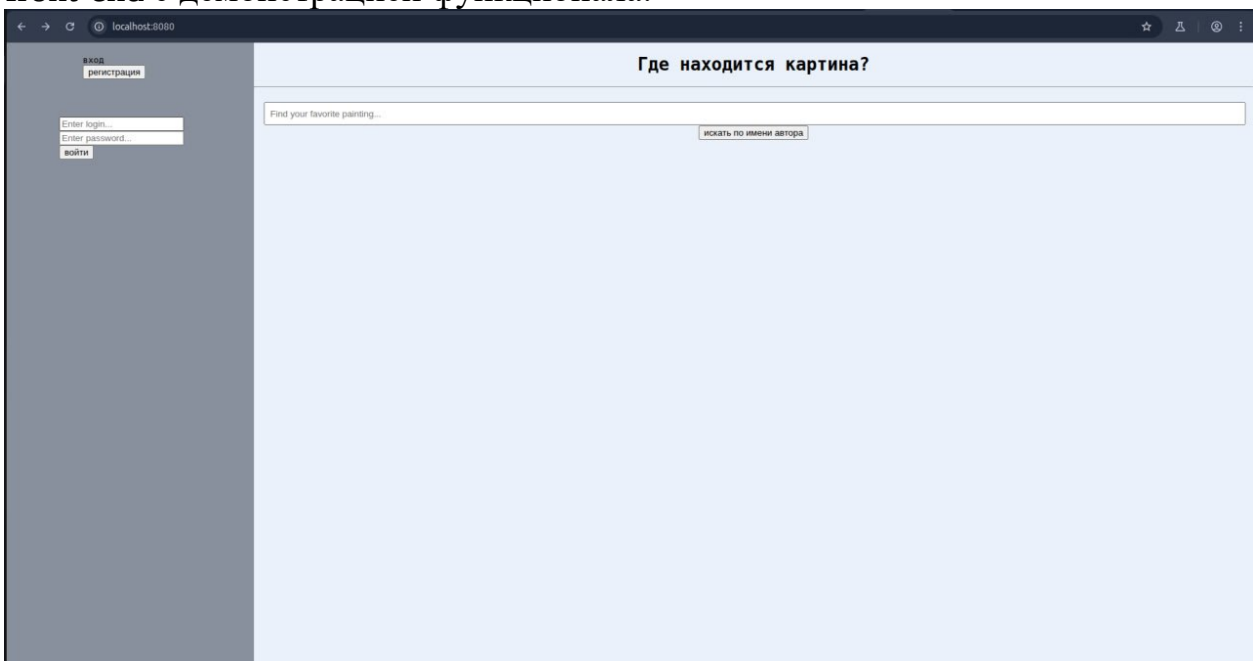

Рисунок 5 – Главная страница сайта для только зашедшего неавторизованного пользователя

