МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Информационные системы и автоматизация производства»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к дипломному проекту

Тема: «Разработка информационной системы интернет магазина»

Факультет: повышения квалификации и переподготовки кадров

Специальность: 1-40 01 73 «Программное обеспечение информационных систем»

Группа: ИС-17

Исполнитель: Атрощенко Константин Анатольевич ФИО

Главный консультант: ФИО

Консультанты:

по экономической части ФИО

по охране труда и

промышленной экологии ФИО

по ресурсосбережению

и энергосбережению ФИО

Нормоконтроль: Соколова А.С.

Утверждаю: Зав. кафедрой ИСАП

Казаков В.Е.

Проект рассмотрен и допущен к защите «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Витебск, 2020

Оглавление

[Введение 3](#_Toc47730045)

[1. Анализ объекта 6](#_Toc47730046)

[1.1. Описание структуры информационной системы интернет магазина. 6](#_Toc47730047)

[1.2. Построение концептуальной модели информационной системы интернет магазина. 8](#_Toc47730048)

[2. Постановка задачи. 12](#_Toc47730049)

[2.1. Опреденление требований к информационной системе интернет магазина. 12](#_Toc47730050)

[2.2. Обзор и обоснование выбора инструментальных средств 29](#_Toc47730051)

[3. Проектирование 46](#_Toc47730052)

[3.1. Разработка архитектуры програмного продукта 46](#_Toc47730053)

[3.2. Проектирование структур хранение данных 56](#_Toc47730054)

[3.3. Описание реализации вариантов использования 64](#_Toc47730055)

[4. Реализация 70](#_Toc47730056)

[4.1. Разработка классов информационой системы 70](#_Toc47730057)

[4.2 Разработка интерфейса программного продукта 85](#_Toc47730058)

[4.3 Разработка алгоритмов реализации вариантов использования 85](#_Toc47730059)

[1.1. Модульное тестирование алгоритмов реализации вариантов использования 99](#_Toc47730060)

[2. Системное тестирование 99](#_Toc47730061)

[2.1. Функциональное тестирование 99](#_Toc47730062)

[2.2. Оценка безопасности 99](#_Toc47730063)

[2.3. Тестирование производительности 99](#_Toc47730064)

[3. Экономическое обоснование эффективности внедрения 99](#_Toc47730065)

[4. Охрана труда(применительно к области внедрения) 99](#_Toc47730066)

[5. Промышленная экология 99](#_Toc47730067)

[6. Ресурсосбережение 99](#_Toc47730068)

[Заключение 99](#_Toc47730069)

[Список использованной литературы 99](#_Toc47730070)

[Приложения 99](#_Toc47730071)

Введение

В современном обществе каждый из нас сталкивается с тем, что ему приходится что – то покупать или же наоборот продавать и оказывать услуги. Товарно-денежные отношения являются наиболее распространенной формой взаимодействия между людьми.

2020 год очень негативно сказывается на субъектах предпринимательской и государственной деятельности из – за COVID-19.

Выбранная мною тема считается актуальной на сегодняшний день т.к. миллионы людей, не выходя из дома совершают покупки в интернет-магазинах. Целью работы является разработка информационной системы интернет-магазина.

Для достижения цели в работе необходимо решить следующие задачи:

провести анализ средств разработки и выбрать инструментальную систему для разработки интернет-магазина,

проанализировать предметную область и представить ее в виде структурной схемы UML продемонстрировать при этом все сущности предметной области, атрибуты, функции и связи.

разработать базу данных средствами сервера баз данных MongoDB express,

реализовать вэб-интерфейс интернет-магазина.

Актуальность темы обусловлена тем, что в мире огромными темпами растет количество интернет-магазинов и как следствие количество потенциальных покупателей интернет-магазинов так же растет.

Интернет-магазины существенно уменьшают издержки их владельца на содержании обычного магазина. Из – за того что интернет-магазин можно посетить из любой точки земного шара, существенно расширяется рынок сбыта.

Разработка и запуск интернет-магазина в целом обходится во много раз дешевле, чем открытие обычного магазина. Если владелец обычного магазина тратится на аренду, коммунальные платежи, оплату налогов на имущество, страхование. То обладатель интернет-магазина может вложить данные средства в качественную проработку и раскрутку интернет-магазина.

Интернет-магазин отличается большей гибкостью. К примеру, ассортимент магазина можно обновлять мгновенно настолько часто, насколько это необходимо. На главной странице размещать акции на товары без необходимости в размещении офлайн рекламы, а также в изготовлении и распространении печатных материалов. Интернет-магазин не имеет ограничения по месту, поэтому предлагаемый ассортимент может приближаться к бесконечности. Здесь нет никакой привязки к сезону. Тем самым владельцу отпадает потребность менять ассортимент в зависимости от сезона. Даже не сезонные товары доступны покупателям в любое время года.

Так же, расширяются покупательские возможности. Покупатель может совершить покупку через сайт или приложение в любой стране, в любом городе, 24 часа в сутки 7 дней в неделю.

В конце концов интернет-магазин - это безопасно. В обычных магазинах происходят вооруженные ограбления, пьяные дебоши и т.д.

Интернет-магазины не ограничиваются тем ассортиментом, который при доставке вы можете визуализировать, осязать и даже попробовать на вкус. Покупатель может приобрести различный медиа контент.

На данный момент большой популярностью пользуются различные онлайн площадки интернет-магазинов, которые предоставляют обучающий контент. К ним относятся различные обучающие курсы начиная от естественных, технических и даже гуманитарных видов наук. Так же есть курсы по изучению иностранных языков.

Набирают популярность курсы по обучению и переобучению кадров различных специальностей.

Данный контент интернет магазинов предоставляет возможность проходить обучение удаленно путем просмотра контента как на интернет площадках данных интернет магазинов, так и с возможностью отсылки выбранного медиа контента покупателю по электронной почте, либо предоставления возможности скачивания покупаемого медиа ресурса.

Это дает интернет-магазинам неоспариваемое преимущество перед другими магазинами. Тем самым интернет-магазины, медленно, но уверенно захватывают рынок становясь монополистами в товарно-денежных отношениях. Это является ключевым моментом при переходе производителей с обычной торговли на торговлю в интернет-магазинах.

1. Анализ объекта
   1. Описание структуры информационной системы интернет магазина.

Для современных магазинов сайт - это не основа, но важный и нужный инструмент.

Большинство современных предпринимателей, при решении создать интернет магазин, стремятся получить сервис, который транслирует нужную информацию на пользовательском экране, предоставляя покупателю нужные возможности, а владельцу предоставляет инструменты по контролю над сайтом и обслуживанию клиентов т.е. структурированную информационную систему интернет магазина которая позволит без каких либо технических навыков программирования эффективно управлять интернет магазином владельцу, а покупателю, без киках либо трудностей, путешествовать по страницам интернет магазина.

Любой человек решившийся создать интернет-магазин встает перед выбором движка/программы/платформы – называть это можно как угодно.

Обычно подавляющее большинство начинают использовать отдельно стоящие системы управления сайтом и его оффлайновой составляющей – например 1С. А так же систему CRM (Customer Relationship Management или «управление взаимоотношением с клиентами»). В этом случае все системы должны сообщаться между собой, они должны постоянно передавать друг другу данные о клиентах, заказах, отгрузках, движениях денег, товаров и так далее.

Однако такую систему необходимо поддерживать и для этого необходим программист. Постоянный, приходящий, или компания, к которой придется периодически обращаться.

Чем больше отдельных систем, тем больше проблем взаимодействия между ними. Особенно в современных условиях, когда практически любая IT-система находится в состоянии «вечной беты», то есть, когда программный продукт непрерывно совершенствуется и обновляется, но при этом не имеет состояния, в котором он, условно, абсолютно стабилен. В любой момент любого обновления любой из систем может перестать работать стыковка между ними. Эти моменты нужно постоянно контролировать, в противном случае рассинхронизация тут же остановит какие-либо процессы. Клиенту

не позвонят в нужный момент, посылку не отправят, принятые деньги не отнесутся на заказ, а отгрузка не попадёт в отчёты. Да мало ли чего может произойти.

Есть интегрированные решения когда платформа интернет магазина – это сразу и ERP (управление ресурсами предприятия) и CRM, CMS(Content Management System – система управления контентом сайта). Но подобных систем на рынке практически нет.

Единственный и возможно самый лучший вариант для успешно развивающегося среднего или крупного проекта является «самописный» движок. Создание такого проекта отнимает огромное количество времени. Однако если все хорошо продумать, то магазин отработает многие годы, по крайней мере пока движок не потеряет своей актуальности.

* 1. Построение концептуальной модели информационной системы интернет магазина.

Концептуальная модель – это модель исследуемой предметной области построенаая на основе “Унифицированного языка моделирования UML”.

С помощью данного языка отображается вся логика системы, а так же, ее поведение. Нотации визуального проектирования применяются довольно долго. На мой взгляд они играют основную роль для взаимопонимания. Хорошая диаграмма часто позволяет обмениваться идеями о проекте, особенно когда вы хотите избежать излишне подробного объяснения. Диаграммы также помогают понять и программную систему, и бизнес план.

Когда группа разработчиков пытается в чем-то разобраться, диаграммы помогают установлению взаимопонимания и распространению такого понимания в команде. Диаграммы, по крайней мере пока, не заменяют языки программирования, но способны оказать существенную помощь.

«Пример диаграммы последовательности приведен в приложении А.1»

Рассмотрим базовые компоненты которые используются для построения концептуальной модели.

Сущность – обозначается в виде прямоугольника

Атрибуты – обозначаются в виде овала

Связь – обозначается в виде ромба которая объединяет все сущности

Самой главной сущностью является сам интернет-магазин. У него имеется шесть главных сущностей в виде страниц интернет-магазина:

* Регистрация и авторизация пользователей
* Главная страница
* Добавить курс
* Курсы
* Корзина
* Заказы
* Профиль пользователя

Рассмотрим каждую сущность в отдельности.

Регистрация и авторизация пользователей. В данной сущности имеются два главных атрибута, кнопки “Зарегистрироваться” и “Войти”.

Сущность регистрация имеет четыре атрибута. Три области ввода: имя пользователя, email пользователя, и пароль(password) пользователя.

Так же имеется автоматически генерируемый id пользователя при удачной регистрации пользователя.

Кнопка “ Войти” имеет всего два атрибута email и password.

Так же страница “Регистрации и авторизации” имеет ссылку “Забыли пароль”. Которая ведет на второстепенную сущность “Страница сброса пароля.” У нее имеются атрибуты область ввода email, и кнопка “Сбросить” при нажатии на которую отправляется письмо на email пользователю. В письме пользователь подтверждает что хочет изменить пароль нажатием на ссылку и попадает еще на одну второстепенную сущность “Страница восстановления пароля”. Данная сущность состоит из двух атрибутов: области ввода нового пароля и кнопки “Обновить пароль”.

Главная страница атрибутов не имеет т.к. она используется лишь для описания интернет магазина.

Страница “Добавить курс” состоит из следующих атрибутов: Области ввода названия курса, области ввода цены курса, области ввода url адреса картинки, и текстовой области для добавления html кода а так же кнопки добавить курс с помощью которой происходит формирование карточки товара и редирект на страницу курсов.

Страница “Курсы” имеет сущность “Карточка товара” которая в свою очередь имеет следующие атрибуты.

Title – название товара,

Price – цена товара,

Img – картинка товара

Так же имеются кнопки “Купить курс”, “Открыть курс”, “Редактировать курс”.

При нажатии на кнопку “Редактировать курс” происходит переход на второстепенную сущность “Страница редактирования курса” которая состоит из следующих атрибутов: Области ввода названия курса, области ввода цены курса, области ввода url адреса картинки, и текстовой области для добавления html кода. Так же имеются два атрибута кнопок “Редактировать” и “Удалить”.

Кнопка “Открыть курс” делает перенаправление на “Страница описания курса”.

Кнопка “Купить курс” делает перенаправление на страницу “Корзина” где формируется список заказов.

Страница “Корзина” имеет следующие атрибуты. “Название курса”, “Количество(count)”, “Счетчик цены(count-price)”, а так же, кнопки “Удалить” и “Сделать заказ”.

Страница “Заказы” имеет следующие атрибуты: id заказа, Имя пользователя, email, Дата заказа, список курсов и их количество, цена.

Страница “Профиль пользователя” имеет следующие атрибуты: email, Имя пользователя, файловый загрузчик, кнопку “Изменить”.

Если присмотреться к данной концептуальной модели, то можно обнаружить, что хоть данная модель максимально отображает архитектуру ПО, однако она не дает полную картину того, что будет происходить при взаимодействии пользователя с данным программным продуктом.

1. Постановка задачи.
   1. Опреденление требований к информационной системе интернет магазина.

Пользовательский сценарий Use-case помогает описывать систему какой она должна быть с точки зрения использования. Данный подход позволяет продумать каким должен быть пользовательский интерфейс.

Пользовательский сценарий Use-case помогает описывать систему какой она должна быть с точки зрения использования. Данный подход позволяет продумать каким должен быть пользовательский интерфейс.

Use-case диаграмма представляет собой последовательность действий выполненных системой в ответ на события инициируемые действующим лицом. Она описывает взаимодействие между пользователем и системой и отражает поведение системы с точки зрения пользователя.

Действующее лицо - это роль которую играет пользователь по отношению к системе. Действующие лица представляют собой роли, а не конкретных людей и наименование работы.

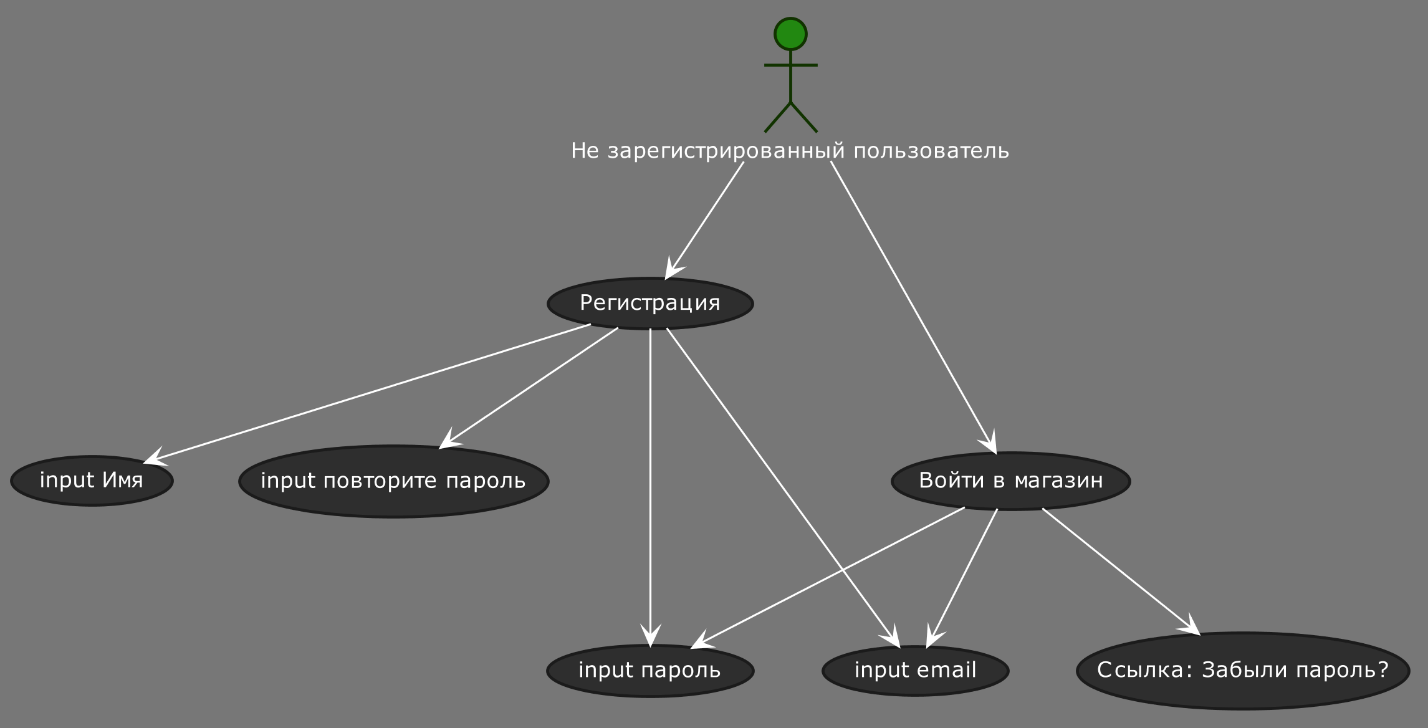
Типы действующих лиц:

- Пользователи системы

- другие системы, взаимодействующие с данными

- время.

Рассмотрим информационную систему интернет-магазина на примере Use-case диаграмм.

Рисунок 2.1 – UC-1 Регистрация и авторизация пользователя.

В пользовательском сценарии UC-1 показано взаимодействие пользователя с формами регистрации и авторизации пользователя.

Actor – значок человечка.

Use-case – пользовательский сценарий овалом.

Между активным субъектом и вариантом использования устанавливается связь т.е. ассоциация (стрелка) которая выполняет коммуникативную функцию сообщая о взаимодействии субъекта с системой в рамках определенного варианта использования.