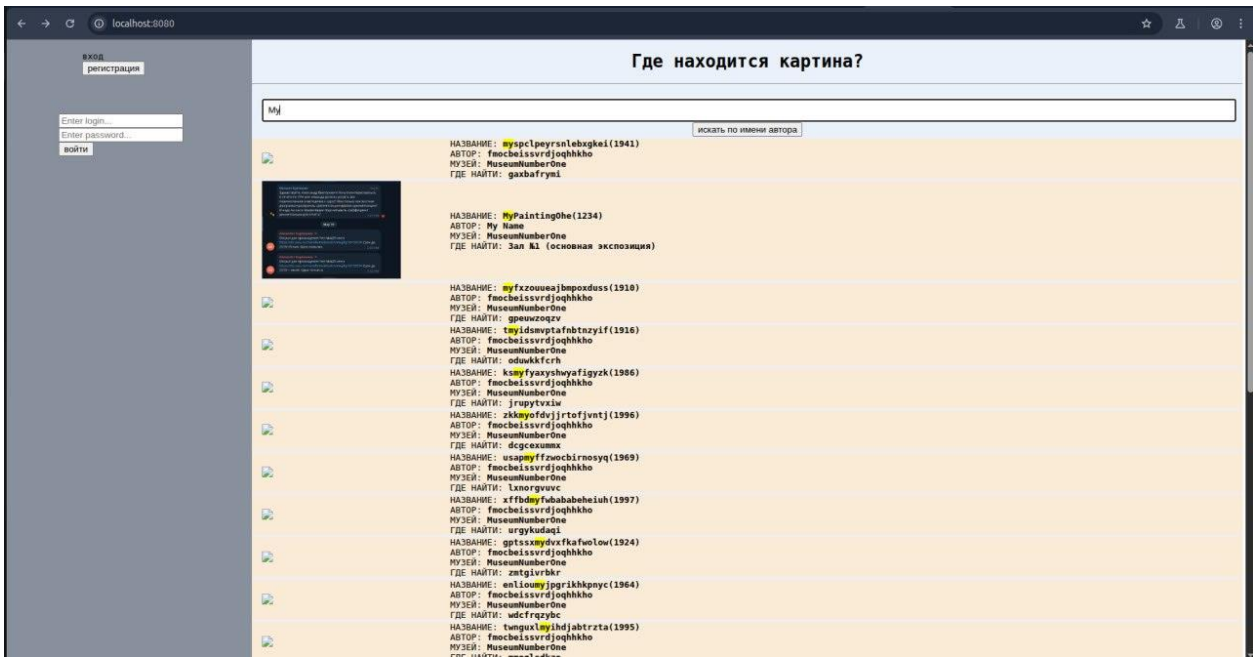

Рисунок 6 – Неавторизованный пользователь пытается найти картину по названию

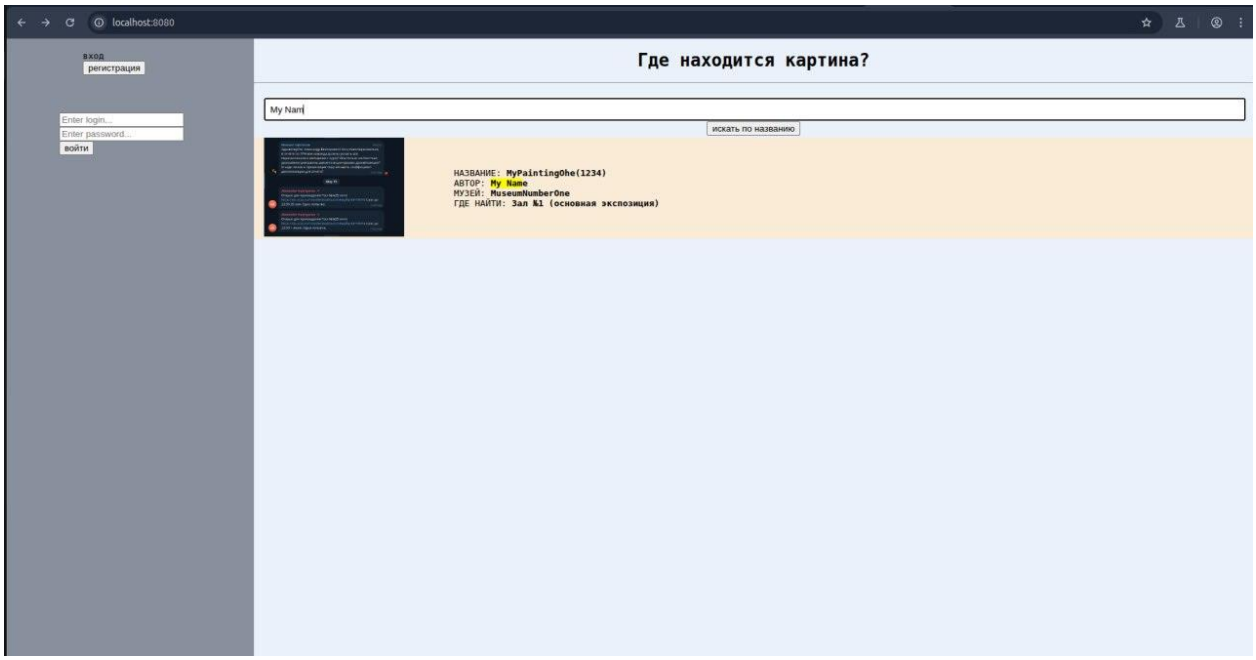

Рисунок 7 – Неавторизованный пользователь ищет по имени автора

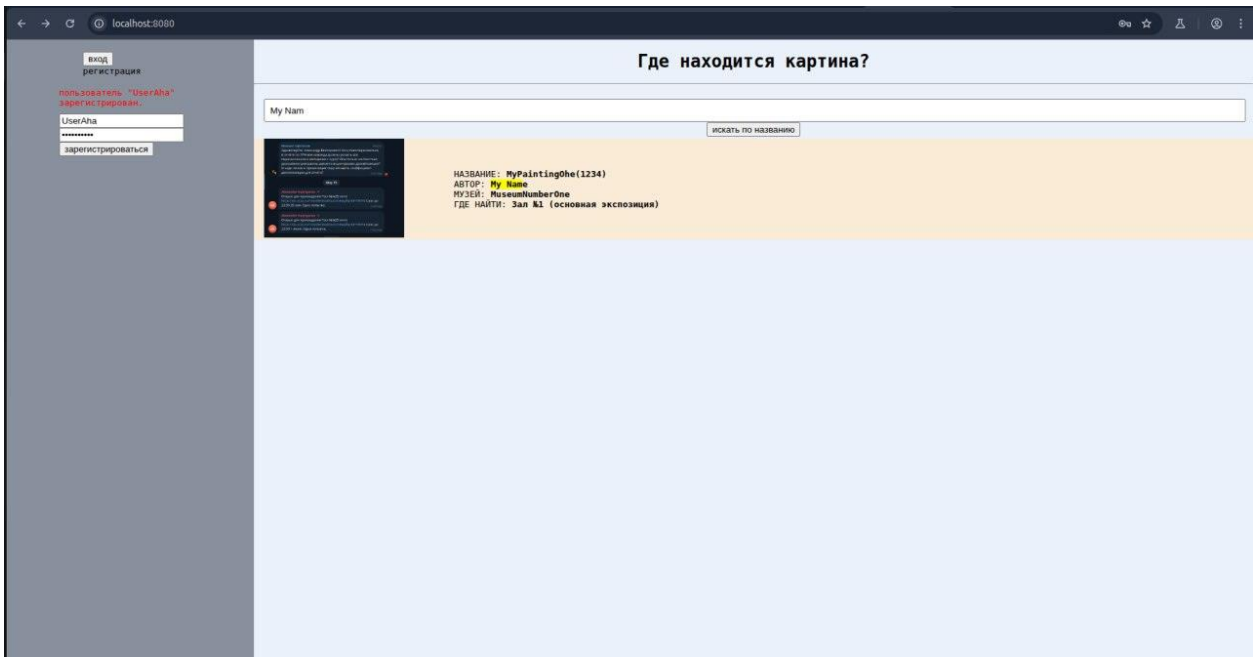

Рисунок 8 – Пользователь успешно зарегистрирован

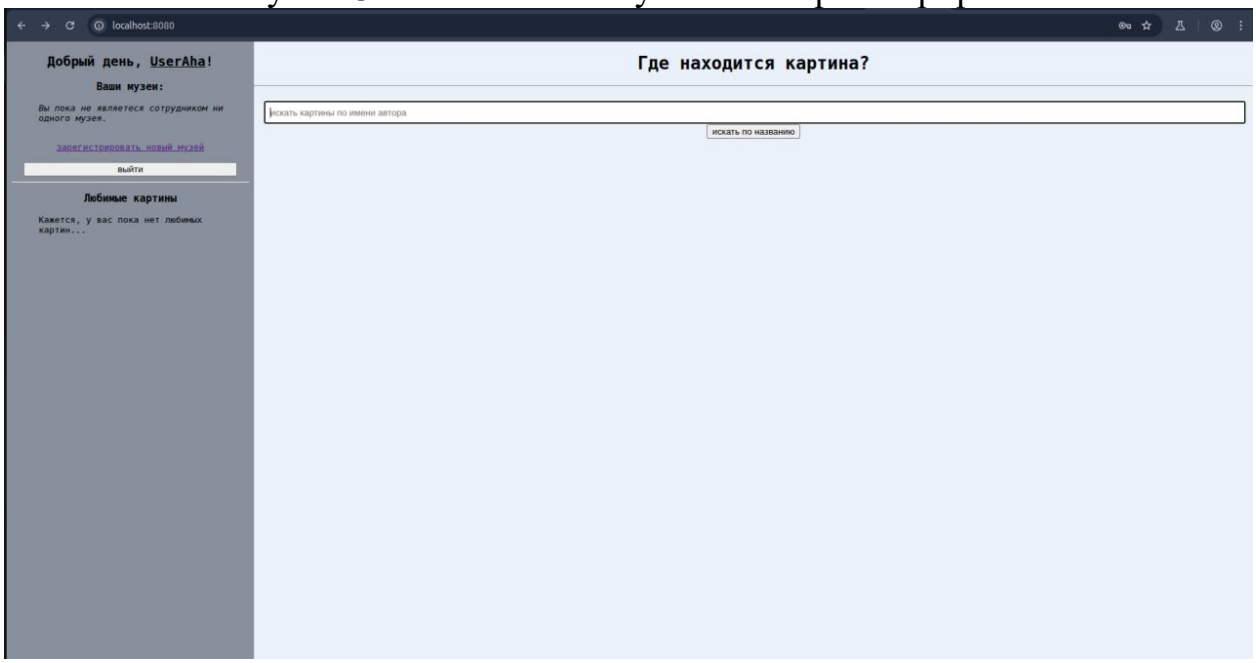

Рисунок 9 – Главная страница для только что авторизованного пользователя (с личным кабинетом слева)

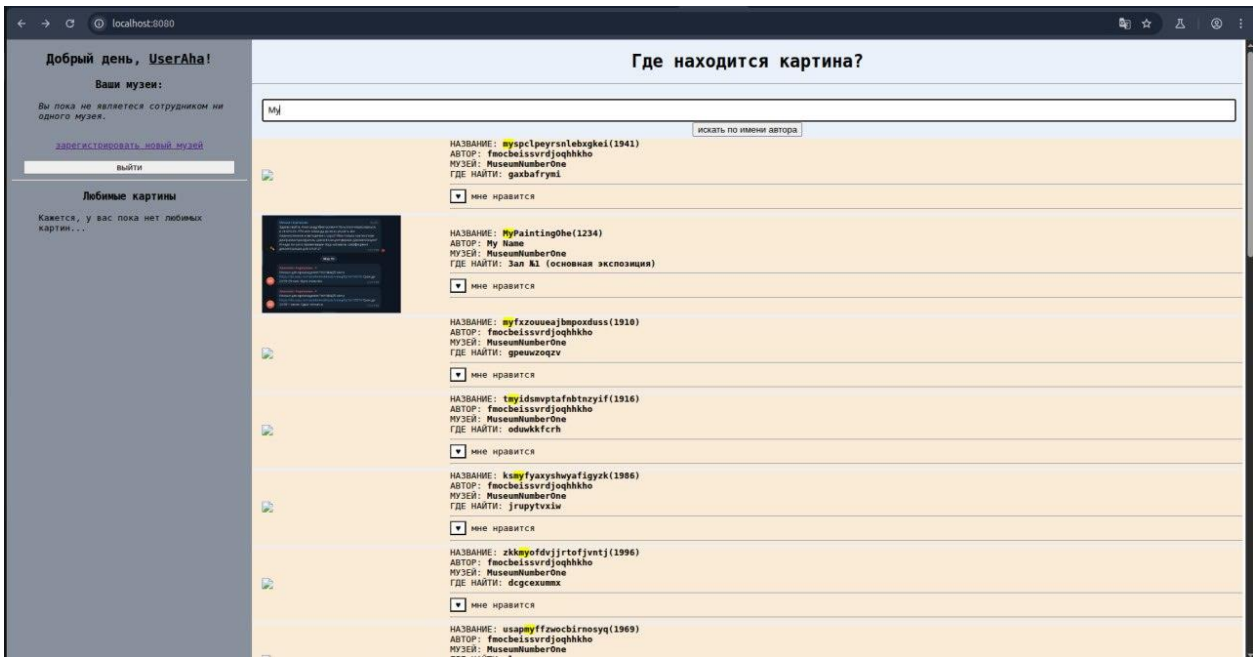

Рисунок 10 – Авторизованный пользователь пытается найти изображение (в результатах появляется опция отметить «мне нравится»

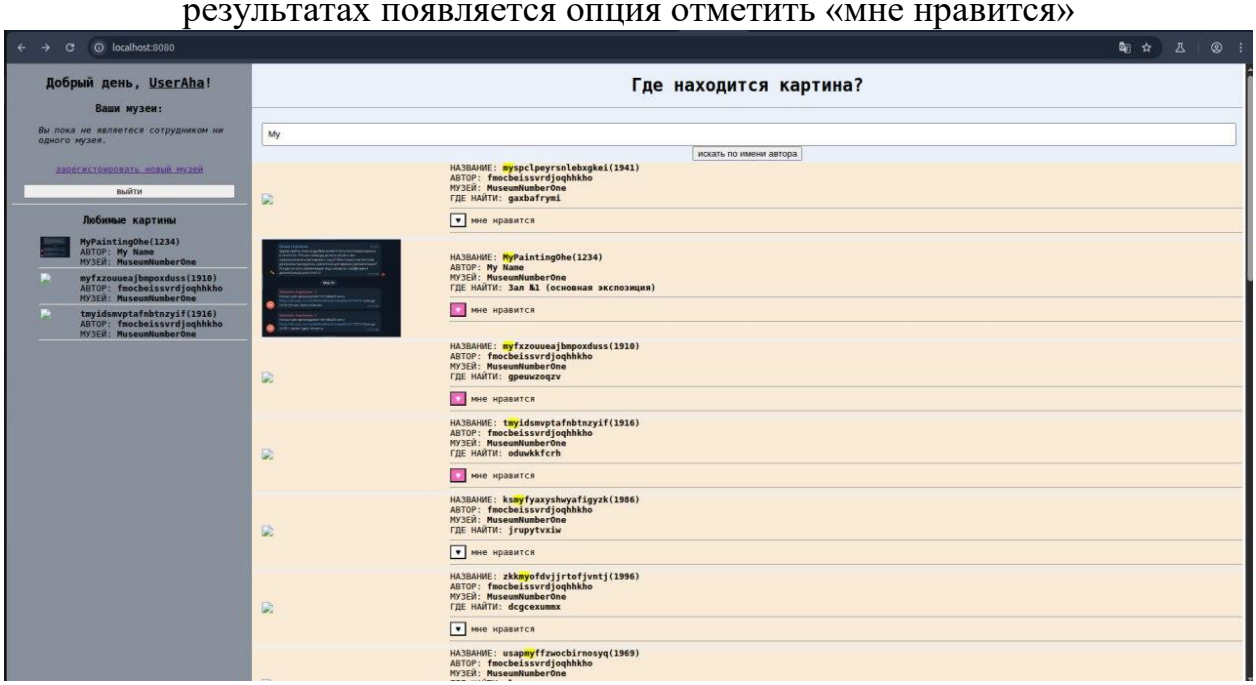

Рисунок 11 – Авторизованный пользователь отмечает найденные картины как понравившиеся и они отображаются в списке в личном кабинете