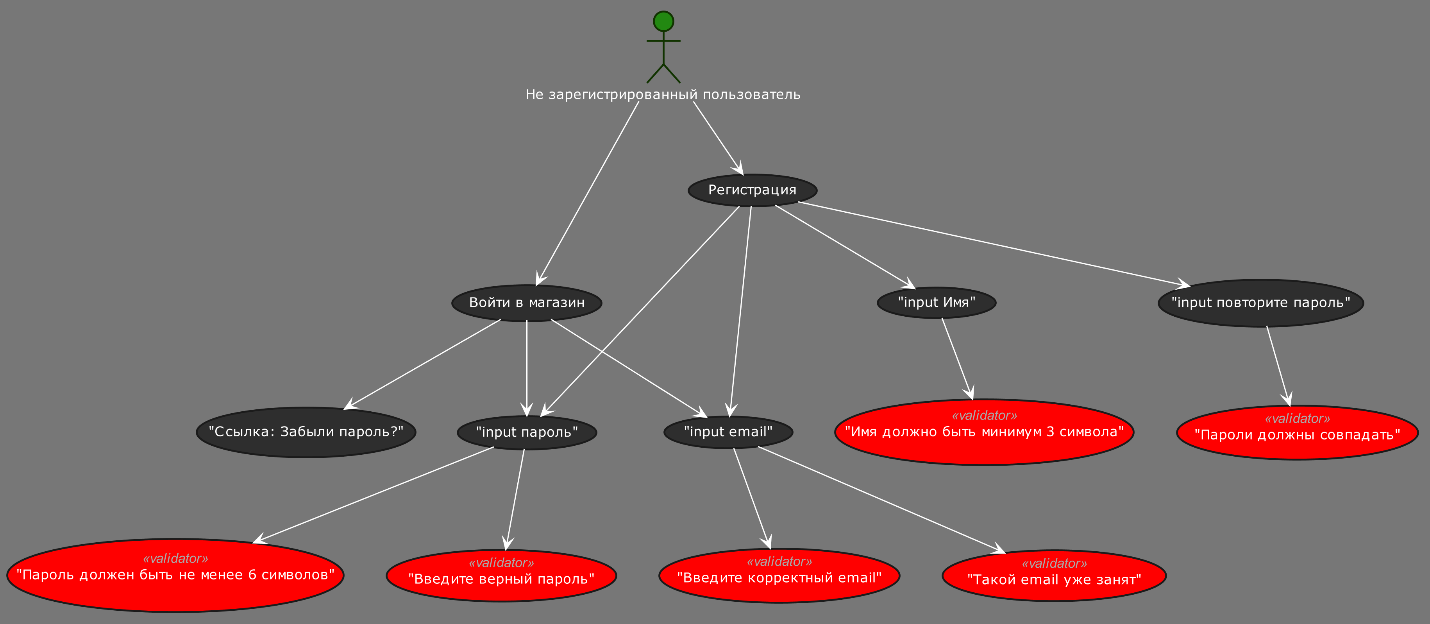


Рисунок 2.2 – UC-2 Валидация форм регистрации и авторизации пользователя.



Рисунок 2.3 – UC-3 Навигация не зарегистрированного пользователя.

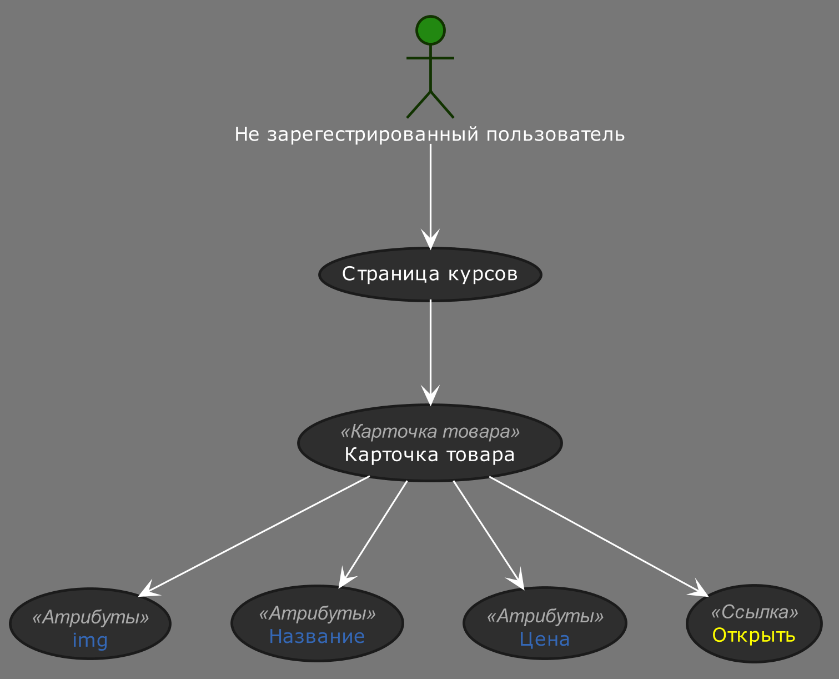


Рисунок 2.4 -UC4 Страница «Курсы» для не зарегистрированного пользователя.



Рисунок 2.4 – UC-5 Зарегистрированный пользователь «Главная».

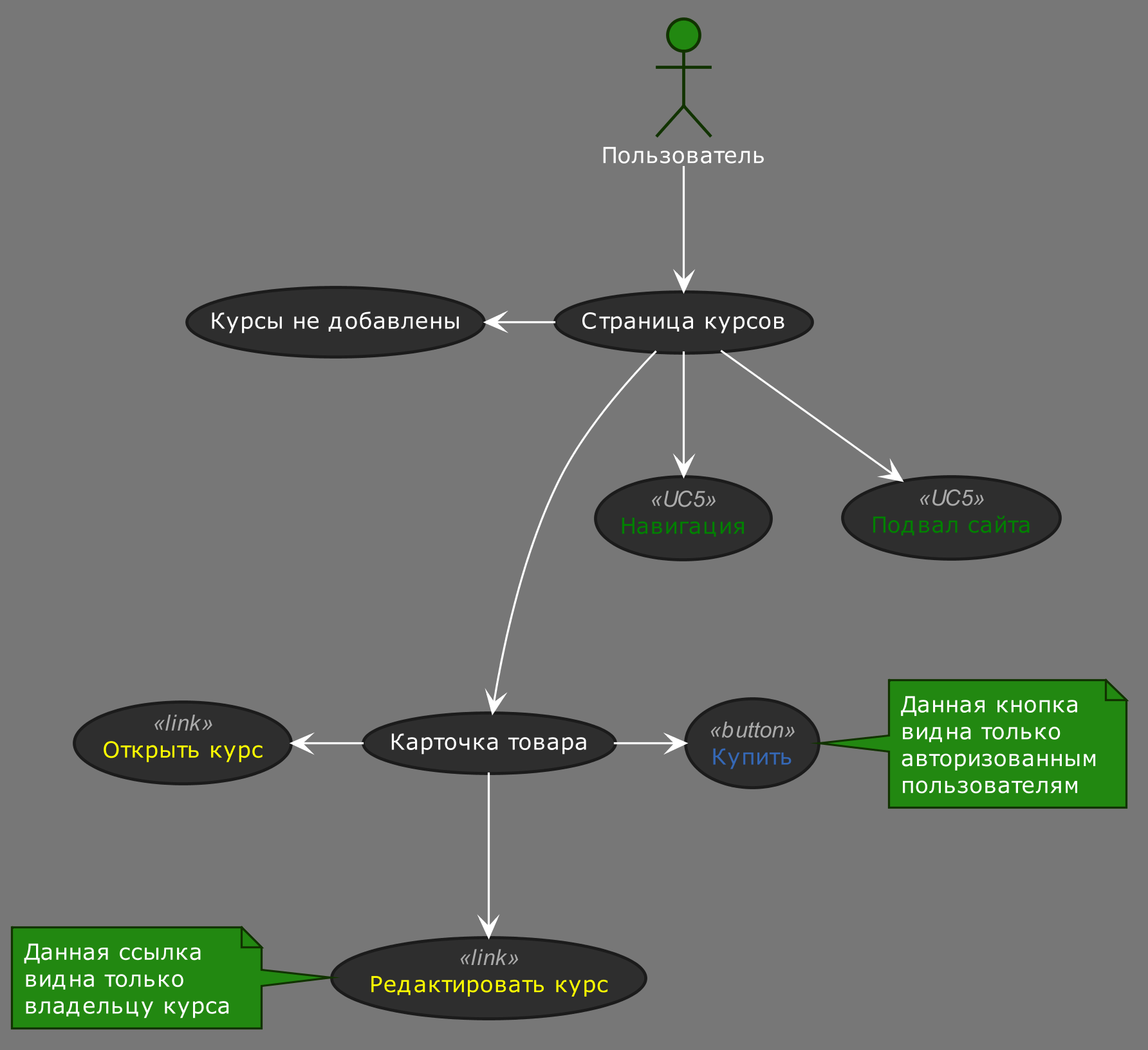


Рисунок 2.4 – UC-6 Страница «Курсы» авторизованного пользователя.

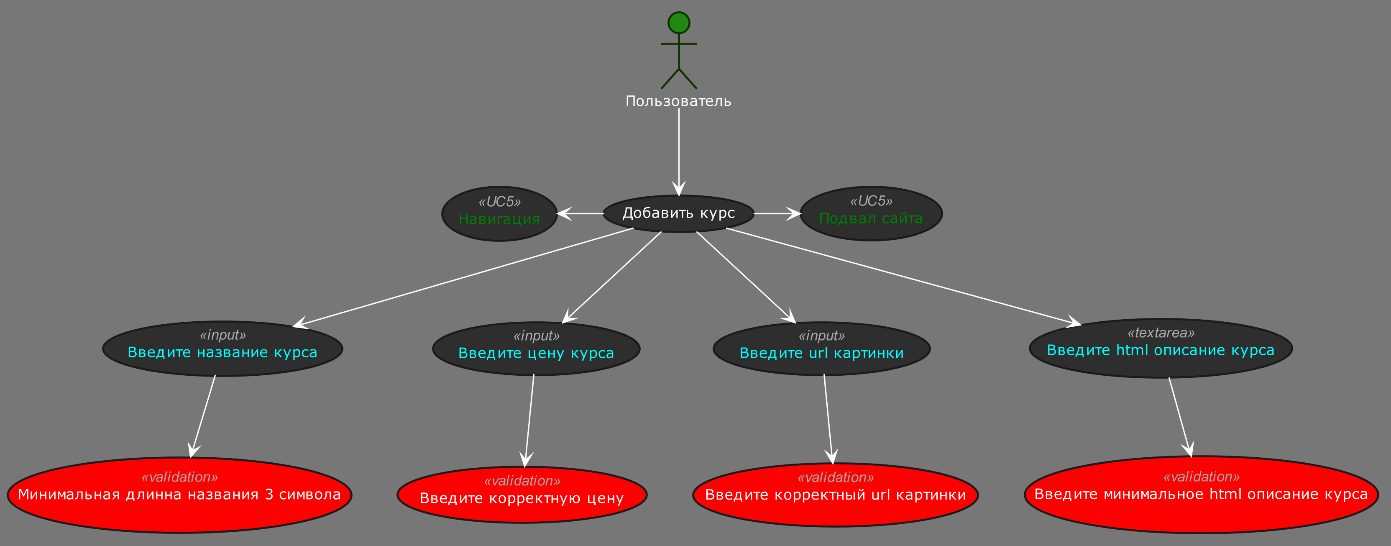


Рисунок 2.5 – UC-7 Страница «Добавить курс».

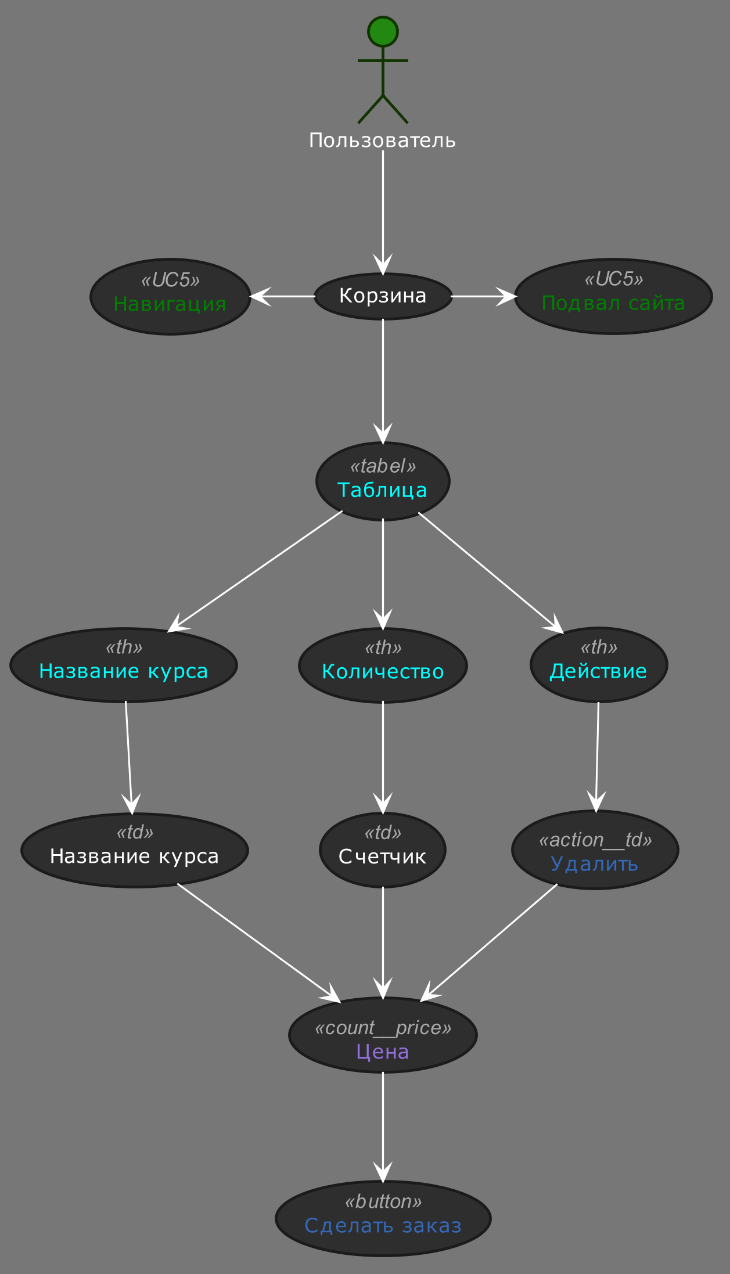


Рисунок 2.6 – UC-8 Страница «Корзина».

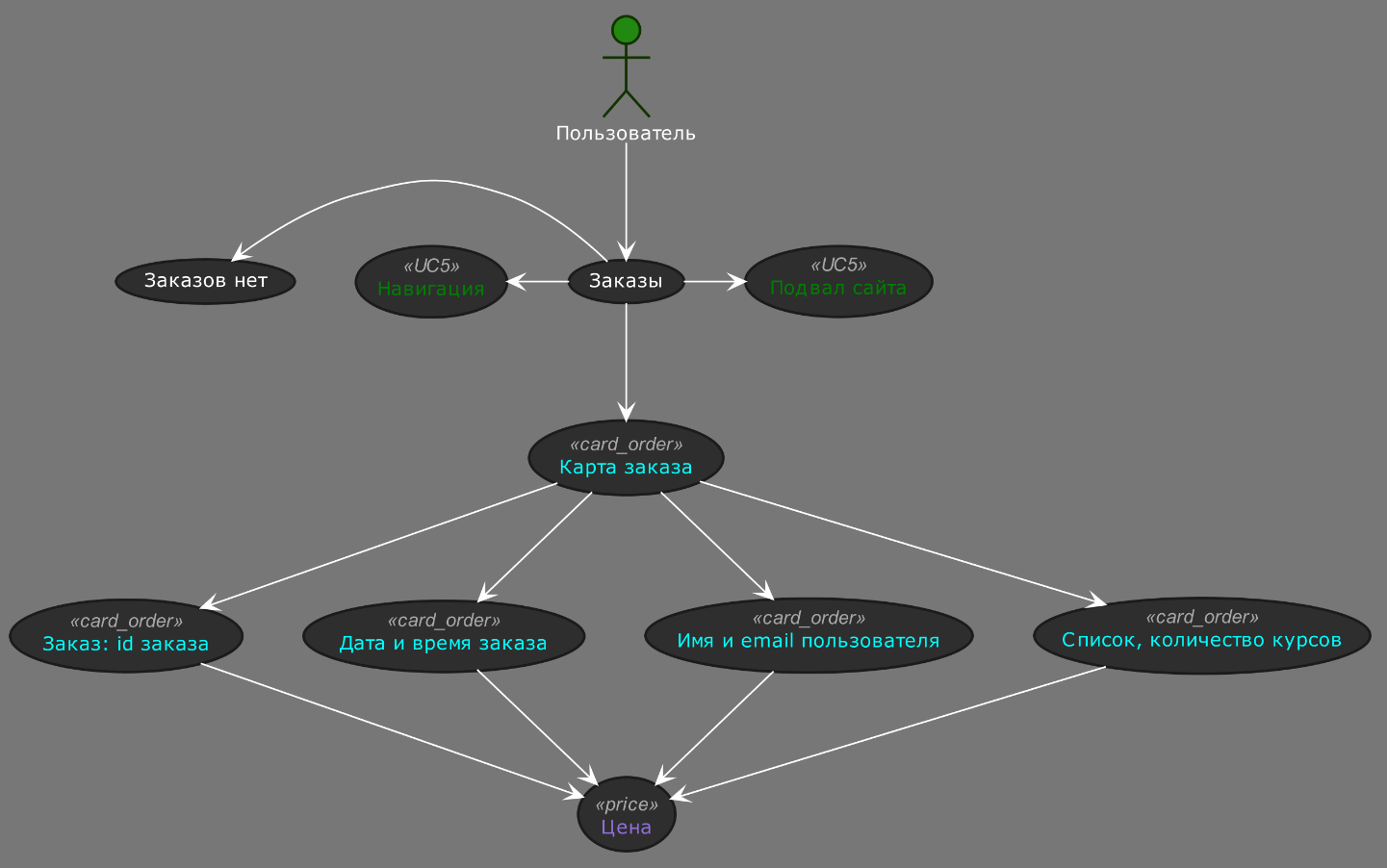


Рисунок 2.7 – UC-9 Страница «Заказы».