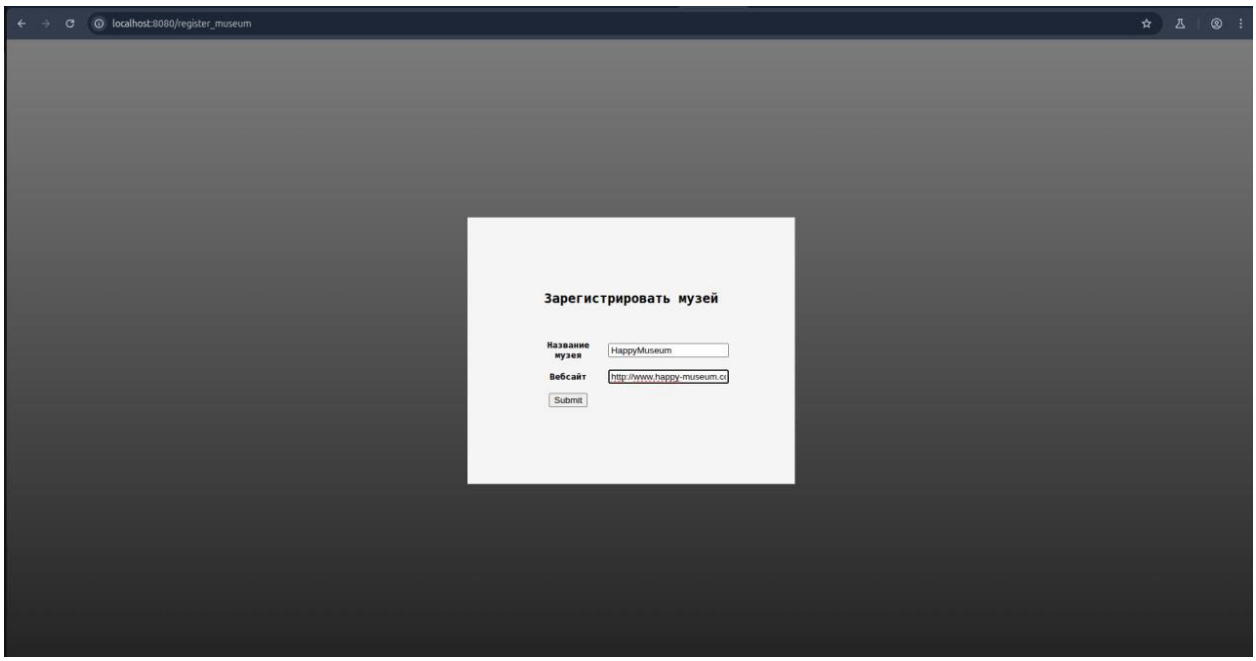

Рисунок 12 – Форма регистрации музея

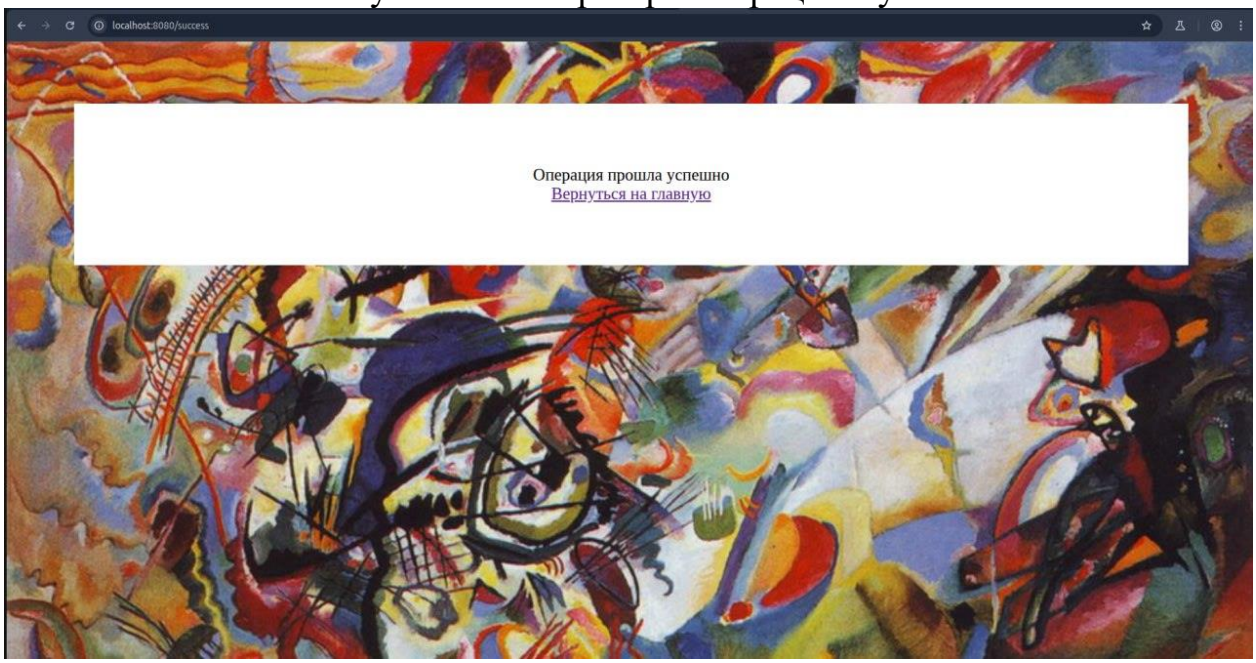

Рисунок 13 – страница успешной регистрации

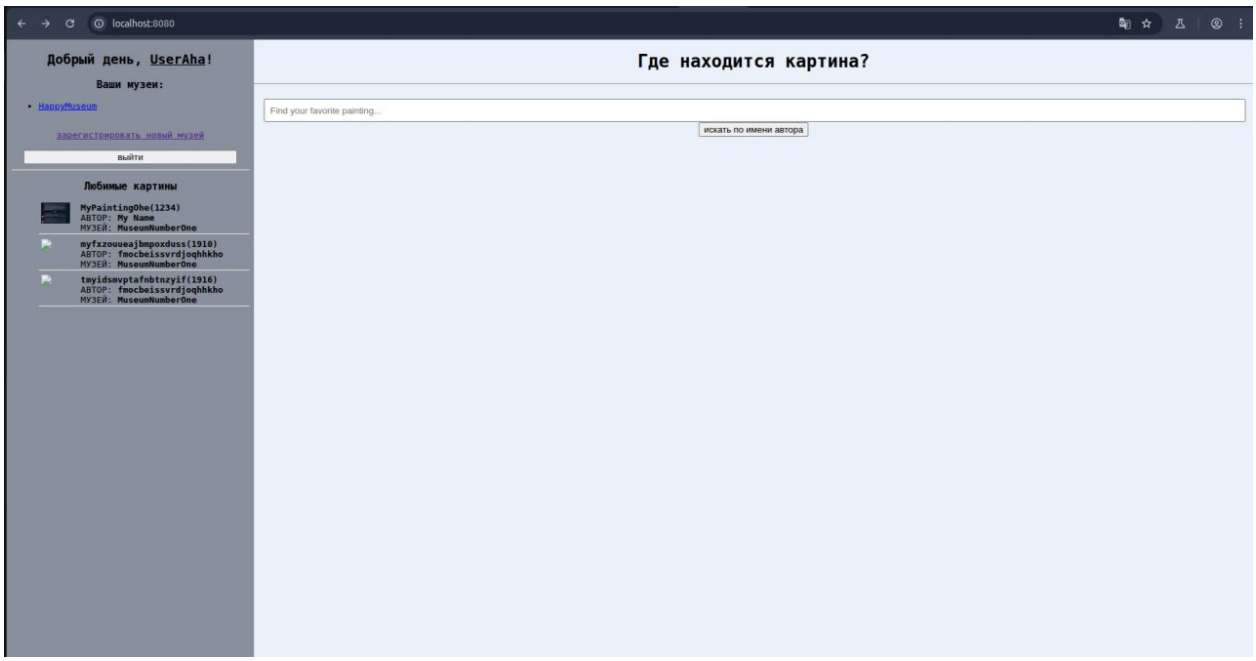

Рисунок 14 – Зарегистрированный музей добавлен в личный кабинет пользователя

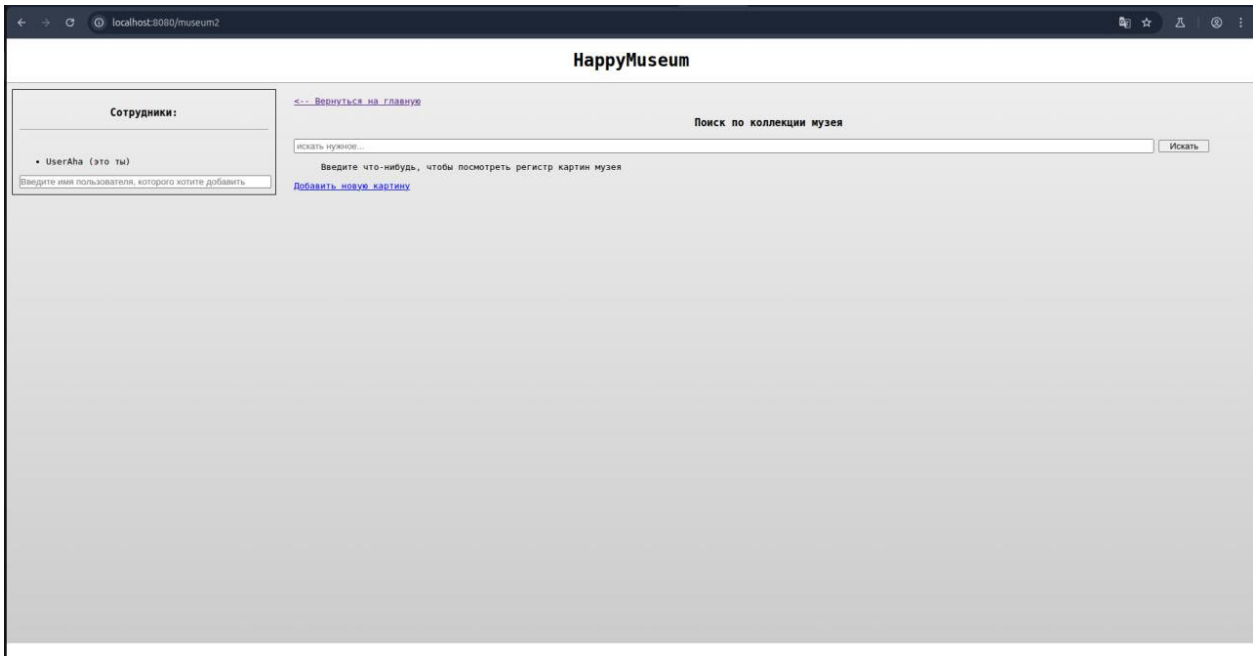

Рисунок 15 – Страница только что созданного музея

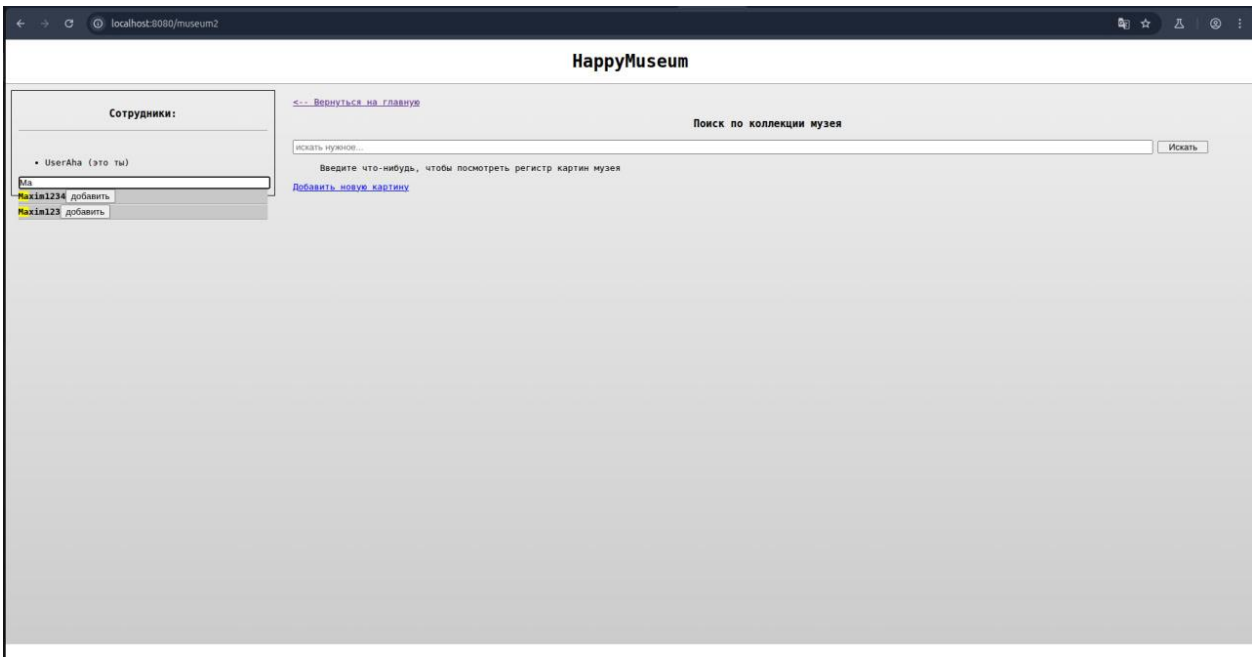

Рисунок 16 – Процесс добавления новых сотрудников в музей (имя вводится в поисковую строку справа и появляется выпадающий список

Рисунок 17 – Страница музея с добавленными сотрудниками так, как её видит администратор музея (можно менять права пользователей, или удалять их)