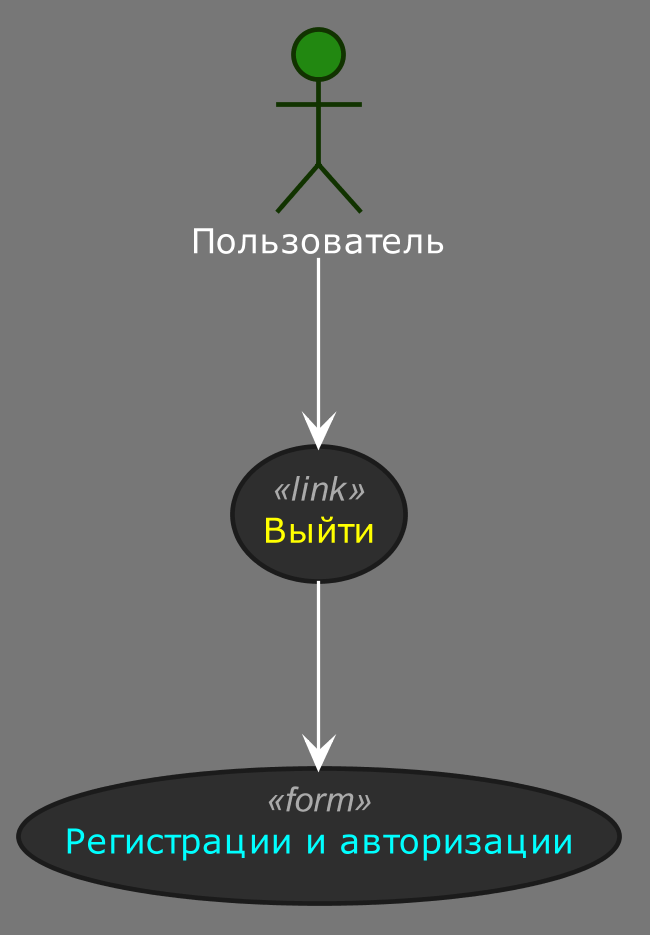


Рисунок 2.8 – UC-10.

Однако Use-case диаграммы являются не самым лучшим способом.

При получении технического задания от заказчика, лучшим способом что бы ничего не пропустить, это составить спецификации требований.

Делается это для того чтобы избежать огромной массы не точностей и в первую очередь для того что бы избежать недопониманий с заказчиком. т.к. зачастую даже сам заказчик не видит конечного результата. Спецификации требований позволяют составить четко структурированный план, который предъявляется заказчику. В свою очередь, если заказчик захочет внести изменения, то это произойдет на этапе формирования пользовательского сценария, а не на этапе разработки программного обеспечения.

Спецификации требований можно представлять не только в виде овала в UML диаграмме, но и в качестве списка с пунктами и подпунктами.

UC-1 Не зарегистрированный пользователь.

Не зарегистрированный пользователь попадает на страницу «Регистрации и авторизации пользователя»

1. Страница «Регистрации и авторизации пользователя» состоит:
   1. Область навигации.
      1. Ссылка «Магазин курсов». Ссылка содержит SVG планеты. При клике как на SVG или на ссылку происходит переход на главную страницу.
      2. Ссылка «Главная страница». Пользовательский сценарий описан в UC- 5.
      3. Ссылка «Курсы». Пользовательский сценарий описан в UC- 6.
         1. Страница «Курсы» для не зарегистрированного пользователя состоит из карточки товара.
            1. Карточка товара не зарегистрированного пользователя состоит:

Картинки или логотипа товара.

Цена курса.

Ссылки «Открыть курс» которая ведет на страницу детального описания курса.

* + - 1. Ссылки «Войти» которая ведет на форму регистрации и авторизации. Пользовательский сценарий UC-2.
  1. Форма регистрации и авторизации пользователя. Пользовательский сценарий UC-2.
  2. Подвал сайта (Footer). Состоит из трех блоков:
     1. Первый блок:
        1. Ссылка SVG планета и «Магазин курсов». При клике переходит на главную страницу.
        2. Свидетельство 0129174, 24 января 2020г. УНП 192761153, дата регистрации в торговом реестре РБ: 08.03.2020г.
        3. SVG карты оплаты Visa, MasterCard, Maestro.
     2. Второй блок:
        1. +375298918971
        2. +375448918971
        3. 210040 г. Витебск ул. Чкалова 41/1 каб 312.
     3. Третий блок:
        1. Ссылка SVG Вконтакте.
        2. Ссылка SVG Facebook.
        3. Ссылка SVG Twitter.
        4. Ссылка SVG Google.
        5. Ссылка SVG Instagram.

1. При возникновении ошибок когда пользователь переходит по ссылкам ведет на страницу ошибки 404:
   1. Область навигации.
   2. Заголовок «404 Ошибка».
   3. Параграф «Страница не найдена».
   4. Ссылка «Перейдите по ссылке на главную страницу».
   5. Подвал сайта.

UC-2 Форма регистрации и авторизации пользователя:

1. Область навигации и подвал сайта описаны в UC-1.
2. Форма регистрации и авторизации состоит из кнопок позволяющих переключаться между формами.
   1. Кнопка «Войти в магазин» перенаправляет на форму авторизации:
      1. Label Email.
      2. Input Email.
         1. Валидация «Введите корректный email».
      3. Label Пароль.
      4. Input Пароль.
         1. Валидация «Не верный пароль».
      5. Кнопка «Войти» производит сверку Email и пароля в БД. После удачной авторизации должно происходить перенаправление на главную страницу интернет-магазина.
      6. Ссылка «Забыли пароль?» При нажатии на которую должно происходить перенаправление на страницу «Сброса пароля».
   2. Кнопка «Регистрация» перенаправляет на форму регистрации:
      1. Label Имя.
      2. Input Имя.
         1. Валидация «Имя должно быть минимум 3 символа».
      3. Label Email.
      4. Input Email.
         1. Валидация «Введите корректный email».
         2. Валидация «Такой email уже занят»
      5. Label Пароль.
      6. Input Пароль.
      7. Валидация «Пароль должен быть не менее 6 символов».
      8. Label Пароль еще раз.
      9. Input Пароль еще раз.
      10. Валидация «Пароли должны совпадать».
      11. Кнопка Зарегистрироваться» при которой должно происходить шифрование сессии с занесением пользовательских данных в hooks и в MongoDB для того, чтобы уменьшить шанс взлома приложения. После этого должно происходить перенаправление на страницу «Войти в магазин», где пользователь должен авторизоваться.

UC-3 Страница «Сброса пароля»:

1. Область навигации и подвал сайта описаны в UC-1.
2. Label Email.
3. Input Email.
   1. Валидация «Введите корректный email».
4. Кнопка «Сбросить» отправляет письмо на email пользователя со следующим содержимым:
   1. Тема письма: «Восстановление доступа».
   2. Заголовок: «Вы забыли пароль?».
   3. Параграф: Если нет, то проигнорируйте данное письмо. Иначе перейдите по ссылке.
   4. Ссылка содержит в себе первичный ключ, путь до страницы а так же сгенерированный в БД resToken и текст ссылки «Восстановление доступа».
   5. Вторая ссылка «Магазин курсов».
5. В письме при подтверждении пользователем о восстановлении доступа, путем нажатия на ссылку, должно происходить сравнение resToken отправленного в письме пользователю с resToken в БД. После чего должно происходить перенаправление на страницу ввода нового пароля.
6. Страница ввода нового пароля состоит:
   1. Label Новый пароль.
   2. Input Новый пароль.
      1. Валидация «Пароль должен быть не менее 6 символов».
   3. Кнопка «Обновить пароль» при нажатии на которую должно происходить редактирование пользовательских данных в сессии, а именно замена существующего пароля на новый и перенаправление на страницу логина, где с помощью ввода новых реквизитов пользователя, должна происходить авторизация пользователя и перенаправление на главную страницу интернет-магазина.

UC-4 Навигация зарегистрированного пользователя:

1. Область навигации и подвал сайта описаны в UC-1.
2. При прохождении авторизации пользователем навигация выглядит следующим образом:
   1. «Главная».
   2. «Курсы».
   3. «Добавить курс».
   4. «Корзина».
   5. «Заказы».
   6. «Профиль».
   7. «Войти».

UC-5 «Главная»:

1. Область навигации описанная в UC-3 или область навигации и подвал сайта описанные в UC-1.
2. Контейнер с описанием.
   1. Данное приложение является дипломным проектом слушателя группы ИС-17. Факультета переподготовки кадров Атрощенко Константина Анатольевича.
3. Контейнер в виде повернутого параллелограмма:
   1. О приложении: Данное приложение не является коммерческим. Это лишь демонстрация изученного материала. «Информационная система интернет-магазина курсов» позволяет каждому зарегистрированному пользователю продавать свои курсы. Более подробную информацию можно узнать при клике на сертификат.
   2. SVG ссылка сертификат.
      1. При возникновении ошибки происходит перенаправление на страницу ошибки 404. Пользовательский сценарий описан в UC-1.

UC-6 Страница курсов:

1. Область навигации описанная в UC-3 или область навигации и подвал сайта описанные в UC-1.
2. Блок курсов состоящий из карточек товаров.
   1. Карточка товара владельца курса состоит:
      1. Картинка или логотип.
      2. Название курса
      3. Цена курса
      4. Ссылки «Открыть курс»
         1. Происходит перенаправление на страницу детального описания курса.
      5. Ссылки «Редактировать курс»
         1. Происходит перенаправление на страницу редактирования курса.
         2. Состоит из области навигации описанной в UC3, подвала сайта описанного в UC2, формы редактирования курса.
         3. Форма редактирования курса состоит:
            1. Input placeholder «Введите название курса». Название курса должно подставляться автоматически в соответствии с редактируемым курсом т.е. placeholder должен заменяться на значение имени редактируемого курса.
            2. Валидация. Минимальная длинна названия 3 символа.
            3. Input placeholder «Введите цену». Значение цены должно подставляться автоматически в соответствии с редактируемым курсом т.е. placeholder должен заменяться на значение цены редактируемого курса.
            4. Валидация. Введите корректную цену.
            5. Input placeholder «Введите url картинки». URL картинки должен подставляться автоматически в соответствии с редактируемым курсом т.е. placeholder должен заменяться на значение url пути картинки редактируемого курса.
            6. Валидация. Введите корректный url адрес.
            7. Текстовая не должна выводить значение редактируемого курса. Вместо этого должен отображаться placeholder «Вставьте html шаблон описания курса».
      6. Валидация. Введите минимальное HTML описание курса. Однако значение выставлено в ноль для того что бы пользователь мог внести описание позже.
         1. Кнопки «Редактировать курс» при клике на которую должен происходить анализ редактируемых параметров и в случае введенных каких-либо новых реквизитов карточки товара, старые данные должны обновится на новые, а не затронутые параметры должны оставаться прежними.
         2. Кнопка «Удалить» производит полное удаление карточки товара. Перенаправляет на страницу курсов где происходит новый поиск всех курсов по id с последующей их отрисовкой.
      7. Кнопка «Купить» добавляет курс в корзину.

UC-7 Добавить курс:

1. Блок навигации описанный в UC3 блок footer описанный в UC2.
2. Форма добавления курса состоит:
   1. Input placeholder «Введите название курса»
      1. Валидация. Минимальная длинна названия 3 символа.
   2. Input placeholder «Введите цену курса»
      1. Валидация. Введите корректную цену
   3. Input placeholder «Введите url картинки»
      1. Валидация. Введите корректный url картинки.
   4. Textarea placeholder «Вставьте html описание курса»
      1. Валидация. Введите минимальное HTML описание курса. Однако значение выставлено в ноль для, того что бы, пользователь мог внести описание позже.
   5. Кнопка «Добавить курс». После нажатия должен сгенерироваться id данной карточки товара и перенаправить на страницу курсов, где происходит отрисовка всех курсов.

UC-8 Корзина:

1. Блок навигации описанный в UC3 блок footer описанный в UC2.
2. Блок корзины состоит:
   1. Если в корзине нет добавленных курсов то выводится параграф «Корзина пуста».
   2. При добавлении курса в корзину формируется таблица которая состоит из:
      1. Заголовков:
         1. Название курса
         2. Количество
         3. Действие
      2. Строк:
         1. Добавляется название курса
         2. Пользователь имеет право заказать несколько единиц одного и того же курса. Счетчик должен проанализировать количество добавленных единиц и отобразить их в графе количество.
         3. В графе действие находится кнопка «Удалить» при нажатии на которую должен срабатывать счетчик и пересчитать количество добавленных курсов и их единиц (одноименных курсов) в корзину.
   3. Цена. При любом срабатывании счетчика цена перерассчитывается.
   4. Кнопка «Сделать заказ». При ее нажатии формируется карточка заказа и добавляется на страницу «Заказы».

UC-9 Заказы:

1. Карточка заказов состоит из:
   1. Если пользователь ничего не добавил в заказы выводится параграф «Заказов нет».
   2. Span Заказ и id заказа.
   3. Дата и время заказа.
   4. Имя и email пользователя.
   5. Список и количество единиц заказываемых курсов.
   6. Цена.

UC-10 Профиль:

1. Блок навигации описанный в UC3 блок footer описанный в UC2.
2. Профиль состоит из:
   1. Аватар.
      1. Параграф «Аватара нет» если пользователь не загрузил фотографию.
   2. Параграфа «Ваш email» где отображаю email который был указан при регистрации пользователя.
   3. Label Ваше имя.
   4. В input, отображаю имя, указанное при регистрации пользователя.
   5. Кнопка «Выберите файл» для загрузки картинки из файловой системы пользователя.
   6. Input в котором отображается имя и расширение загружаемого файла.
   7. Кнопка «Изменить» при нажатии на которую происходят изменения которые внес пользователь а именно: изменил имя, выбрал файл для загрузки.

UC-11 Выйти:

1. Кнопка «Выйти» позволяет пользователю закончить сессию и выйти из системы интернет магазина.
2. Кнопка «Войти» перенаправляет пользователя на станицу логина где пользователь может вновь авторизоваться и зайти в систему интернет магазина.

Такой последовательный вариант описания требований, в виде пунктов и подпунктов сильно упрощает работу. Данный вариант я считаю лучшим т.к некоторые детали в блок схеме было бы рисовать крайне сложно.

По этим же сценариям можно проводить тестирование системы. И если на каком – то этапе возникают сложности, тестировщик может указать в каком пункте или подпункте того или иного пользовательского сценария баг.

* 1. Обзор и обоснование выбора инструментальных средств
* **Visual Studio Code (**<https://code.visualstudio.com/>**).**

Visual Studio code – это легкий и мощный редактор исходного кода. При правильной настройке его можно превратить в настоящую IDE. Он поставляется со встроенной поддержкой JavaScript, TypeScript и Node.js и имеет богатую экосистему расширений для других языков таких как С++, С#, Java, Python, PHP, Go и сред выполнения таких как .NET и Unity.

Как и многие другие редакторы кода, VS Code использует общий пользовательский интерфейс. В панели слева показывается древовидная иерархия папок и файлов – это структура проекта. Справа показано содержимое открытых файлов.

Интерфейс VS Code делится на пять основных областей.

* Редактор – основная область для редактирования файлов.
* Боковая панель – содержит различные виды что бы помочь при работе с проектом
* Строка состояния – информация об открытом проекте и файлах
* **Панель активности** - расположена в крайнем левом углу, она позволяет переключаться между представлениями и дает дополнительные индикаторы, зависящие от контекста, например, количество исходящих изменений, когда Git включен.
* **Панели** – Внизу я могу отображать различные панели под областью редактора для вывода или отладки информации, ошибок и предупреждений или встроенного терминала. Панель также можно перемещать вправо для увеличения вертикального пространства.

Для своего проекта я использую встроенный терминал для установки различных зависимостей, библиотек и т.д.

Для настройки VS code я использую следующие плагины:

1. Auto Close Tag - автоматически добавляет закрывающий тег при вводе закрывающей скобки открывающего тега. Область применения языки гипертекстовой разметки HTML/ XML, а так же для PHP, Vue, JavaScript, TypeScript, JSX и т.д.
2. Auto Complete Tag – объединяет функции автоматического закрытия тегов и автоматического переименования тегов.
3. Bracket Pair Colorizer – подсвечивает скобки разными цветами, что прекрасно помогает при навигации, визуализации вложенных функций. Можно настроить список цветов, а также определенный цвет для лишних скобок. Данная функция плагина прекрасно помогает при поиске ошибочно напечатанных скобок.
4. Code Spell Checker – данный плагин проводит базовую проверку орфографии которая хорошо работает с кодом Camel Case.
5. CSS Peek – Данный плагин настраивает навигацию между классами прописанными в HTML, JS, JSX и тд и классами прописанными в каскадной таблице. Для быстрого поиска классы в каскадной таблице нажимаю правой кнопкой мыши на классе прописанном в HTML и выбираю *Перейти к определению*, происходит переход к нужному классу в каскадной таблице стилей.
6. CSSTree validator – плагин проводит проверку CSS в соответствии со спецификациями W3C и реализациями браузера. Расширение подсвечивает неправильные свойства и их значения.
7. EditorConfig for VS Code- данный плагин в фоновом режиме проверяет кодировку каждого открытого файла. Контролирует табуляцию.
8. ESLint – статически анализирует код JavaScript. Находит ошибки. Многие проблемы, обнаруженные этим плагином, могут быть исправлены автоматически. Так же можно написать собственные правила, которые работают со встроенными правилами.
9. Git History Diff- данный плагин позволяет посмотреть детальную историю коммитов. Посмотреть какие изменения были внесены в файл, проект. Работа с ветками проекта и многое другое.
10. HTML CSS Support – этот плагин автоматически дополняет названия id классов на основе определений найденных в рабочей области или внешних файлов.
11. JavaScript (ES6) – это расширение содержит фрагменты кода для JavaScript в синтаксисе ES6.
12. JS Refactor – это инструмент автоматического рефакторинга Javascript для кода Visual Studio, созданный для упрощения и оптимизации процесса разработки. Он предоставляет обширный список автоматизированных действий, включая обычно необходимые: метод извлечения, извлечение переменной, встроенная переменная и псевдоним для переименования встроенного кода VS. JS Refactor также поддерживает множество общих фрагментов и связанных действий для упаковки существующего кода в общие выражения блоков.
13. Prettier - Code formatter - средство форматирования кода. Упорядочивает код.
14. SASS – подсвечивает синтаксис SASS, автоформатирует и заполняет код.
15. W3C Validation - Расширение для включения поддержки проверки W3C библиотекой. Это приложение требует установленной JDK. После установки нужно настроить переменные среды.
16. PlantUML – С моей точки зрения это наиболее адекватный инструмент для создания UML диаграмм. Смысл этого инструмента прост, он позволяет задавать диаграммы в виде текста, описывающего элементы и связи между ними. Графический вид создается автоматически. Плюсы PlantUML:
17. Диаграммы задаются в виде текста, а значит в любом текстовом редакторе можно найти требуемый текст в наборе файлов и заменить его.
18. Рефакторинг вполне удобен.
19. Текстовый формат дает возможность организовывать групповую работу и отслеживать изменения в системе контроля версий.