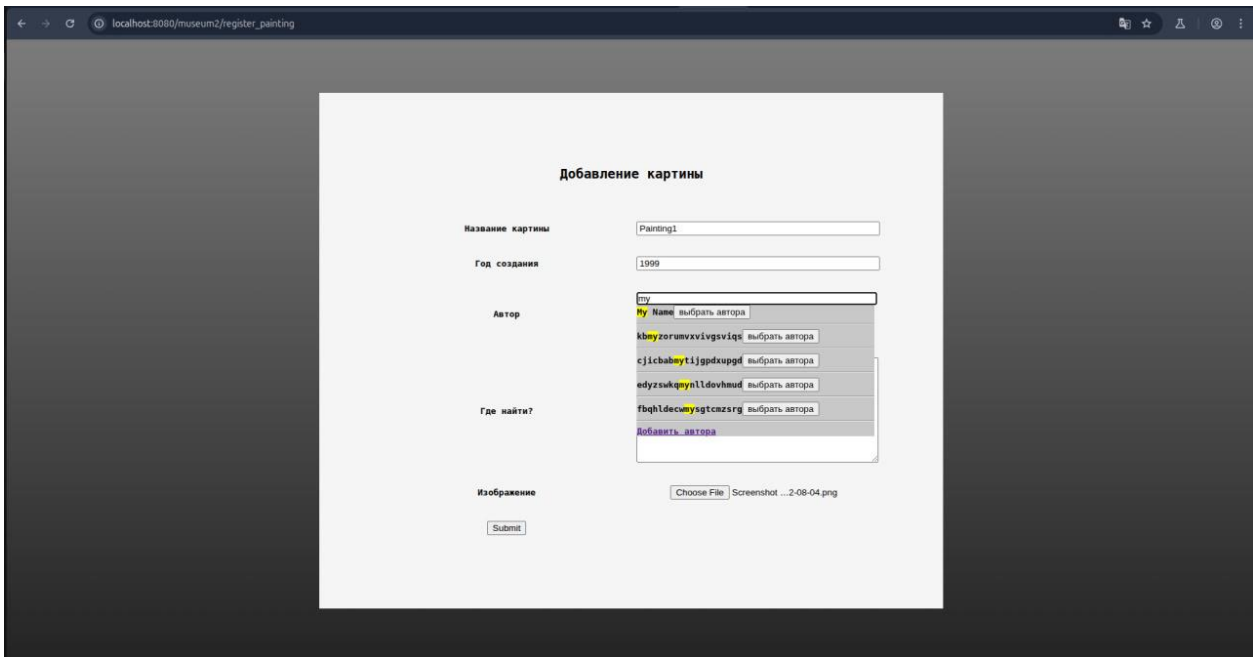

Рисунок 18 – Добавление новой картины в музей, а также процесс выбора автора картины при помощи выпадающего списка

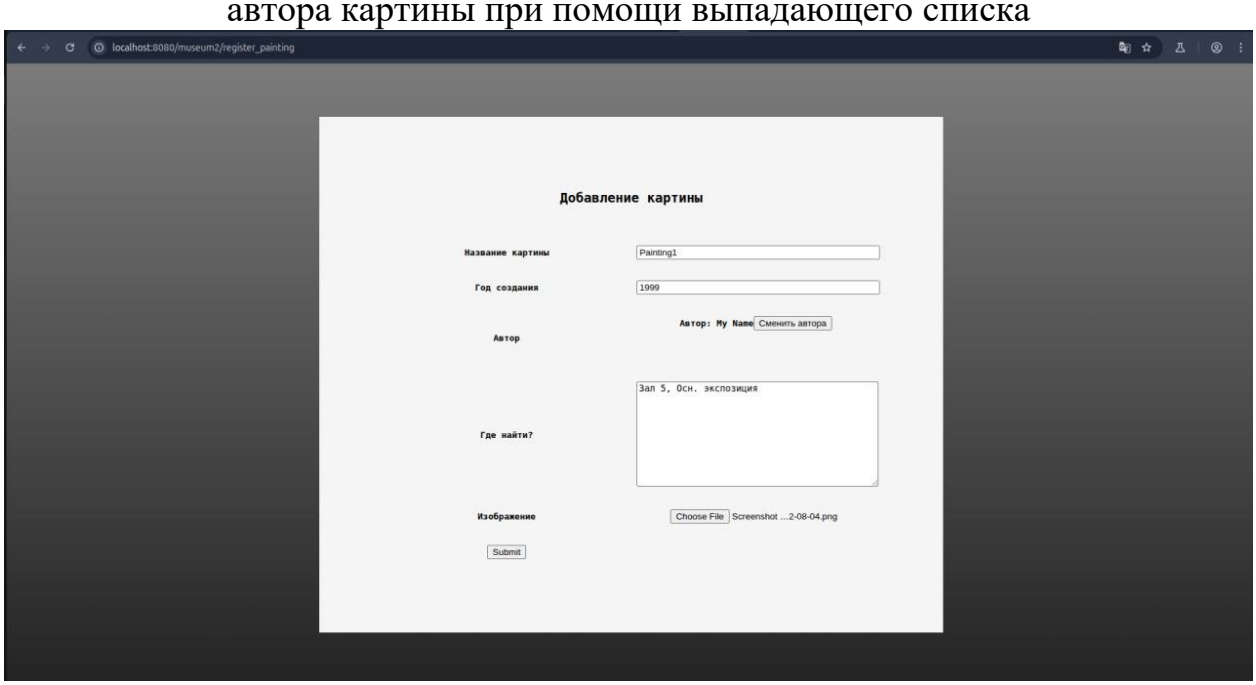

Рисунок 19 – Форма добавления новой картины в музей при выбранном авторе

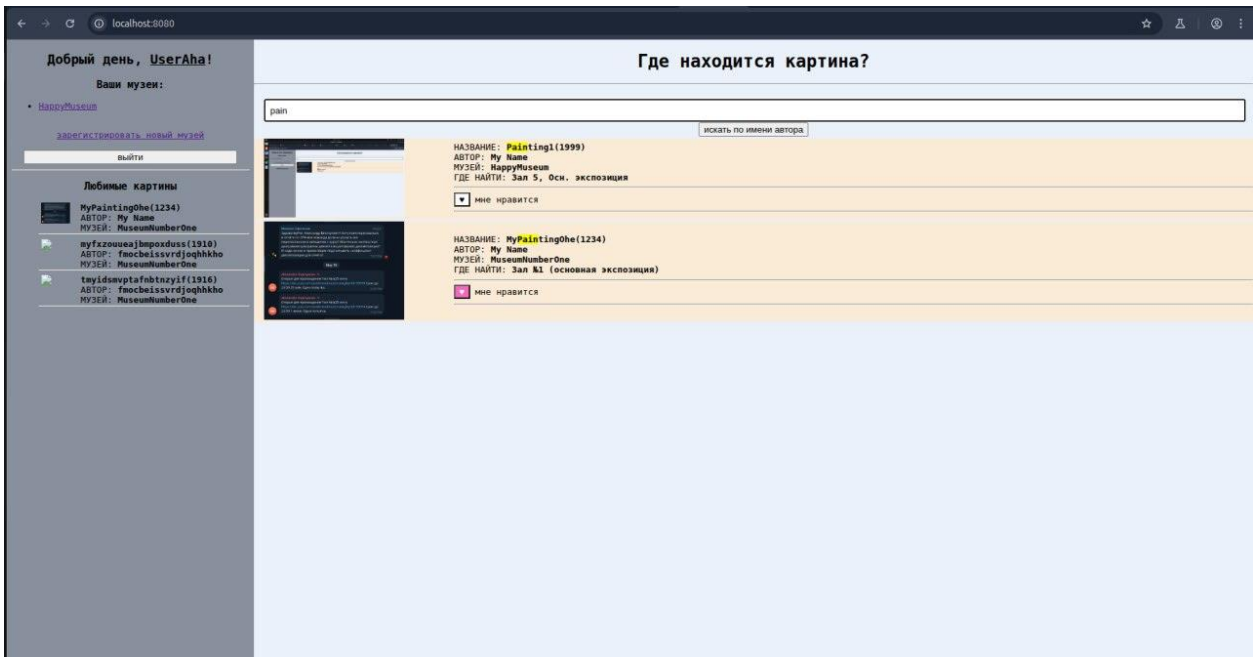

Рисунок 20 – Только что созданная картина на главной странице

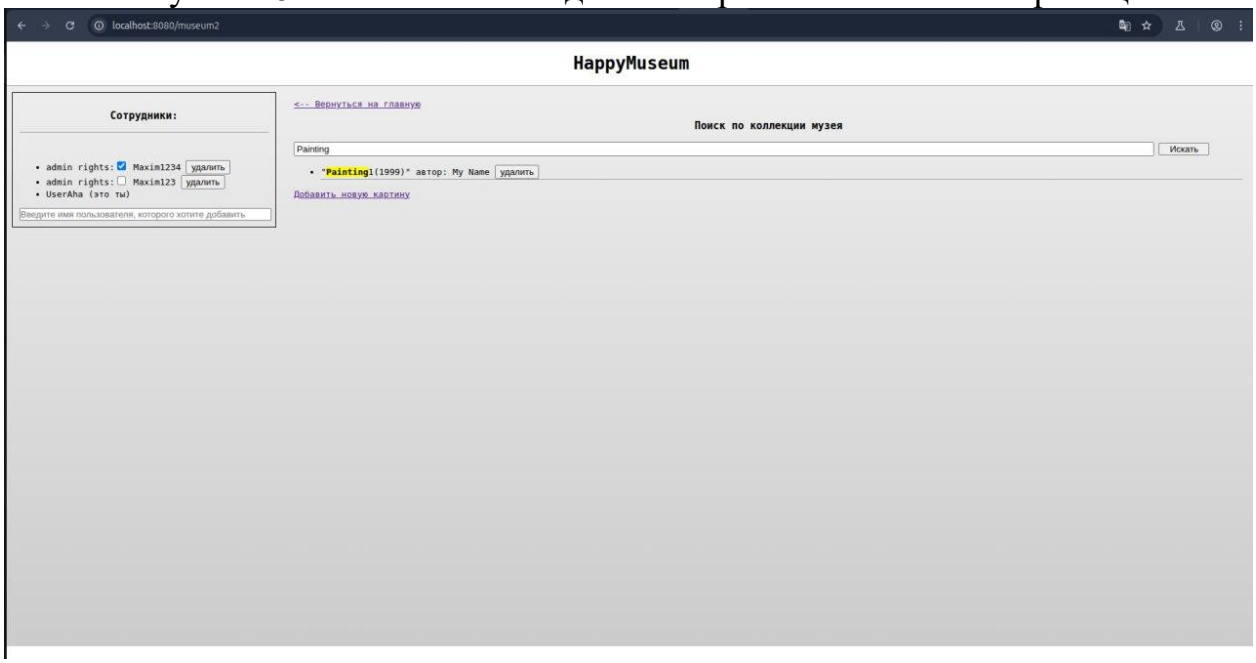

Рисунок 21 – Только что созданная картина в поиске по коллекции музея

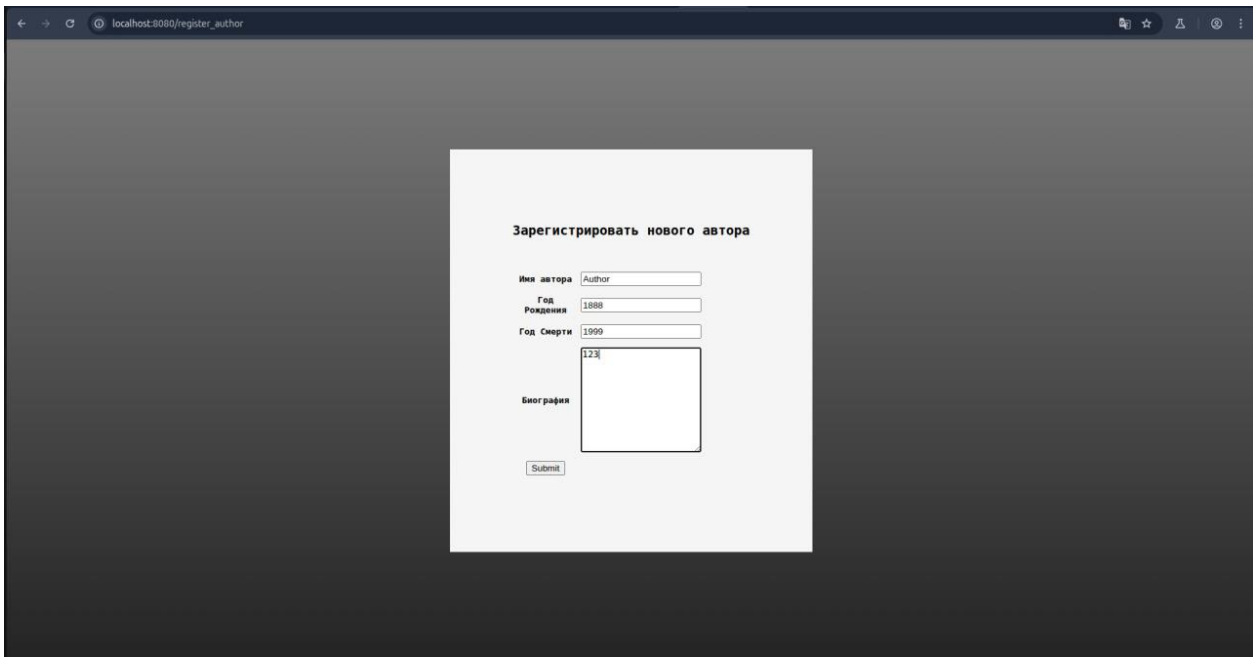

Рисунок 22 – Форма регистрации нового автора

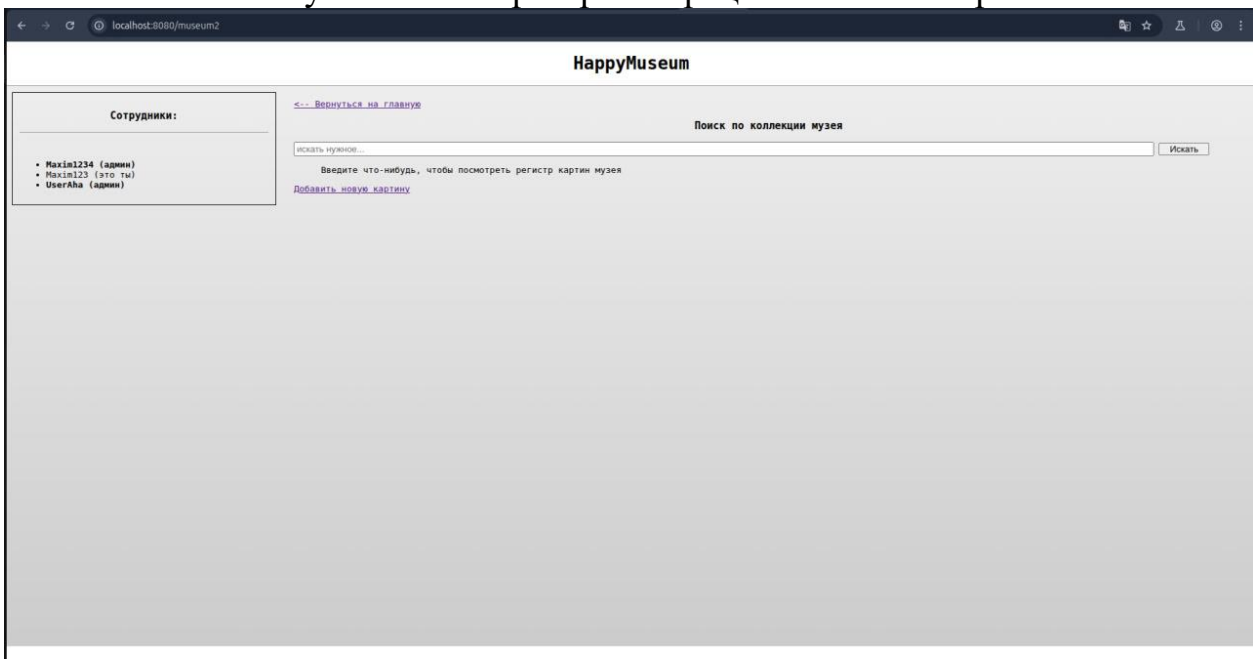

Рисунок 23 – страница музея так, как её видит сотрудник без прав администратора

Контрольная точка 5: Финализация и упаковка в Docker

Я контейнеризовал свой проект при помощи Docker, используя два контейнера: один для базы данных (postgres) и один для приложения (where-is-the-painting). Контейнеры взаимодействуют друг с другом. Также я написал docker-compose.yaml, запускающий оба контейнера. Причем, я сделал возможным запуск приложения как локально (для упрощенной разработки и дебага), так и в связке с контейнером. При разных запусках необходимо слушать разные порты, поэтому я добавил проверку на переменную среды, показывающую нахожусь ли я в контейнере. Также в ходе работы настроил .dockerignore. На рисунке 24 показано, как контейнеры работают (docker desktop).