* **Node.js (**<https://nodejs.org/en/>**).**

Платформа NodeJS – это программная платформа, основанная на движке v8 (транслирующем JavaScript в машинный код) превращая JavaScript из узкоспециализированного языка в язык общего назначения.

Делает он это с некоторыми особенностями, которые ранее не были учтены другими разработчиками. Он очень эффективно и очень быстро транслирует JavaScript т.е. этот движок понимает какие куски кода нам не нужны, и он не тратит время на их выполнение.

Раин Даль в 2009 году, после нескольких лет экспериментов, разработал очень эффективный серверный язык, который отличается по парадигме от старых подходов т.е. были однопоточные, многопоточные среды разработки. Он разработал серверный язык на событиях. JavaScript событийно-ориентированный язык. По сути, он вытащил этот язык из браузера и написал пару плагинов на С++. В итоге, он получил движок, который транслирует JS отдельно от браузера. При этом он добавил различные устройства ввода вывода. Добавил определенный функционал при работе с файлами, для создания различных серверов и прочего.

NodeJS – это просто движок V8 который транслирует JS. Поверх него написана оболочка на C++. Это абсолютно отдельная сущность, которая способна понимать JS, причем делает это очень эффективно, и она может работать где угодно. А с учетом того, что есть C++ он может запускаться где угодно. Он не ограничен возможностями браузерного JS.

К примеру. В 2009 году, когда NodeJS только вышел, про модульность в JS не было и речи. Никто не знал про ES6 import и ничего такого не было.

В NodeJS появились модули, и разработчики смогли писать JS в разных файлах, причем делать это очень просто и нативно. Появилась возможность работать с файлами, создавать сервера.

Сейчас, по сути, из-за того, что JS использует событийно-ориентированный подход – это очень оптимизирует данную платформу, выполнение кода, потому что выполняются только те куски, которые нужны. У него есть концепт, который называется EventLoop – это определенная очередь куда складываются события и за счет этого разработчики получают очень оптимизированную платформу для работы с JS.

Плюсы использования NodeJS. Во - первых это тот же самый JS. JS во front-end – это монополист, потому что нет конкурента языку JS на front-end. Даже если сайт будет на php, все равно на front-end будет использоваться JS. Плюс в том, что, зная один язык я могу писать серверную часть. Node.js использует мало системных ресурсов в силу своей однопоточности, тогда как традиционные веб-серверы многопоточны. Помимо этого, NodeJS позволяет использовать различные сборщики пакетов, к примеру Webpack который я использую в своем проекте и который будет рассмотрен чуть ниже.

А также позволяет использовать менеджер пакетов npm (аббр. node package manager) – это стандартный менеджер пакетов, автоматически устанавливающийся вместе с NodeJS. Он используется для скачивания пакетов из облачного сервера npm, либо для загрузки пакетов на сервера.

Основная его задача – это менеджмент различных модулей. Все приложения NodeJS начинаются с данного инструмента. Через npm доступно более 46000 пакетов для скачивания.

* **JavaScript**

Одна из главных причин популярности Node.js такова: для нее программируют на знакомом большинству программистов языке – JavaScript. Раньше, как минимум нужно было в совершенстве знать как минимум два языка программирования: JavaScript для клиентской части и что – то еще, например PHP или Ruby – для серверной.

По сравнению с первоначальным замыслом разработка на JavaScript заметно изменилась. Трудно представить какой длинный путь прошел этот язык от своей исходной реализации в браузере Netscape до современных мощных машин, таких как созданный Google механизм V8. Это был тернистый путь переименований, объединений и конечной стандартизации в виде ECMAScript. Сегодняшние возможности превзошли самые смелые мечты первых разработчиков.

Но не смотря на успех и популярность JavaScript, его по-прежнему зачастую недооценивают. Лишь немногие знают, что это очень эффективный и динамичный объектно-ориентированный язык. Однако у него имеются и более развитые свойства как прототипное наследование, использование модулей и пространств имен.

В последнее время JavaScript – машины и браузеры приобрели такую мощность, что создание полноценных приложений на JavaScript стало не только реально выполнимой, но и весьма популярной задачей. Такие приложения как Gmail и Google Maps, создали предпосылки до совершенно иного взгляда на веб-приложения, и пользователи стали еще требовательнее. Компании начали нанимать разработчиков программ на JavaScript на постоянную работу. JavaScript перестал быть вспомогательным инструментом, предназначенным только для написания простых сценариев и подсистемы проверки приемлемости данных формы. Теперь он стал вполне самостоятельным языком, имеющим право на реализацию своего полного потенциала.

Этот рост популярности означает, что было создано множество новых JavaScript приложений. К сожалению, вероятно, из – за сложной истории этого языка многие из этих приложений весьма убоги. По каким-то причинам, как только дело доходит до JavaScript, все общепринятые шаблоны и богатый практический опыт куда-то улетучиваются. Разработчики игнорируют такие архитектурные модели как схему модель-представление-контроллер (Model View Controller, MVC), создавая в своих приложениях некую небрежную связь из файлов и папок. MVC будет рассмотрена в главе 3.1 «Разработка архитектуры программного продукта».

* **Framework Express (**<https://expressjs.com/>**).**

Express выпущен в 2009 году Т.Дж. Головайчуком и с тех пор стал самым популярным фреймворком для Node.js. Его исходный код открыт и разрабатывается более чем 100 участниками проекта.

Поскольку Node.js - платформа которая не ограничивает то, как ее следует настраивать и использовать. Это одна из ее сильных сторон. Но при создании сайтов и веб-приложений каждый раз приходится выполнять несколько распространенных задач. Express – фреймворк веб-приложений для Node.js, спроектирован для выполнения таких задач проверенным и повторяющимся способом.

Как уже отмечалось Node.js – это платформа, а не сервер. Это дает возможность проявить при настройке сервера свои творческие способности и заставить его выполнять то, что другие серверы делать не способны. Но это же делает более трудным создание и запуск простого сайта.

Express делает эти сложности не существенными благодаря настройке веб сервера для прослушивания входящих запросов и возврата соответствующих ответов. Дополнительно он задает структуру каталогов. Один из этих каталогов настроен для обработки статических файлов не блокирующим образом. Например если папку, в которой находятся css файлы, не сделать статической, клиент будет ждать пока к данной папке будут происходить запросы на обработку.

Данную возможность можно сконфигурировать в Node.js однако в Express это делается автоматически.

Замечательная особенность Express — по-настоящему простой интерфейс для маршрутизации входящего URL к соответствующему фрагменту кода. Неважно, будет ли этот код выдавать статическую веб-страницу, выполнять чтение из БД или записывать в нее, — интерфейс прост и единообразен.

Здесь Express фактически уменьшил сложность выполнения всего этого в .js, ускоряя тем самым написание кода и облегчая его поддержку.

В Node.js обычно происходит много ответов на много запросов к приложению путем отправки браузеру HTML-кода. Express упрощает эту задачу по сравнению с нативным Node.js.

Express обеспечивает множества различных шаблонизаторов, таких как Handlebars который я использую в своем проекте, упрощающих создание страниц HTML с помощью повторно используемых компонентов, а так же данных из приложения. Express собирает их вместе и выдает браузеру ввиде HTML-кода.

Будучи однопоточным, Node.js не запоминает посетителей между запросами. У него нет отдельной области оперативной памяти, выделенной для конкретного пользователя, он просто видит ряд запросов HTTP. HTTP- протокол без сохранения состояния, так что в нем отсутствует понятие состояния сеанса.

У Express имеется решение и этой задачи. Express позволяет использовать сеансы что позволяет идентифицировать отдельных посетителей на протяжении нескольких запросов и страниц.

Будучи надстройкой над Node.js, Express помогает вам и предоставляет надежную отправную точку для создания веб – приложений. Он снимает множество сложностей и повторяющихся задач, о которых большинство из нас не должно или не хочет беспокоится.

* **MongoDB (**<https://account.mongodb.com/>**).**

Способность хранить и использовать данные жизненно важна для большинства приложений. MongoDB подобно Node.js славится своей быстротой работы и масштабируемостью.

Обычные реляционные базы данных состоят из таблиц, в которых содержатся столбцы. Обычно столбец задает имя и тип данных, а каждая строка задает отдельную сущность.

В MongoDB это не так. MongoDB – документоориентированная БД. Строки в ней присутствуют, но столбцов как таковых нет. Вместо столбцов, описывающих, что должно быть в строке, каждая строка представляет собой документ, который содержит как данные так и их описание и располагается без соблюдения порядка. Такой менее структурированный подход означает, что коллекции документов могут содержать самые разнообразные данные.

Хранение данных в MongoDB схоже с хранением данных в формате JSON (JavaScript Object Notation). Для хранения данных в MongoDB применяется формат, который называется BSON (БиСон) или сокращение от binary JSON. BSON позволяет работать с данными быстрее: быстрее выполняется поиск и обработка.

MongoDB отличается от многих других документоориентированных БД поддержкой вторичной индексации и способностью работать с запросами, обладающими широкими возможностями. Это значит, что можно создавать индексы не только по полю уникального идентификатора, а так же что запросы к проиндексированным полям выполняются намного быстрее. Можно создавать весьма сложные запросы к базе данных MongoDB – не на уровне огромных команд SQL с соединениями повсюду, но с возможностями, которых достаточно для выполнения большинства пользовательских сценариев.

MongoDB не должна использоваться как транзакционная база данных. Транзакционная БД может выполнять несколько отдельных операций как одну транзакцию. Если какая-либо операция в транзакции завершится неудачей, то вся транзакция завершается неудачей и ни одна из операций не выполнится. MongoDB работает не так. Она рассматривает каждую операцию отдельно, и если какая-либо операция завершится неудачей, то завершится неудачей только она, а остальные операции будут выполняться.

* **Mongoose (** [**https://mongoosejs.com/**](https://mongoosejs.com/)**).**

Гибкость MongoDB относительно хранимых данных в документах – отличное качество для БД. Но большинство приложений требуют для своих данных определенной структуры. Разумнее всего будет определить структуру данных в самом приложении, а не в базе данных.

С этой целью компания, разработавшая MongoDB, создала Mongoose. По их собственным словам Mongoose обеспечивает «изящное моделирование объектов MongoDB для Node.js».

Моделирование данных в контексте Mongoose и MongoDB – это определение того, какие данные могут, а какие – должны присутствовать в документе. При хранении данных пользователя может потребоваться возможность сохранить имя, фамилию и адрес электронной почты, причем адрес почты должен быть не повторяющимся. Эта информация задается в схеме, которая используется как основа для модели данных.

Помимо моделирования данных, Mongoose добавляет к возможностям MongoDB множество возможностей, полезных при создании веб-приложений. Mongoose облегчает управление подключениями к базе данных MongoDB, а так же сохранение и чтение данных. Благодаря Mongoose приобретается возможность добавления проверки данных на уровне схемы, гарантируя, что в БД можно сохранять только корректные данные.

* **Git (**<https://github.com/>**).**

Сохранять код на своем компьютере или сетевом диске, конечно, замечательно, но так сохраняется только последняя версия. При этом получить доступ к ней может лишь разработчик писавший этот код.

Git – распределенная система контроля версий и управления исходным кодом. Это значит что несколько человек могут работать с одной базой исходных текстов одновременно с различных компьютеров из различных сетей. Они могут работать совместно, сохраняя и записываю все изменения. Также возможно при необходимости откатиться к предыдущему состоянию.

Для разработки «Информационной системы интернет магазина» Git потребуется по двум причинам.

Во – первых исходный код приложения будет храниться на GitHub, для различных промежуточных этапов разработки будут созданы различные ветви. А различные версии приложения будут сохранены в коммитах.

Во – вторых будет применяться в качестве развертывания приложения на действующий веб – сервер, что бы его мог увидеть весь мир. Для хостинга будет использоваться Heroku.

* **Heroku (**<https://www.heroku.com/>**).**

Хостинг приложений Node.js может оказаться не простой задачей. Многие провайдеры не поддерживают Node.js. Однако некоторые провайдеры могут установить его при необходимости, но в целом такие серверы не настроены в соответствии со специфическими потребностями Node.js.

Heroku – один из ведущих провайдеров хостинга для приложений Node.js, и у него имеется великолепный бесплатный пакет, которым я и воспользуюсь.

Приложения на Heroku, по сути дела, представляют собой репозиторий Git, что делает процесс публикации чрезвычайно простым. Как только все настроено, можно опубликовать приложение в промышленной среде с помощью одной единственной команды: $ git push heroku master.

* **SengGrid (**<https://app.sendgrid.com/>**).**

В PHP есть функция mail которая позволяет с помощью сервера отправлять email на любой почтовый адрес. Но в Node.js это не так. В Node.js для того что бы отправлять какие либо email нужно пользоваться сторонними сервисами которые предоставляют данную возможность т.е. через определенные скаченные пакеты, которые позволяют сделать подключение с сервисом и уже через сервис работать с email.

Одним из таких сервисов является SendGrid. Плюсом данного сервиса является то что до определенного плана он может быть бесплатен. При этом он крайне прост в настройке и удобен в использовании.

* **Webpack (**<https://webpack.js.org/>**).**

Webpack – это статический модульный сборщик. В проекте он обрабатывает все файлы и ресурсы как модули. При этом сборщик опирается на граф зависимостей, в котором описывается взаимосвязь модулей с помощью ссылок (операторы require и import).

Таким образом, webpack статически перемещается во всем модулям для построения графа и использует его для генерации одного бандла.

Бандл – это файл JavaScript, содержащий код всех модулей проекта и объединенных в правильном порядке. Когда webpack создает граф зависимостей, он не выполняет исходный код а объединяет модули и их зависимости в бандл.

Базовые понятия Webpack.

Entry – модуль, который используется для построения внутреннего графа зависимостей. С его помощью webpack определяет, от каких модулей и библиотек зависит точка входа. Затем включает их в граф, пока не останется ни одной зависимости. По умолчанию для свойства entryустановлено значение ./src/index.js. Но можно указать другой модуль в файле конфигурации сборщика.

Output – это свойство указывает, где webpack должен сохранять бандл и как называть его файл или файлы. Значением по умолчанию является ./dist/main.js для основного бандла и ./dist для других сгенерированных файлов.

Загрузчики. По умолчанию webpack понимает только файлы JavaScript и JSON. Что бы обработать другие типы файлов и конвертировать их в модули, сборщик использует загрузчики. Например, загрузчик может трансформировать файлы из языка CoffeeScript в JavaScrip или встроенные изображения в URL-адреса. С помощью загрузчиков можно даже импортировать CSS-файлы прямо из модулей JavaScript.

Плагины. Они используются для задач, которые не могут выполнять загрузчики.

**Режимы.**  Webpack позволяет настроить режим на **development, production**или **none**. Благодаря этому он может использовать встроенные оптимизации для каждой среды. По умолчанию установлено значение Режим **none** означает, что все опции оптимизации по умолчанию будут отключены.

* **Babel (**<https://babeljs.io/>**).**

Babel – это JavaScript компилятор, но в отличии от классических компиляторов которые переводят код в машинный код, babel переводит EcmaScript2015+ (самая свежая спецификация) в более старую спецификацию, например в EcmaScript5. Однако на этом он не останавливается. В своем проекте я использую препроцессор Sass. И аналогично тому как Babel компилирует современные спецификации JavaScript в более старые, он компилирует исходный код Sass в обычную каскадную таблицу стилей css.

* **Handlebars (**<https://handlebarsjs.com/>**).**

Handlebars (а точнее расширение Mustache, еще одного распространенного шаблонизатора) – это шаблонный процессор, который динамически генерирует HTML страницу, что экономит время на ручном обновлении. Причиной его использования стало то, что данные регулярно меняются.

Существуют две основные причины, по которым по которым создавать шаблон для сайта является хорошим experience (опытом).

Прежде всего, создание шаблона побуждает отделять логический код от фактического представления, помогая придерживаться шаблона View/Controller. Во – вторых, шаблоны сохраняют код чистым и поддерживаемым, что, в свою очередь, делает процесс обновления сайта быстрым.

Handlebars генерирует HTML используя структуру JSON и запускает ее через шаблон. Конечно, вызовы AJAX могут возвращать фрагменты HTML, которые можно вставить в DOM как есть, но Handlebars на стороне клиента дает возможность получать результаты вызова AJAX в виде данных в формате JSON и форматировать их для нужд приложения. Шаблоны написаны в обычном HTML и набиты заполнителями. Заполнители позволяют при необходимости вводить данные.

Пример вывода картинки в карточке товара:

**<div class="card-image">**

**<img src="{{img}}" alt="{{title}}" />**

**</div>**

Атрибут {{img}} – это картинка выводимая в карточке товара. Атрибут {{alt}} выводит описание которое позволяет поисковым роботам «увидеть» изображение и проиндексировать его. Этот заполнитель соответствует свойству в структуре данных JSON.

Так же Handlebars поставляется с некоторыми встроенными помощниками. Они позволяют работать с более сложными данными. Один из таких помощников оператор each c помощью которого выполняется итерация массива с последующим выводом каждого элемента массива тем самым генерируя динамический HTML.

Пример страница «Заказы» интернет-магазина курсов:

**{{#each orders}}**

**<div class="orders">**

**<div class="order\_\_conteiner">**

**<span class="orders-title">Заказ:&nbsp;<small>{{\_id}}</small></span>**

**<p class="orders-date">{{date}}</p>**

**<p><em>{{user.userId.name}}</em>&nbsp;&nbsp;({{user.userId.email}})</p>**

**<ol>**

**{{#each courses}}**

**<li>**

**{{course.title}}(x<strong>{{count}}</strong>)**

**</li>**

**{{/each}}**

**</ol>**

**<p>Цена:&nbsp;&nbsp;<span class="price">{{price}}</span></p>**

**</div>**

**</div>**

**{{/each}}**

В Handlebars можно так же получить доступ к вложенным данным (user.userId.name).

Так же присутствуют и другие операторы такие как if, else и др.

Пример запуска приложения «Информационная система интернет магазина» в production или development режиме.

**if (process.env.NODE\_ENV === 'production') {**

**module.exports = require('./keys.prod');**

**} else {**

**module.exports = require('./keys.dev');**

**}**

Handlebars дает возможность создавать своих пользовательских помощников в виде функций. Есть два типа помощников:

Функциональные помощники – это в основном обычные функции, которые после регистрации могут быть вызваны в любом месте шаблона. Handlebars записывает возвращаемое значение функции в шаблон.

Блок помощники похожи по своему характеру на if, each и т.д. Они позволяют изменить контекст того, что внутри.

Пример регистрации функции helpers «Информационной системы интернет магазина»:

**module.exports = {**

**ifeq(a, b, options) {**

**if (a == b) {**

**return options.fn(this);**

**}**

**return options.inverse(this);**

**},**

**};**

* **Препроцессор SCSS (**<https://sass-lang.com/>**).**

Препроцессор SCSS – является языком надстройкой над языком разметки css т.е. это лишь компилированный язык для каскадной таблицы стилей. SCSS позволяет создавать много дополнительного функционала не доступного в CSS: создание переменных, функций, циклов, задание условий и т.д. Так же здесь имеется удобная вложенность. На выходе получается красивый, легко поддерживаемый код с которым легко работать в отличие от обычного CSS.

1. Проектирование
   1. Разработка архитектуры програмного продукта

Секрет создания больших JavaScript приложений кроется не в создании огромных приложений как таковых, а в том, что нужно разбить приложение на ряд практически независимых компонентов. Разработчики часто допускают ошибку, создавая приложения, содержащие множество взаимосвязей, с огромными и прямолинейными JavaScript файлами, генерирующими массу HTML-тегов. Приложения такого сорта трудно обслуживать и расширять, поэтому их нужно избегать любой ценой.

Существенно повлиять на конечный результат может внимание к структуре приложения, проявленное в самом начале разработки. JavaScript полноценно объектно-ориентированный язык. Поэтому самым лучшим вариантом будет применение и использование классов, наследования, объектов и схемы точно также как, как это делается при создании приложений на других языках.

MVC – это шаблон проектирования, разбивающий приложение на три части: Данные (Model, модель), уровень их отображения (View, представление) и уровень взаимодействия с пользователем (controller, контроллер).

«Пример модели MVC информационной системы интернет магазина кусов приведен в приложении А»

Иными словами, события протекают следующим образом.

1. Пользователь взаимодействует с приложением.
2. Вызываются обработчики событий контроллера.
3. Контроллер запрашивает данные из модели, предоставляя их представлению.
4. Представление предоставляет данные пользователю.

Модели – это фундамент проекта. В модели жизненно важно не загрязнять код любыми презентациями или кодом взаимодействия с пользователем.

В идеальном мире модели и уровень хранения данных должны быть полностью разделены. Конечно это достижимо – но обычно определенной ценой. Очень часто логика в моделях является серьезно зависимой от постоянства, и разделение двух уровней может быть куда большей проблемой.

В своем приложении я пошел по пути наименьшего сопротивления – использовал Mongoose (который относится к MongoDB) для определения своих моделей. Т.е. я создал в корне приложения каталог с названием models, что бы я мог хранить все свои модели там. Всякий раз когда у меня появляется логика для внедрения или данные для хранения, я делаю это в файле в пределах каталога models. Например, я храню данные курса и логику в файле, названном models/course.js

Пример кода model/courses.js представлен в ПРИЛОЖЕНИИ Б.

Модели представления дают возможность сохранять модель обобщенной и в тоже время предоставляющей осмысленные данные к представлению.

Возьмем предыдущий пример. У меня есть модель под названием Course. Сейчас я хочу создать представление, показывающее данные карточки курса. Однако есть данные которые я не хочу показывать потенциальному покупателю. К таким данным в модели Course относится userId который взят из другой модели User. Покупателю абсолютно не нужно знать id пользователя который создал и добавил для продажи данный курс.

Пример кода модели представления views/courses.hbs представлен в ПРИЛОЖЕНИИ Б

Контроллеры отвечают за обработку пользовательского взаимодействия и выбор соответствующих представлений к отображению, базирующихся на этом пользовательском взаимодействии. Звучит очень похоже на маршрутизацию запросов не правда ли? В действительности единственной разницей меду контроллером и маршрутизатором является то, что контроллеры обычно группируют соответствующую функциональность.

Пример кода контроллера добавления курса routes/add.js представлен в ПРИЛОЖЕНИИ Б

Данный контролер ответственен за отображение формы для ввода необходимых полей для создания нового курса. За занесение заполненных полей в БД, а так же создание карточки курса.

Одну модель могут обрабатывать несколько контроллеров. К примеру методы редактирования, удаления и удаления карточки курса реализованы в контроллере routes/courses.js.

Пример кода контроллера routes/courses.js представлен в ПРИЛОЖЕНИИ Б

Как и многие парадигмы или шаблоны в программировании, MVC – это скорее общая концепция, нежели специфическая техника. Подход который я использовал, в основном таков: я лишь делал его более формальным, назвав обработчики маршрута контроллером и отделив маршрутизацию от функциональности. Так же я представил концепцию модели представления, которую я считаю весьма важной для сохранения целостности модели.

* 1. Проектирование структур хранение данных

В данном разделе необходимо представить результаты проектирования базы данных или другого способа хранения данных предметной области. В разделе необходимо привести даталогическую модель базы данных, а также описание других элементов, реализующих хранение данных предметной области.

Проектирование базы данных – это процедура выбора наилучшего представления набора данных с учетом возможностей СУБД, природы данных и требований приложения. Принципы проектирования реляционных баз данных уже давно устоялись. Однако разработчики приходящие к MongoDB из мира РСУБД не приятно удивляются отсутствием твердых правил проектирования схемы.

Структура единиц данных в MongoDB представляет собой хранилище ключей и значений. В хранилище ключей и значений, имеются ключи, указывающие на аморфные значения. В MongoDB структурной единицей является BSON – документ.

Определившись со структурным делением данных, нужно понять, как ими манипулировать. В РСУБД предлагаются произвольные запросы и операции соединения. В MongoDB произвольные запросы так же поддерживаются, но соединение не поддерживается.

MongoDB, хранилище документов, а не традиционная табличная БД со строками и столбцами. Это дает MongoDB колоссальную свободу и гибкость, но иногда желательна, то есть необходима структура данных.

Возьмем, например, карточку товара курса.



Рисунок 3.5 – Карточка курса.

У карточки товара имеются четко определенные требования к данным и их структуре. Эти элементы данных нужны странице для всех мест, и у записей данных для каждого места, должна быть согласованная структура именования. Без этого приложение не сможет найти данные и использовать их. В терминах архитектуры MVC работа с данными начинается в представлении, а затем представление передает их на шаг назад, в контроллер. После этого остается переместить их еще на один шаг назад, туда, где они должны находится, - в модель.

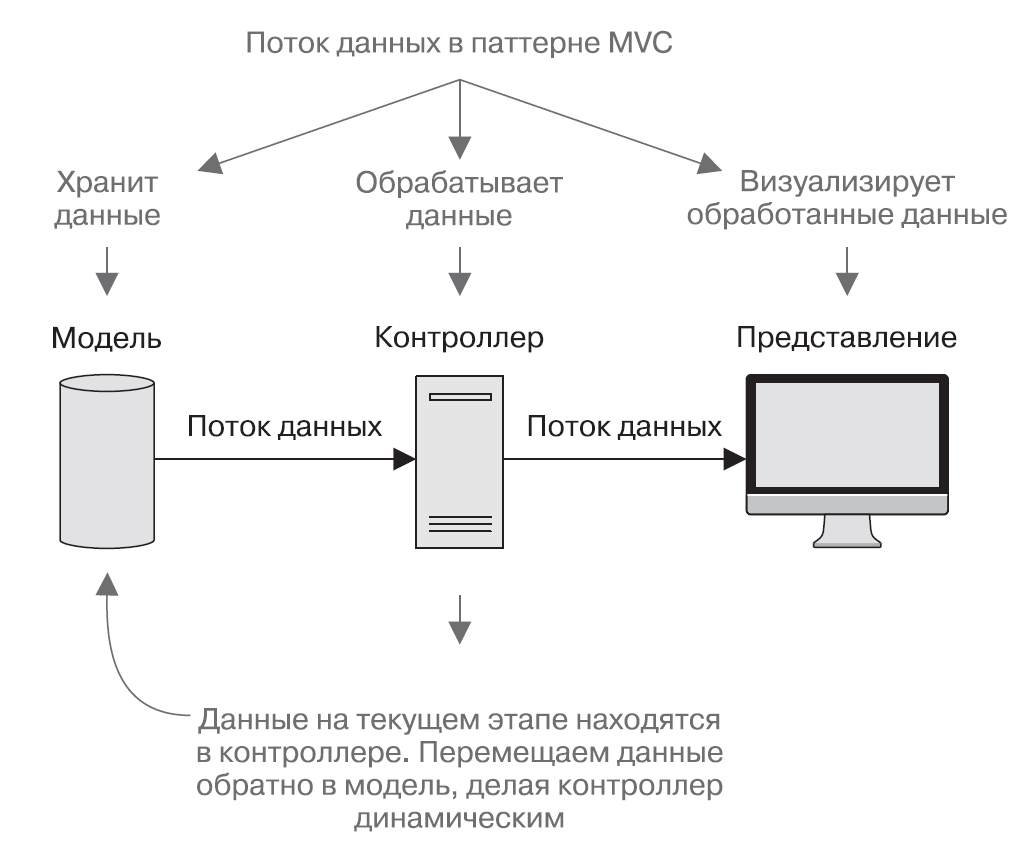


Рисунок 3.6 – Поток данных в паттерне MVC.

Перемещение данных назад по потоку MVC, шаг за шагом, заключается в том, что благодаря этому выкристаллизовываются требования к структуре данных. Это гарантирует точное отражение структурой данных нужд приложения. Если начать сначала описывать модель, то это закончится гаданием о том, как будет выглядеть приложение и как оно будет работать.

Говоря о моделировании данных, мы на самом деле описываем свои пожелания относительно структуры. В приложении можно создавать описание вручную и управлять ими вручную и делать всю тяжелую работу самостоятельно, или использовать Mongoose, позволив ему делать тяжелую работу за нас.

Mongoose был специально создан в качестве объектно-документного средства моделирования (Object-Document Modeler (ODM)) MongoDB для приложений NodeJS. Один из ключевых принципов – возможность управления моделью данных изнутри приложения. Не нужно возится непосредственно с базами данных, внешними фреймворками или реляционными подпрограммами отображения – можно просто описать модель данных, находясь в комфортных условиях своего приложения.

* В MongoDB каждая запись в БД называется документом.
* В MongoDB набор документов называется коллекцией.
* В Mongoose описание документа называется схемой.
* Каждая отдельная сущность данных в схеме называется путем.

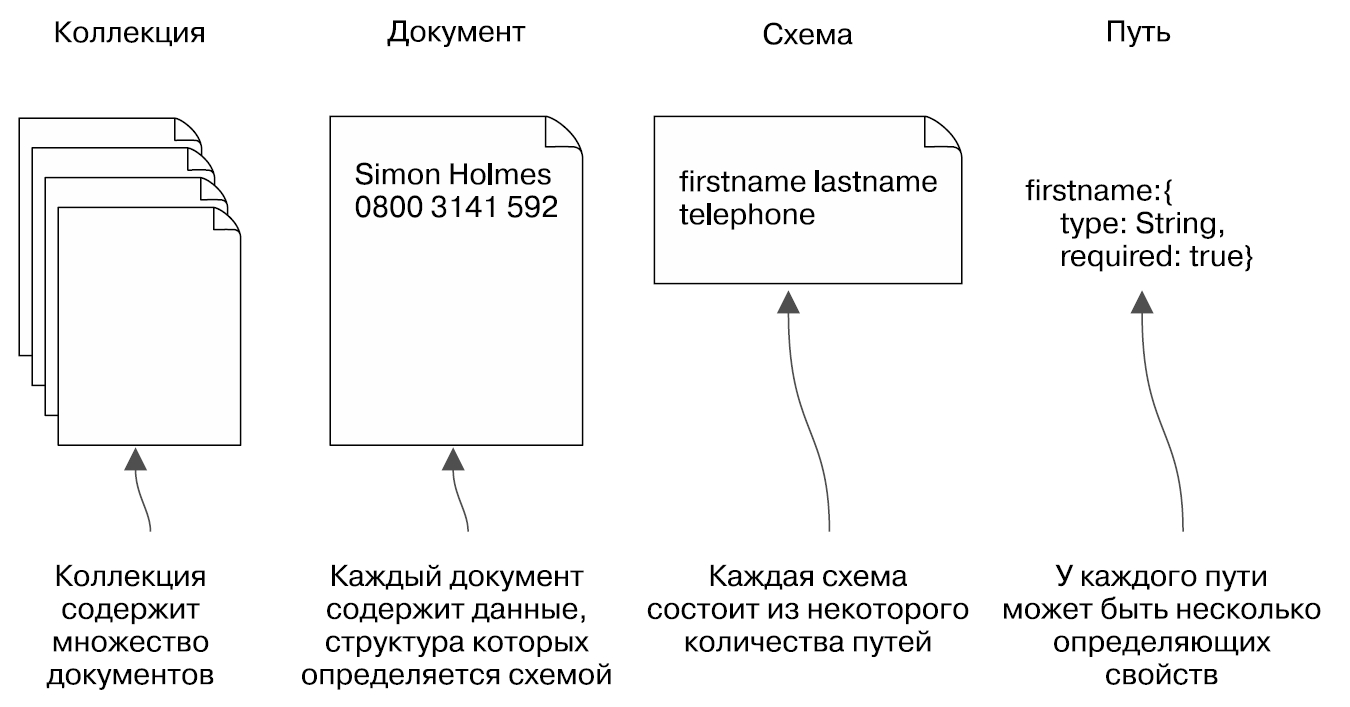


Рисунок 3.7 – Отношение между коллекциями, документами, схемами и путями в MongoDB и Mongoose на примере визитных карточек.

В MongoDB структурной единицей является BSON – документ.

Структурной единицей которого в MongoDB является хранилище ключей и значений. В хранилище ключей и значений, имеются ключи, указывающие на аморфные значения.

Определившись со структурным делением данных, нужно понять, как ими манипулировать. В РСУБД предлагаются произвольные запросы и операции соединения. В MongoDB произвольные запросы так же поддерживаются, но соединение не поддерживается.

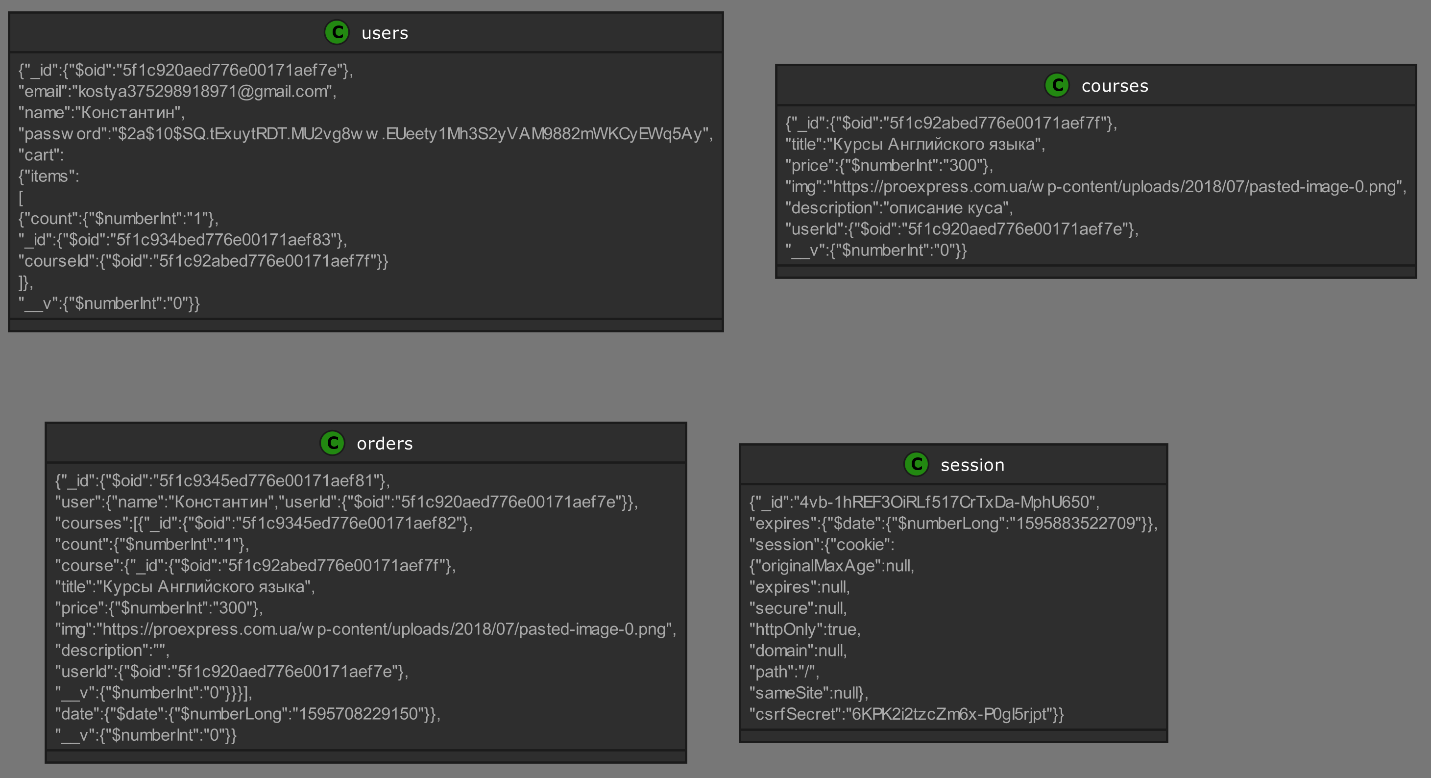


Рисунок 3.8 - пример хранения данных в MongoDB на примере «Информационной системы интернет магазина».