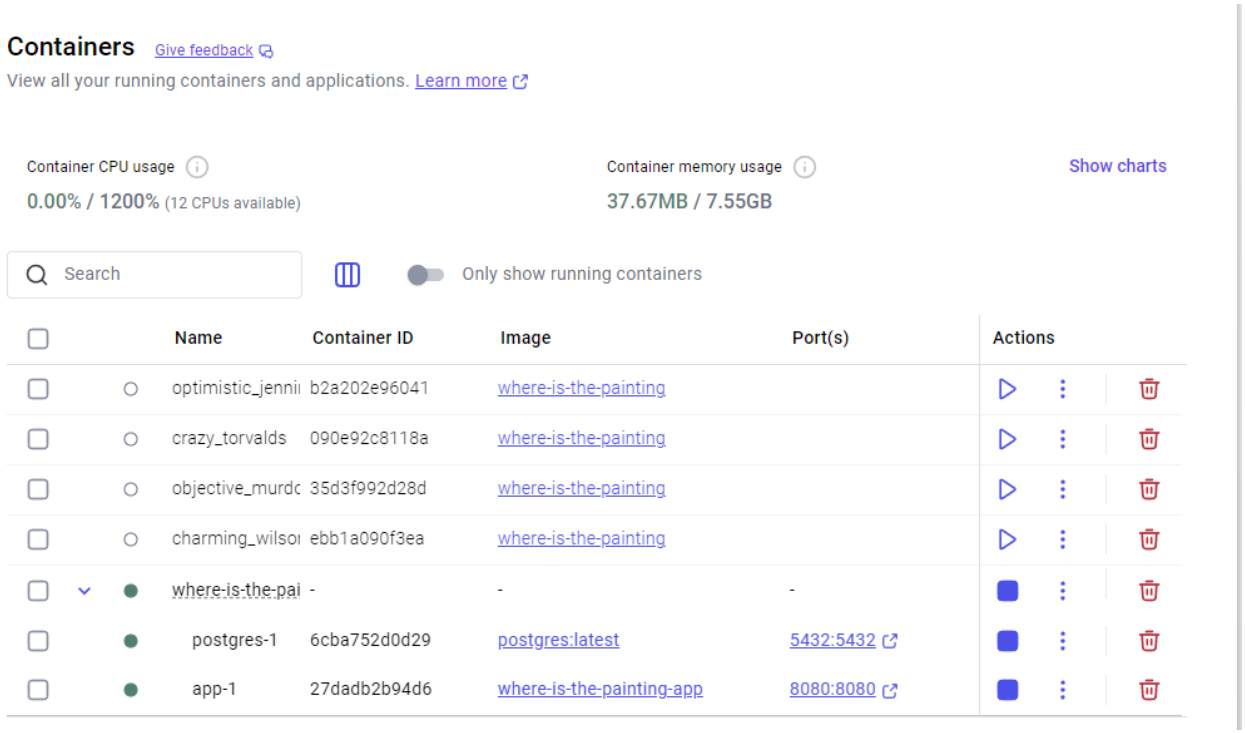

Рисунок 24 – Запущенные контейнеры

Заключение

Проделав данную серию лабораторных работ я приобрел фундаментальные навыки разработки веб-приложений и попрактиковался создавать таковые с использованием языка программирования go. В ходе работы я научился подключаться к базе данных (как при помощи драйвера, так и с помощью ORM), глубже понял работу с реляционными базами данных, научился обрабатывать REST запросы, обрабатывать ошибки. В ходе работы я также научился базовым навыкам создания frontend с использованием стека html/css/js и связал созданный в ходе работы backend с frontend-ом. Также итоговый проект был контейнеризован с помощью Docker.Таким образом, в результате работы у меня получилось создать полноценное веб-приложение.

Конечно, созданный мною проект не идеален и в реальной практике требовал бы больших доработок. В ходе изучения курса я понял, что мог изменить и улучшить многое, что делал, сделать иначе. Среди таких вещей: использование jwt-токенов access и refresh, использования специального хранилища для изображений и проч. Тем не менее полученные мною навыки позволяют мне разобраться с этими технологиями самостоятельно.