Рассмотрим на примере модели courses в MongoDB как может выглядеть схема Mongoose.

const courseSchema = new **Schema**({

  title: {

**type**: **String**,

    required: true,

  },

  price: {

**type**: **Number**,

    required: true,

  },

**img**: **String**,

  description: {

**type**: **Object**,

  },

  userId: {

**type**: **Schema**.**Types**.**ObjectId**,

    ref: 'User',

  },

});

Как видно, схема очень напоминает, сами данные. Она описывает имя для каждого пути данных и содержащихся в нем тип данных. В данном примере я просто объявляю title и img как строковое значение, price числовое, description объявляю объектом, userId обращаюсь к схеме пользователя в которой указан уникальный id пользователя.

В схеме Mongoose я не объявляю путь \_id. \_id – это уникальный идентификатор, первичный ключ для каждого документа. MongoDB автоматически генерирует этот путь при создании каждого документа и присваивает ему уникальное значение objectId. Значение спроектировано с расчетом на то, чтобы оставаться уникальным при любых условиях посредством сочетания времени, прошедшего с момента начала отсчета времени Unix, с идентификаторами машины и процесса, а также счетчиком.

Базовая структура описания пути представляет собой название пути со следующим за ним объектом свойств. И так путь схемы состоит из двух частей – названия пути и объекта свойств – следующим образом:



Рисунок 3.9 – Базовая структура описания пути.

Типы схем – свойство, задающее тип данных для данного пути. Оно необходимо для всех путей. Если тип – единственное свойство пути, то можно использовать сокращенное описание. Существует восемь типов схем:

* String - любая строка в кодировке UTF-8;
* Number – Mongoose не поддерживает длинные числа или числа с двойной точностью, но допускает расширения для их поддержки с помощью плагинов Mongoose. Поддерживаемого по умолчанию типа достаточно в большинстве случаев;
* Date – обычно возвращается из MongoDB в виде объекта ISODate;
* Boolean – true или false;
* Buffer – для двоичной информации, например изображений;
* Mixed – любой тип данных;
* Array – может быть или массивом данных соответствующего типа, или массивом вложенных поддокументов;
* ObjectId – для уникального ID в пути, отличным от \_id. Обычно используется для ссылок на пути \_id в других документах.
* Во время работы с данными приложение не взаимодействует непосредственно со схемой – взаимодействие происходит через модели.

Модель в Mongoose – это скомпилированная версия схемы. Однократно скомпилированный, отдельный экземпляр модели отображается непосредственно на отдельный документ в БД. Именно благодаря этому взаимному однозначному соответствию модель может создавать, читать, сохранять и удалять данные.

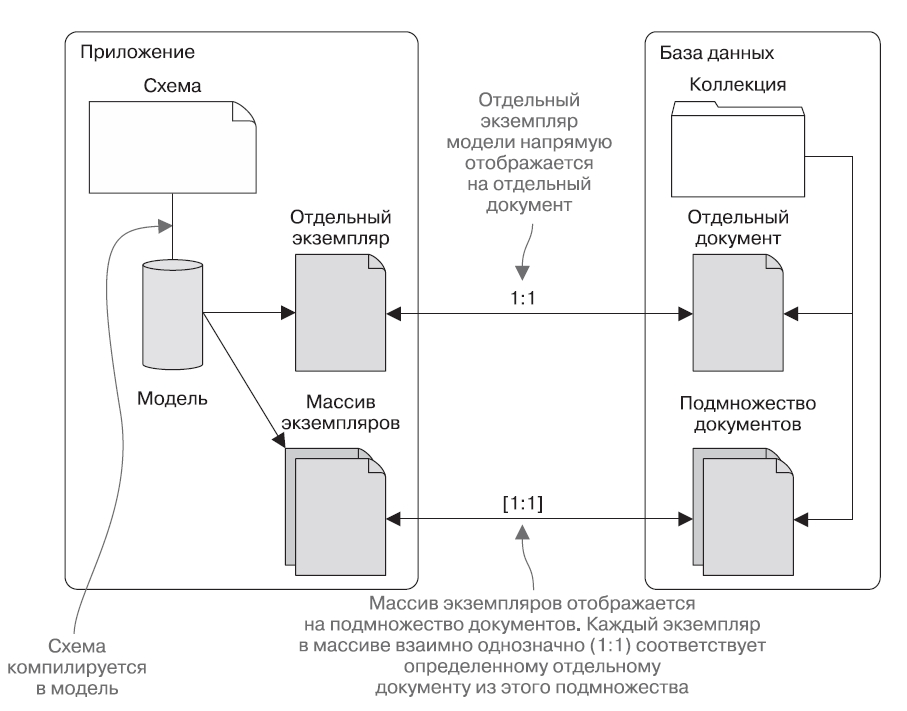


Рисунок 3.10 – Приложение и БД взаимодействуют друг с другом посредством моделей. Отдельный экземпляр модели взаимно однозначно (1:1) соответствует отдельному документу в БД. Благодаря этому соответствию происходит управление созданием, чтением, обновлением и удалением данных.

* 1. Описание реализации вариантов использования

Рассмотрим реализацию алгоритма данных на примере трех UML- диаграмм классов: Создание/добавление курса, редактирование курса и удаление курса.

* Добавление курса.

«Пример добавления курса приведен в приложении А»

Пользователь посылает get запрос, происходит аутентификация пользователя и если пользователь успешно проходит аутентификацию происходит перенаправление на страницу добавления курса.

Если же нет, то происходит перенаправление пользователя на страницу логина.

Страница «Добавить курс» состоит из формы, в которую входят следующие поля: title, price, img, description. При заполнении данных полей происходит валидация формы. Если формы заполняются не должным образом, то пользователь получает сообщение с ошибкой.

При заполнении и отправке форм происходит сверка полей в модели курса, в которой указано обязательны ли данные поля к заполнению. Так же происходит сверка id пользователя который создает курс.

При удачном сценарии происходит добавление курса в базу данных с последующим перенаправлением пользователя на страницу «Курсы».

* Редактирование курса

«Пример редактирования курса представлен в приложении А»

Пользователь посылает get запрос, происходит аутентификация пользователя. Далее происходит проверка query параметра, который отвечает за доступ пользователя к редактированию курса. И если по данному запросу не будет найдено заданное поле allow, то будет происходить перенаправление на страницу «Главная».

Далее производится поиск id курса после чего происходит перенаправление на страницу «Редактировать курс». Если же нет, то происходит перенаправление на страницу «Курсы».

При перенаправлении пользователя происходит аутентификация пользователя. И если пользователь не зарегистрирован в системе будет происходить перенаправление на страницу «Регистрации и авторизации». И лишь при успешной аутентификации будет происходить перенаправление ни страницу «Редактировать курс».

Страница «Редактировать курс» состоит из формы, в которую входят следующие поля: title, price, img, description. При заполнении данных полей происходит валидация формы. Если формы заполняются не должным образом, то пользователь получает сообщение с ошибкой. При успешном прохождении валидации происходит замена старых значений полей на новые, после чего происходит перенаправление на страницу «Курсы».

* Удаление курса.

«Пример удаления курса представлен в приложении A»

Пользователь посылает get запрос, происходит аутентификация пользователя. Далее через модель курса происходит сверка \_id курса и id пользователя, после чего происходит удаление курса из БД с последующим перенаправлением на страницу «Курсы».

1. Реализация
   1. Разработка классов информационой системы

Рассмотрим информационную систему интернет-магазина согласно разработанной архитектуре MVC представленной в главе 3.1 «Разработка архитектуры программного продукта».

Как было сказано ранее Express делает сложные вещи не существенными благодаря настройке веб сервера для прослушивания входящих запросов и возврата соответствующих ответов. Дополнительно он задает структуру каталогов.

«Пример структуры каталогов, созданных с помощью framework Express, представлен в приложении A»

Для начало необходимо проинициализировать проект. В терминале прописываю **npm init -y.** C помощью флага -y я соглашаюсь со всеми полями которые предлагает **npm**  и сразу же получаю файл **package.json.** В данном файле хранятся все зависимости проекта.

Далее устанавливаю еще два пакета:

* npm install express. В package.json появилось поле dependencies (зависимости) в которую установился данный пакет. В dependencies хранятся зависимости для работы приложения
* npm i -D nodemon. Флаг -D устанавливает пакет в зависимость devDependencies. В данную зависимость входят те пакеты которые нужны для разработки.

В корне приложения создаю входной файл приложения index.js. В первую очередь в данный файл необходимо подключить const express = require(‘express’).

Далее создаю новый сервер const app = express(). Переменная app является результатом работы функции express. Переменная app является аналогом объекта Server т.е. у данного объекта я могу вызвать метод который называется listen и сюда передаю тот порт на котором слушается приложение. Первым параметром в функции listen указываю переменную PORT. Вторым параметром передаю callback функцию в которой при удачном подключении выводится сообщение ‘Сервер был запущен на порту ${PORT}’.

В переменной PORT есть метод procces.env.PORT. Данный метод позволяет задавать порт вручную из консоли. Однако если этот порт не определен, то приложение запуститься на порту 3000.

Для того что бы разработка проходила с применением MongoDB, а не в каких-нибудь динамических файлах на локальной машине, необходимо подключить базу данных.

Для подключения MongoDB ее нужно предварительно настроить. При создании пользователя указываю имя пользователя и пароль. Данный пароль требуется для подключения к БД. Указываю разрешенные IP адреса. Далее предлагается url адрес по которому будет проходить подключение. В строке url есть удаляю переменную password и вставляю ранее сгенерированный пароль.

Для того что бы максимально упростить себе работу я устанавливаю npm install mongoose. Подключаю данный пакет const mongoose = require(‘mongoose’).

И теперь с помощью него подключаюсь к MongoDB. Создаю асинхронную функцию start. И для того что бы она заработала, после тела функции, я ее вызываю start().

В теле функции через оператор await я обращаюсь к пакету mongoose и для подключения к БД вызываю у него метод connect. В connect первым параметром передаю keys.MONGODB\_URI. Эта переменная которая находится в файлах keys.dev.js и keys.prod.js. И в зависимости от какого режима запускается приложение production или development В данную переменную передается url адрес с паролем для подключения к БД.

Вторым параметром указываю объект куда передаю useNewUrlParser: true это синтаксический анализатор строки который позволяет подключится к БД. Модель useFindAndModify влияет на модель и функции запроса. useUnifiedTopology обнаруживает и мониторит сервер.

После этого в тело функции start() добавляю функцию прослушки порта. И вставляю ее после подключения к БД. Делается это для того что бы на момент запуска приложения уже точно была доступна БД.

Для обработки потенциальных ошибок, блок подключения к БД и прослушки порта, оборачиваю в try catch.

«Пример запуска приложения представлен в приложении А»

«Пример подключения Handlebars представлен в приложении А»

В терминале с помощью команды npm i express-handlebars устанавливаю данный пакет. Для того что бы Handlebars позволял получать доступ к методам прототипа и свойствам входного объекта из шаблона устанавливаю пакет npm i @handlebars/allow-prototype-access

Далее в index.js подключаю пакет const exphbs = require(‘express-handlebars’) и пакет const {allowInsecurePrototypeAccess} = require('@handlebars/allow-prototype-access');

. После чего создаю новый объект в который передаю конфигурацию. Создаю переменную const hbs = обращаюсь к константе exphbs у которой вызываю метод create(). В вызванный метод я передаю объект конфигурации:

* defaultLayout: 'main'. Устанавливаю название главного макета т.е. в приложении будет несколько макетов.
* extname: 'hbs'. Указываю расширение файлов.
* helpers: require('./utils/hbs-helpers'). Созданная мной вспомогательная функция для handlebars которую можно использовать в шаблонах. Данная функция предоставляет пользователям права доступа с помощью которых только пользователь добавивший курс сможет его редактировать.
* handlebars: allowInsecurePrototypeAccess(Handlebars). Отключаю ограничения безопасности тем самым позволяю в модулях проводить проверку прототипов.

«Пример подключения Webpack и Babel представлен в приложении А»

В первую очередь устанавливаю данные пакеты:

* webpack
* webpack-cli
* babel-loader
* @babel/core
* @babel/preset-env

В терминале прописываю npm i -D и далее просто перечисляю данные пакеты. После установки данных пакетов в devDependencies в корне проекта создаю файл webpack.config.js.

В данном файле подключаю модуль const path = require(‘path’). Данный модуль используется для обработки и преобразований путей. После чего экспортирую модуль module.exports ={}. Инкапсулированный код данного модуля используется во множестве других файлов. Объект конфигурации данного модуля состоит из:

* mode: ‘development’. Указываю режим разработки.
* devtool: sourse-map. Использую для отладки кода.
* context: path.resolve(\_\_dirname, 'src'). Указываю папку где будет вестись разработка
* entry: ['./main.js', './main.scss']. При запуске webpack ищет точку входа в приложение. Через entry указываю главные входные файлы.
* output: {path: path.resolve(\_\_dirname, 'public')}. Результат сгенерированного кода будет помещен в папку public.

Далее указываю необходимые модули для которых прописываю правила. Правила состоят из массива объектов. Первый объект состоит:

* test: /\.(js|jsx)$/. В регулярном выражении указываю расширение обрабатываемых файлов.
* exclude: /node\_modules/. Запрещаю обрабатывать папку node\_modules т.к. в ней находятся все необходимые расширения, библиотеки, скаченные пакеты и т.д.
* loader: 'babel-loader'. Указываю что Babel будет работать в связке с Webpack.

После создания данной связки создаю конфигурационный файл .babelrc. В данном файле в формате JSON указываю лишь один preset: @babal/preset-env. Через данный preset babel подтягивает все необходимые плагины для работы babel. Плюс данного preset в том что не нужно перечислять огромное количество плагинов для работы babel.

Далее устанавливаю пакет npm i -D babel-minify-webpack-plugin.

Перехожу в webpack.config.js и подключаю данный пакет const MinifyPlugin = require('babel-minify-webpack-plugin'). Далее подключаю этот плагин и передаю в него два объекта. Объект указанный в первом параметре оставляю пустым, а во второй заношу comments: true, тем самым говоря что когда будет происходить минификация кода, из него нужно убрать все комментарии.

После устанавливаю npm i -D sass-loader node-sass css-loader postcss-loader mini-css-extract-plugin.

Подключаю плагин const MiniCssExtractPlugin = require('mini-css-extract-plugin'); После чего в rules (правила) добавляю еще один объект конфигурации:

* test: /\.(scss|css)$/. Через регулярное выражение указываю расширение обрабатываемых файлов.
* exclude: /node\_modules/. Запрещаю обрабатывать папку node\_modules.

Далее все в том же объекте передаю пользовательские настройки в виде цепочки loader (загрузочный).

* loader: MiniCssExtractPlugin.loader. Этот плагин минифицирует (сжимает) CSS файлы. Он создает файл CSS для каждого файла JS, который содержит CSS. Он поддерживает загрузку по требованию CSS и SourseMap.
* options: {reloadAll: true,}. Этот модуль производит перезагрузку и обновление CSS файлов в процессе разработки.

Далее указываю еще три loader.

* css-loader. Преобразовывает CSS – в модуль JavaScript
* postcss-loader – данный loader автоматически добавляет вендорные префиксы и проводит проверку соблюдения стандарта оформления кода. Данный loader работает за счет подключаемых в него плагинов.
* sass-loader. Загружает файлы SCSS и компилирует их в CSS.

Последовательность загрузки loader имеет огромное значение. Загрузка файла начинается с конца документа. В связи с этим sass-loader скомпилирует все в css, postscss-loader добавит вендорные префиксы и приведет к определенному стандарту, css-loader преобразует css в модуль JavaScript и после этого произойдет минификация (сжатие) css файла.

В корне приложения создаю конфигурационный файл postcss.config.js, устанавливаю два плагина npm i -D cssnano autoprefixer, далее добавляю эти два плагина в конфигурационный файл.

И последний штрих для обработки CSS, папку public делаю статической. Для этого в index.js прописываю app.use(express.static(path.join(\_\_dirname, 'public')));

Для того что бы вести разработку на front-end и back-end нужно сделать так, чтобы webpack и nodemon запустились и работали в связке. Для этого устанавливаю пакет npm i -D concurrently. Далее в package.json прописываю скрипты для запуска этих зависимостей по отдельности

* "build": "webpack -w",
* "dev": "nodemon index.js",

С помощью пакета currently создаю еще один скрипт, где связывают данные скрипты между собой.

* "webapp:dev": "concurrently 'npm:build' 'npm:dev'"

Теперь если ввести npm run webapp:dev, webpack и nodemon будут запускаться одновременно и работать синхронно. Автоматизация данного процесса приводит к тому, что получаемые ошибки легче отслеживать

«Пример регистрации пользователей представлен в приложении А»

«Пример Login пользователей представлен в приложении А»

В модели представления views/auth создаю файл login.hbs. В данном файле я реализую определенную форму, с помощью которой пользователь сможет регистрироваться и логиниться в системе.

В контроллере routes/auth.js из express достаю объект Router. После присваиваю данный объект переменной router. Делаю я это для того, что, бы, можно было более удобно взаимодействовать с данным объектом.

Для того что бы отобразить страницу я обращаюсь к переменной router и вызываю метод get, где первым параметром создаю относительный путь обрабатываемой страницы ‘/login’, а вторым параметром указываю асинхронную функцию которая в свои параметры принимает request(запрос), response(ответ). Результат работы данной функции представлен в теле функции. Обращаюсь к res.render где первым параметром указываю обрабатываемы путь ‘auth/login’, а вторым параметром указываю объект конфигурации в который входят следующие поля:

* title: 'Авторизация',
* isLogin: true,
* loginError: req.flash('loginError'),
* registerError: req.flash('registerError')

Рассмотрим обработку данных полей на примере регистрации пользователя. Обращаюсь к router.post. Первым параметром создаю маршрут ‘/register’, Вторым параметром указываю утилиту, которая проводит валидацию формы. Третьим параметром передаю асинхронную функцию, в которую входят два параметра req, res. В теле функции всю логику оборачиваю в try catch и прописываю что, если возникают какие – либо ошибки их нужно выводить в консоль.

И первым делом все поля email, password, name, получаемы от пользователя, я заношу в объект req.body. И далее, для того, что бы можно было взаимодействовать с пользователем я создаю модель пользователя model/user.js.

После того как я задал обязательные поля для ввода я создаю нового, в routes/auth, пользователя. Переменной user присваиваю новый класс User который принимает объект со следующими полями.

* Email
* Name
* Password : hashPassword

Email и name передаются в явном виде, как и есть. А вот пароль необходимо зашифровать. Для шифрования пароля устанавливаю.

* npm i bcrypt

Далее создаю переменную hashPassword и в присваиваемом значении которой обращаюсь к оператору await, после обращаюсь к пакету bcrypt его методу hash, с помощью которого происходит шифрование, первым параметром принимается пароль, приходящий от пользователя, вторым параметром указываю количество символов шифрования. После зашифровки пароля данное значение присваивается переменной hashPassword. После чего данную переменную, при создании пользователя, передаю в поле password.

Что бы перенаправление на страницу логина не произошло преждевременно, обращаюсь к оператору await user.save() тем самым говоря что перенаправление не должно происходить до тех пор пока все данные не сохраняться.

Сейчас на данный момент пользователь может вводить абсолютно любые значения и, как ни странно, они занесутся в БД. Что бы такого не происходило, нужно задать метод, который будет проводить анализ заданных условий и проверять вводимые пользователем данные на соответствия требованиям.

Одним из таких методов является валидация формы. И для этого я устанавливаю библиотеку:

* npm i express-validator

Как и другие пакеты в express данная библиотека использует middleware для определения полей, которые нужно отвалидировать.

Самым лучшим решением вынесение валидаторов в отдельный файл utils/validators.js. Подключаю  const { body } = require('express-validator'); Далее я воспользуюсь полем exports куда я буду складывать именованные объекты. Для страницы регистрации я использую объект registerValidators. Т.к. полей достаточно много, я описываю данный объект как массив.

Для того что бы создать сообщение вызываю withMassage('Введите корректный email'). Для пароля body('password') пароль должен быть минимальной длинны. Для этого я использую функцию isLangth({}) в качестве объекта указываю min: 6, max: 66. Помимо этого я могу добавить валидатор .isAlphanameric() потому что мы требуем пароль написанный латиницей и состоящий из цифр и букв.

Что касается поля, которое повторяет пароль. body('confirm'). как проверить данное поле, совпадало с тем значением которое есть в пароле. Для этого я воспользуюсь кастомным валидатором который напишу самостоятельно. И вызывается он через функцию custom(). В данную функцию, я передаю callback, где буду писать логику данного валидатора. Данная функция принимает (value) которое написано в поле confirm. И вторым параметром принимаем объект у которого есть поле {req}.

И здесь что бы проверить валидатор просто добавляю проверку if (value !== req.body.password) если не совпадает то в таком случае я буду делать thow new Error('Данные пароли не совпадают'). А если пароли совпадают, то возвращаю истину.

Валидатор для имени withMessage('Имя должно быть минимум 3 символа'),

Для работоспособности валидаторов в auth.js подключаю const { validationResult } = require('express-validator'); и прописываю путь до файла валидаторов const { registerValidators } = require('../utils/validators'); и добавляю его как midleware в форму регистрации.

Асинхронные валидаторы.

Для этого прописываю еще один кастомный валидатор первым параметром которого принимаю value т.е. значение, переданное пользователем, а вторым параметром callback функцию в теле которой прописываю блок try catch для проверки ошибок. Импортирую модель пользователя const User = require('../models/user').

И дальше. по-сути все очень просто. Я создаю переменную const user  или может быть переменную candidate, и как обычно с помощью оператора await проверяю User.findOne({email: req.body.email}), но на самом деле лучше использовать не req.body.email оно конечно сработает. Но лучше использовать значение value которое я получаю в кастомном валидаторе. И дальше в этой функции возвращаю promise потому, что здесь операция асинхронная, она требует некоторого количества времени и поэтому здесь я проверяю. Если if есть какой – то пользователь (user) то это означает что человек уже не может зарегистрироваться, и поэтому я должен вернуть return, и так как мне нужно вернуть только ошибку я обращаюсь к глобальному классу Promise и вызывать у него метод который называется reject ('Такой email уже занят'). И после этого, валидаторы будут ждать, когда promise завершиться, и только после этого, выдаст какой - то результат.

Еще в библиотеке валидаторов присутствуют дополнительные функции, которые улучшают данные перед тем, как они поступят в БД. Такие функции называются Sanitizers. Они служат для того, чтобы, чистить данные, защищать данные и т.д. Для email добавляю функцию normalieEmail(). Теперь если пользователь будет вписывать какой – то кривой email, то данная функция поправит это. Что касается пароля, то сюда я добавляю функцию trim(). Валидатору повтора пароля так же добавляю функцию trim(). Данная функция удаляет лишние пробелы.

Логин пользователя.

В routes/auth.js обращаюсь к router.post который принимает два параметра. Первым параметром указываю обрабатываемый путь ‘login’, вторым параметром указываю асинхронную функцию в которую входит два параметра req и res. В теле данной функции прописываю блок try catch с помощью которого будут отлавливаться ошибки.

В блоке try, из объекта req.body достаю два поля email и password. После чего с помощью класса User и его метода findOne произвожу поиск одного email в БД и найденное значение присваиваю переменной candidate.

После нахождения пользователя по email в переменной candidate произвожу проверку по следующим параметрам. С помощью оператора обращаюсь к библиотеке bcript и его методу compare(сравнение) и первым параметром передаю пароль переданный пользователем password, а вторым параметром передаю зашифрованный пароль из БД candidate.password. Оператор await ждет пока произойдет данное сравнение и после этого заносит полученное значение в переменную areSame.

После сравнения пароля пишу еще одно условие где в переменной areSame с помощью req.session.user заношу данного candidate в сессию. С помощью аутентификации произвожу проверку подлинности. С помощью req.session.save, в параметрах помещаю callback функцию с помощью которой если что то пошло не так вывожу сообщение ошибки ‘Неверный пароль’ или ‘Такого пользователя не существует’ и делаю перенаправление на страницу логина, иначе если данные успешно были занесены и сохранены в сессии, произвожу перенаправление на главную страницу приложения.

Сброс пароля пользователя.

«Пример сброса пароля пользователя представлен в приложении А»

В PHP есть функция, которая позволяет с помощью сервера отправлять email на любой почтовый адрес.

Так вот в NodeJS это не так. Для того что бы отправлять какие-либо email необходимо пользоваться сторонними сервисами, которые предоставляют данную возможность. Для этого я пользуюсь сервисом SendGrid. Данный сервис крайне прост в настройке. При настройке генерируется определенный ключ, который будет рассмотрен далее.

Устанавливаю дополнительные пакеты для работы с сервисов SendGrid и мне потребуется два пакета nodemailer - общий пакет который позволяет подключать в себя другие транспортеры, и второй пакет nodemailer-sendgrid-transport

* npm install nodemailer nodemailer-sendgrid-transport

В routes/auth.js подключаю

* const nodemailer = require('nodemailer');
* const sendgrid = require('nodemailer-sendgrid-transport');

Теперь создаю определенный транспортер, который будет являться объектом служащим для того, чтобы отправлять email. Для этого создаю переменную transporter и далее я обращусь к библиотеке nodemailer и у него есть метод который называется createTransport(). В эту функцию передаю тот сервис которым пользуюсь т.е. sendgrid который тоже является функцией поэтому так же вызываю sendgrid().

И в функцию sendgrid передаю объект auth, который в свою очередь так же принимает объект в котором находится api\_key: keys.SENDGRID\_API\_KEY импортируемый из файла keys.prod.js или keys.dev.js в зависимости от режима в каком запущено приложение.

Хорошей практикой является добавление кнопки забыли пароль. Здесь логика не совсем стандартная.

В папке auth создаю новый файл reset.hbs.

В контроллере auth.js прописываю get запрос. Первым параметром указываю путь ‘/reset’, вторым параметром указываю callback функцию которая в свои параметры принимает request и response. В теле функции пишу результат отображаемой страницы, обращаюсь к res его методу render. Первым параметром принимаю обрабатываемы путь ‘auth/reset’, а вторым параметром указываю объект конфигурации в который входит поле title=’Забыли пароль’ и поле которое генерирует сообщение об ошибке error: req.flash(‘error’) которое я достаю из пакета flash.

Для того что бы приступить к написанию post запроса на восстановление пароля необходимо понять в чем вообще заключается идея восстановления пароля. По сути будет генерироваться какой-то рандомный, случайный ключ который запишется в БД. После чего произойдет отправка письма пользователю, в котором будет содержаться данный сгенерированный ключ. Человек переходит по ссылке, в которой содержится данный ключ, и если он будет совпадать с тем ключом который содержится в БД и жизнь Token не истечет, то тогда пользователю будет предоставлена возможность изменить пароль. Однако помимо синхронизации token необходимо произвести проверку пользователя по id и email.

Первое что я делаю это генерирую какой – то случайный ключ. Для этого подключаю встроенную в NodeJS библиотеку.

* crypto. const crypto = require('crypto');

Для того что бы сгенерировать какой - то случайный ключ, в блоке try обращаюсь к библиотеке crypto и ее методу randomBytes(). Первым параметром указываю символы в байтах, допустим 32. После генерации ключа он попадает во второй параметр, который является callback функцией.

Callback первым параметром принимает ошибку err. Вторым параметром принимает buffer. Ошибка выводится в том случае если по какой – то причине ключ не смог сгенерироваться.

Если же ключ сгенерировался успешно, происходит перенаправление на страницу auth/res. После чего создаю переменную token которой присваиваю результат работы buffer, вызываю метод toString(‘hex’) тем самым указываю что получаемое значение будет являться строкой формата hex. Это и будет значением token.

Далее убеждаюсь, что email который отправился с клиента вообще соответствует какому – либо пользователю в БД. Создаю переменную candidate и в ней обращаюсь к модели пользователя User. У данной модели вызываю метод findOne в параметрах которого указываю объект с искомым полем email: req.body.email. Так как данная конструкция является асинхронной то дописываю оператор await.

Теперь если данный пользователь обнаружен, то происходит отправка письма данному пользователю. А иначе если данный пользователь не найден выводится сообщение об ошибке req.flash('error', 'Данный email не найден'); и происходит перенаправление на страницу сброса пароля res.redirect('/auth/reset').

Если пользователь, который хочет обновить пароль, обнаружен, то для этого нужно в БД добавить два значения. Тот token который я создал и время жизни данного token.

Для этого в модели пользователя прописываю два поля resetToken: String, и resetTokenExp: Date т.е. Expiration.

После чего в контроллере auth.js обращаюсь к переменной candidate и вызываю метод resetToken в значение которого заношу token.

Далее снова обращаюсь к переменной candidate, вызываю метод resetTokenExp. В значении обращаюсь к текущей дате Date.now()

+60\_60\_1000 и задаю время жизни token один час. После чего необходимо подождать await пока candidate будет сохранен, так как необходимо занести эти данные в БД. И после этого нужно отправить новое письмо пользователю с помощью await transporter.sendMail(). В функцию sendMail() необходимо передать созданное email письмо.

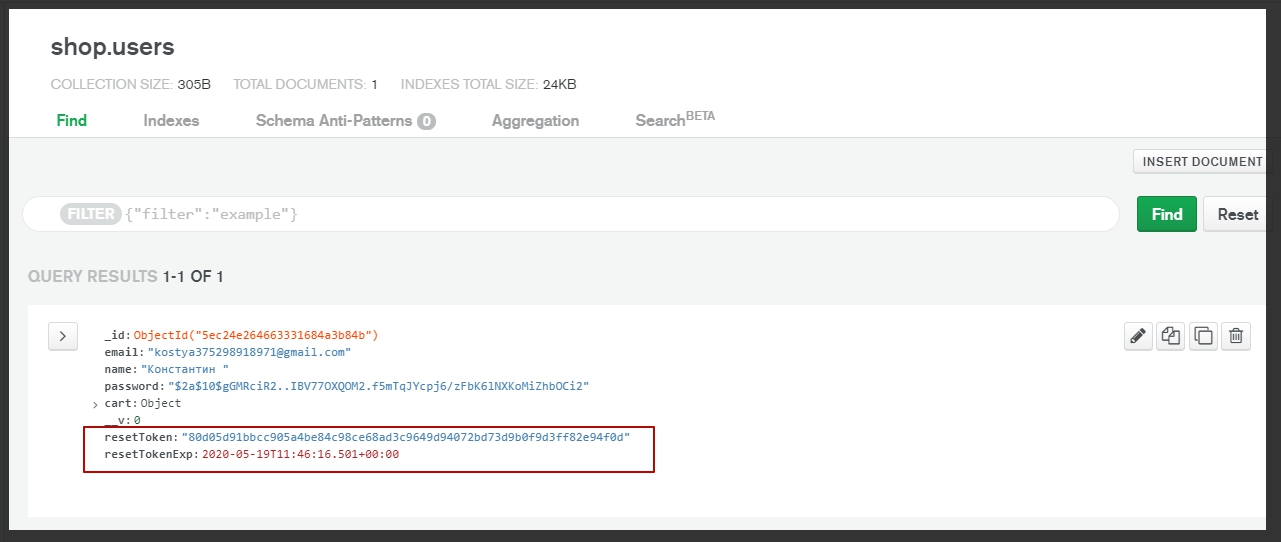


Рис- Пример reserToken и resetTokenExp в База данных MongoDB.

В папке emails создаю файл reset.js и из модуля я экспортирую function(email). В теле функции генерирую новый объект. URL для восстановления необходимо сгенерировать на основе двух параметров. Необходимо указать BASE\_URL. Необходимо передать путь который позволит получить доступ к восстановлению пароля. Так же необходимо передать token в get параметре который был заранее сгенерирован. И плюс для того, чтобы token сработал добавляю его вторым параметром к функции.

const keys = require('../keys');

module.exports = function (email, token) {

return {

to: email,

from: keys.EMAIL\_FROM,

subject: 'Восстановление доступа',

html: `<h1>Вы забыли пароль?</h1>

<p>Если нет, то проигнорируйте данное письмо</p>

<p>Иначе передите по ссылке</p>

<p><a href="${keys.BASE\_URL}/auth/password/${token}">Восстановление доступа</a></p>

<hr />

<a href="${keys.BASE\_URL}">Магазин курсов</a>

`,

};

};

Теперь в файле auth.js подключаю письмо.

* resetEmail = require('../emails/reset');

Теперь в параметры функции sendMail передаю функцию resetEmail. В параметры функции resetEmail передаю candidate.email т.е. email того пользователя который восстанавливает пароль. Вторым параметром функция resetEmail принимает token. После отправки письма пользователю происходит перенаправление пользователя на страницу логина.

router.post('/reset', (req, res) => {

try {

crypto.randomBytes(32, async (err, buffer) => {

if (error) {

req.flash('error', 'Что-то пошло не так, повторите попытку');

return res.redirect('/auth/reset');

}

const token = buffer.toString('hex');

const candidate = await User.findOne({ email: req.body.email });

if (candidate) {

candidate.resetToken = token;

candidate.resetTokenExp = Date.now() + 60 \* 60 \* 1000;

await candidate.save();

await transporter.sendMail(resetEmail(candidate.email, token));

res.redirect('/auth/login');

} else {

req.flash('error', 'Данный email не найден');

res.redirect('/auth/reset');

}

});

} catch (e) {

console.log(e);

}

});

Страница нового пароля.

После перехода пользователя по ссылке он попадает на страницу представления password.hbs.

В контроллере auth.js задаю новый обработчик router.get('/password/:token', async (req, res) => {} в обрабатываемый путь добавляю :token для того что бы обрабатывался наш токен. Делаю функцию асинхронной. В тело функции я буду передавать определенные данные для еще большей защиты. Потому что подобные страницы можно потделать. И я сделаю так что бы максимально сложно было подстроить какие - то параметры.

Во - первых я спрашиваю у объекта request. Если if в объекте (!req.params.token) нет параметра token, то тогда такую страницу не буду допускать к открытию return res.redirect('/auth/login').

Т.е. пользователи не имеющие токена не будут допускаться на данную страницу.

После этого произвожу поиск пользователя у которого имеется в БД необходимый token. Для этого создаю переменную user и в присваемом ей значении, через оператор await обращаюсь к модели пользователя User, вызываю метод findOne в параметры которого передаю объект. Во – первых проверяю на совпадение restToken и req.params.token. После чего убеждаюсь что поле resetTokenExp еще валидно. resetTokenExp сравниваю с объектом в котором так же происходит сравнение $gt, которое должно быть больше чем текущая дата, с Data.now().

Далее прописываю условие. Если данный пользователь не найден, то происходит перенаправление на страницу логина. Т.е. если все те свойства не соответствуют требованиям пользователя, то происходит перенаправление на страницу логина. Одним из таких требования является resetTokenExp, время жизни token.

Иначе если все требования выполнены происходит перенаправление на страницу нового пароля. Страница нового пароля состоит из следующих полей.

* Title: ‘Восстановить пароль’ – описание страницы.
* Error: req.flash(‘error’) – вывод ошибок.
* userId : user.\_id.toString() – для защиты страницы.
* Token: req.params.token

В пост запросе прописываю следующую логику. Первым параметром указываю обрабатываемый маршрут ‘/password’. Вторым параметром асинхронную функцию в параметры которой входит req, res. В тело функции добавляю try catch для отлова ошибок.

Далее я обрабатываю форму с новым паролем. Но для начало необходимо проверить наличие пользователя с такими данными.

<input type="hidden" name="\_csrf" value="{{csrf}}">

<input type="hidden" name="userId" value="{{userId}}">

<input type="hidden" name="token" value="{{token}}">

Создаю переменную const user = await User.findOne({}); и снова провожу поиск пользователя с такими условиями. Где \_id: req.body.userId, и так же нужно проверить валидацию токена resetToken: req.body.token, и resetTokenExp: должен быть больше чем { $gt: Date.now() }. Иначе он потерял время жизни и поэтому пользователь будет не найден.

Далее пишу условие что если пользователь найден, то задаю ему новый пароль, а иначе произойдет перенаправление на страницу логина.

Теперь если пользователь найден и все валидно. В таком случае я обращаюсь к user.password = и дальше снова шифрую уже новый пароль который поступил в форму. Для этого обращаюсь к библиотеке await bcrypt. и с помощью функции hash() передаю параметр req.body.password, 10 количество символов.  
После этого необходимо удалить все данные которые относятся к токену восстановления т.е. user.resetToken = indefined и user.resetTokenExp = undefined; После сохранения новых данных происходит перенаправление на страницу логина.

Диаграмма классов добавления, редактирования и удаления карточки курса рассмотрено в главе «4.3 Разработка алгоритмов реализации вариантов использования».

«Пример добавления курса в корзину представлен в приложении А».

В представлении courses.hbs добавляю кнопку которая передает id того курса который пользователь хочет приобрести.

<form action="/card/add" method="POST">

<input type="hidden" name="id" value="{{id}}" />

<button type="submit" class="btn btn-primary">

Купить

</button>

</form>

В контроллере card.js импортирую класс Router из библиотеки express и сразу для удобства работы заношу данный класс в переменную router.

Далее обращаюсь к переменной router, вызываю метод post где первым параметром указываю маршрут ‘/add’ а вторым параметром указываю callback функцию.

После чего в точке входа в приложение, entry point, в файле index.js я импортирую и регистрирую данный маршрут.

В теле функции создаю переменную course т.к. действие асинхронное я обращаюсь к оператору await, к модели Course вызываю метод getById в параметрах которого прописываю искомый req.body.id. Так как корзина привязана к каждому пользователю, т.е. для каждого пользователя она уникальна я получаю пользователя с помощью объекта req.user.

и теперь смотрим способ который предоставляет mongoose для того что бы более удобно выстраивать подобные логические цепочки.

Перехожу в models user.js. У меня есть userSchema которую я могу расширить. В конце документа, перед экспортом модуля обращаюсь к userSchema. к его объекту methods. далее здесь я могу определить другой любой метод который вынесет логику прямо в объект пользователя. И называю этот метод addToCart = addToCard это метод который нужно определить как функцию, но здесь есть один нюанс. Здесь обязательно использовать ключевое слово function(){}. Т.е. конечно я могу задать функцию через стрелочную функцию. Но здесь важно использовать ключевое слово function потому, что, я буду здесь работать с ключевым словом this.  
Когда я создаю функцию через стрелочную функцию т.е. без ключевого слова function, контекст this я не могу использовать. Я не смогу с ним работать. Поэтому это критично важно.

И внутри этой функции я буду прописывать логику, которая позволит добавлять элемент в корзину.  
В параметр function я принимаю объект (course).

Данный метод req.user.addToCart(course); является асинхронным, т.е. он будет что то сохранять в БД. Поэтому к нему также применяю оператор await. Для того что бы NodeJS подождал пока выполнится данный req.user.addToCart(course) запрос.

И вот теперь необходимо реализовать логику, которая позволит добавить этот курс в корзину. И в случае, если в корзине лежит такой курс, то тогда нужно просто увеличить количество.

Перехожу в models user.js и продолжаю писать.

Для начало, мне потребуется, обратится к корзине cart и забрать у нее массив items. Именно в данном массиве лежат те элементы у которых есть свойство count и courseId.

Для того что бы в JS не было мутаций в теле функции создаю массив const items и просто клонирую массив items т.е. обращаюсь к this.cart.items и с помощью ES6 синтаксиса создаю новый массив this.cart.items. Однако для того что бы все сработало корректно необходимо развернуть данный массив. Для этого я использую Spreed оператор.

Для логичности переименовываю переменную на clonedItems которая принимает новый клонированный массив.

Дальше нужно найти в данном массиве clonedItems тот курс с которым я работаю, а точнее найти его индекс const idx =. Далее я обращаюсь к clonedItems. и у этого массива я могу вызвать метод findIndex() для того что бы найти индекс того или иного элемента, я прописываю условие где на каждой итерации я получаю объект findIndex(c =>{}).

Далее в callback мне необходимо вернуть условие. У каждого элемента есть поле coursed, поэтому обращаюсь к return c.courseId. Но здесь стоит учитывать, что courseId объект типа Schema.Types.ObjectId. И для того что бы его сравнивать с другими строками, необходимо вызывать метод toString(). Это очень важный момент так как могут возникнуть определенные проблемы.

И сравниваю return c.courseId.toString() === с тем курсом который был передан в параметры function (course), т.е. course.\_id. и также привожу к строке.

И теперь у меня есть потенциальный id, позиция данного курса в массиве, которая есть в корзине. Т.е. если данный idx найдется т.е. >= 0, то значит я уже несколько раз добавлял этот курс и по этому нужно увеличить количество.  
А если же курс не нашелся то тогда его нужно добавить в корзину.

И для этого пишу следующую логику. В случае, когда idx будет равняться -1, то тогда будет происходить добавление элемента т.е. если idx >= 0 это означает что в корзине уже есть такой курс и счетчик должен увеличится. Внутри тела условия у cloneItems[idx]. обращаюсь по индексу с которым работаю, далее обращаюсь к его полю count = и в него я заношу новое значение которое должно увеличится от предыдущего clonedItems[idx].count + 1.

Иначе если idx равен минус одному я обращаюсь к clonedItems и с помощью метода push добавляю новый элемент где по сути нужно описать два поля count и courseId. courseId: будет равняться idx который сюда передавал т.е. course.\_id. Ну а количество по умолчанию count: 1.

Сокращаю код. Далее я обращаюсь к this.cart = {} и в нее заношу значение items: clonedItems. Таким образом я избавляюсь от создания одной переменной. Но более того если бы мы назвали переменную clonedItemsпросто items, то тогда нам бы с этим былобы просто работать. Меняю все clonedItems на items

И теперь когда ключ и значение совпадают this.cart = { items: items }; я могу просто написать this.cart = { items }; т.е. просто опустить значение. И последнее что необходимо сделать в методе addTocart это вернуть return this.save() Обязательно вызываю данный метод.

Отображение корзины.

В контроллеру card.js и в router.get первым параметром добавляю маршрут ‘/’. Вторым параметром принимает callback функцию. В теле функции создаю переменную user которой присваиваю значение объекта req.user в котором содержится id пользователя т.к нужно подождать пока данный пользователь найдется я принимаю оператор await.

Задача заключается в том что бы получить корзину, а корзина находится в модели пользователя. Для получения корзины в объекте req.user я вызываю метод populate c помощью которого я вытаскиваю из модели пользователя из объекта cart массив items в котором содержится объект courseId. Данное обращение я помещаю в параметр метода populate(‘cart.items.courseId’) и для того что бы все работало вызываю метод exectPopulate.

Теперь значения находящиеся в courseId необходимо привести в надлежащий вид для того что бы front-end понимал как с этим работать. В контроллере card.js В get запросе добавляю еще одно поле courses: обращаюсь к модели user.

router.get('/', async (req, res) => {

const user = await req.user.populate('cart.items.courseId').execPopulate();

console.log(user.cart.items);

res.render('card', {

title: 'Корзина',

isCard: true,

courses: user,

price: computePrice,

});

});

Далее я хочу сформировать массив который буду передавать на front-end. Для этого создаю функцию function mapCartItems в параметры которой заношу cart. В get запрос передаю результат работы этой функции const courses = mapCartItems(user.cart);

В теле функции mapCartItems я возвращаю новый массив cart.items.map() в параметры помещаю callback функцию где на каждой итерации будет получен некоторый объект у которого все данные относятся к конкретному курсу. они лежат в поле c.courseId. и что бы убрать эту всю метадату нужно обратиться к его полю \_doc. И использую ES6 ...c.userId.\_doc для того что бы развернуть данный курс, и что касается счетчика просто указываю count: c.count.

Далее для корректного отображения цены создаю функцию computerPrice. В параметры передаю массив courses. И в get запросе в поле price передаю результат работы данной функции computerPrice.

Для реализации функции computerPrice, что лежит в массиве courses, необходимо умножить цену на количество курсов и сложить это все со всеми курсами которые есть в массиве, то получиться финальная цена. Для этой задачи я использую метод который называется reduce который есть у массивов, который выполняет задачу в одну строку. Данная функция первым параметром принимает в себя callback функцию, а вторым параметром принимает целочисленное стартовое значение ноль.

В callback первым параметром принимаю объект total который на каждой итерации будет увеличиваться, а вторым принимаю объект course. И в теле callback логика простая. Она будет просто возвращать переменную return total += и будем ее увеличивать на значение course.price \* course.count.

«Пример удаления курса из корзины представлен в приложении А».

Удаление из корзины является асинхронным действием т.к. все данные хранятся в БД.

В контроллере card.js обращаюсь к объекту router и вызываю метод delete. Первым параметром принимаю маршрут ‘/remove/:id’. Вторым параметром принимаю асинхронную callback функцию в параметры которой принимаю req, res. В теле функции обращаюсь к оператору await, у объекта req.user вызываю метод, который я создам позже, removeFromCart в параметры которого принимаю req.params.id. После этого на front-end необходимо вернуть объект корзины в том же формате в котором она была до этого. Для того что бы клиентский JS смог заново динамически отобразить страницу.

Опять создаю объект пользователя const user = await далее обращаюсь к req.user вызываю метод populate('') в котором так же обращаюсь к cart.items.courseId, потому что из id его необходимо превратить в полноценный объект с курсами, и после этого вызываю метод exectPopulate().

Теперь у меня есть пока что пустой объект cart,  res.status(200).json(cart);.

Создаю объект const cart. Так же отдельно нужно создать переменную const courses  которую я буду получать из функции mapCartItems(user.cart) из user.cart. И теперь эти курсы я могу передавать в объект cart. Ну а price  я получаю из функции computePrice(course) куда я передаю course.

Теперь реализую метод removeFromCart куда я передаю id того курса, а именно [req.param.id](http://req.param.id/), который необходимо удалить. Для этого я вызываю модель пользователя. Перехожу в models user.js и здесь, перед module.exports что в конце документа, обращаюсь к userSchema, его мотоду methods и определяю новый метод который называется removeFromCart. Это такая же функция, которую обязательно определять через ключевое слово function. Которая принимает в свой параметр  id курса.

По сути для начало здесь так же нужно клонировать массив const items = [...this.cart.items], с помощью spred оператора обращаюсь к this.cart.items. Создаю новый массив для того что бы не было потенциальных мутаций. Далее я как обычно создаю переменну idx т.е. это потенциальный индекс того курса в массиве items который я должен найти. Но в методе removeFromCart этот индекс будет постоянно определяться, потому что, если нет элемента на front - end, его соответсвенно нет и в карточке. Поэтому здесь ошибка практически не возможна только если кто-то специально будет отсылать не правильные id отдельными запросами.  В const idx = я обращаюсь к items. методу findIndex где на каждой итерации получаю объект c. Далее я возвращаю условие по которому нахожу данный индекс т.е. я сравниваю c.courseId.toString() === id.toString().

Далее в условии проверяю если items[idx]. count эквивалентен одному т.е. один курс добавлен в корзину, тогда его необходимо удалить, а иначе срабатывает счетчик items[idx].count--. После срабатывания счетчика, сразу же переопределяю объект this,cart ={items}. После чего возвращаю this.save() тем самым говоря что пока items в корзине не обновятся дальнейшее действие происходить не будет.

Что касается логики if то здесь я переопределяю массив items = items.filter((c) => c.courseId.toString() !== id.toString()). Я просто удаляю с помощью метода filter элементы т.е. существующий id в корзине c.courseId.toString() я сравниваю с id.toString()который поступил.

Единственная проблема в данной функции заключается в том, что массив items определен через ключевое слово const. Для того что бы было все в норме нужно const заменить на let.

Однако сейчас хоть вся логика и реализована, удаление не будет происходить из-за конфликта id и \_id с нижнем подчеркиванием которое использует MongoDB. Для решения этой проблемы необходимо произвести трансформацию данных на клиенте. Перехожу в models courses.js.

Для того что бы исправить такую ситуацию. Когда id не правильно определяется.

Перерехожу models course.js.

Первым делом я обращаюсь к объекту courseSchema. далее я вызову метод который назыается method() и в его параметрах я могу описать тот метод который я хочу использовать. Первым параметром прописываю toClient т.е. это тот момент когда у нас будет трансформация данных к клиенту.  
Вторым параметром я передаю функцию через ключевое слово function(){}. И в теле функции создаю переменную const course через ключевое слово this. а оно доступно если используется ключевое слово function, и далее вызываю метод toObject() для того что бы получить объект данного курса.  
И по сути теперь необходимо сделать трансформацию. У объекта [course.id](http://course.id/) = course.\_id. И далее для того что бы много трафика не летоло я удаляю лишнюю информацию delete course.\_id. Ну и после этого мне нужно просто вернуть этот курс.

Второй момент заключается в том что кода происходит populate в контроллере card.js поле в …с.courseId.\_doc по умолчанию будет прописываться \_id. Для исправление данной ошибки я прописываю значение id а ключ задаю c.courseId.id. После чего логика удаления курса полностью готова.

«Пример страницы заказов представлен в приложении А»

Страница заказов имеется у каждого пользователя. Для этого в представлении views в файле card.hbs создаю еще одну форму. В данную форму помещаю кнопку “Сделать заказ”. В форме прописываю атрибут action=”/orders” в котором указываю обрабатываемый путь и вторым атрибутом указываю каким методом будет обрабатываться данная форма method =”POST”.

Так же в представлении navbar.hbs добавляю ссылку на данную страницу.

В контроллере создаю файл orders.js и далее в главном файл index.js подключаю и регистрирую данный маршрут.

В orders.js как обычно из mongoose вытаскиваю класс Router и присваиваю его переменной router. После чего обращаюсь к router, его методу get. Первым параметром создаю маршрут ‘/’. Вторым параметром принимаю callback функцию в параметрах которой находятся req, res. В теле функции я прописываю результат рендеринга страницы заказов. Для этого я обращаюсь к функции res.render. Первым параметром добавляю обрабатываемы путь к основному созданному маршруту. Вторым параметром принимаю объект, состоящий из полей в которые входит ключ и его значение т.е. isOrder: true и title: ‘Заказы’.

Теперь в модели представления создаю данную страницу orders.hbs.

После обрабатываю POST запрос на данную страницу. Первым параметром добавляю маршрут ‘/’. Вторым параметром принимаю асинхронную callback функцию параметры которой принимают req и res. В теле функции пока делаю перенаправление на res.redirect(‘orders’).

Однако для того, чтобы работать с заказами мне необходимы данные. И для этого нужна особенная модель. Перехожу в models и создаю orders.js. В файле импортирую mongoose и из него достаю класс Schema и конструктор model. Далее создаю переменную const orderSchema которой присваиваю новый объект класса new orderSchema. И после этого с помощью конструкции model.exports где регистрирую новую модель model('Order', orderSchema).

В orderSchema передаю поля которые будут присутствовать у каждого из заказов. Определяю массив courses:[] где каждый элемент будет являться объектом. Добавляю объект course со следующими полями: type: Object, required: true т.е. задаю тип объект, и делаю данное поля обязательным. Так же указываю объект count, счетчик который будет считать количество курсов. В данном объекте указываю поля содержащие ключи и значения а именно: type: Number, required: true.

Далее описываю пользователя совершивший заказ. В объект user задаю имя пользователя name : String. Создаю объект userId который является своего рода референцией на модель пользователя. Прописываю поля: type: Schema.Types.ObjectId, и указываю ссылку откуда достать этот объект ref: ‘User’ и после чего указываю что userId является обязательным.

Далее создаю объект даты в котором прописываю следующие поля: type: Date, default: Date: now т.е. указываю что тип является датой, по умолчанию берется дата текущего момента.

Теперь на страницу order.js я импортирую модель Order.

И далее прописываю логику в контроллерах.

Получение данных заказов.

Для создания заказа необходимо получить все данные, которые есть в корзине. Для этого в контроллере routes order.js в post запросе создаю переменную const user. Обращаюсь к оператору await который ждет пока будет получен пользователь req.user вызываю метод populate в параметры которого передаю (‘cart.items.courseId’) и для того что бы объекты курсов превратить в объекты добавляю execPopulate(). Тем самым я получаю наполненный объект пользователя где есть все объекты курсов.

Однако нужно привести к человеческому формату. Создаю переменную const users, обращаюсь к user.cart.items.map(). В параметрах map указываю callback функцию в которой на каждой итерации получаю items т.е. i . В callback оборачиваю объект в круглые скобки, и после этого мне не обходимо писать return. И я просто возвращаю count: i.count, а объекту course присваиваю новый экземпляр course: {…i.courseId.\_doc}. Через Spreed оператор разворачиваю данный объект.

После получения читаемого формата курсов я создаю новый объект const Order = new Order в объект которого передаю два обязательного параметра. Это объект user у которого есть поле name:req.user.name и поле userId: req.user.

Сами курсы я передаю как courses: courses, но так как ключ и значение совпадают пишу просто courses.

И после этого необходимо подождать пока создастся новый заказ await order.save(). Но когда заказ создан это означает что корзину необходимо почистить. Поэтому здесь так же вызоваю метод await req.user. который будет называться к примеру clearCart().

Перехожу models user.js.

userSchema.metods.clearCart = function(){}

И внутри этой функции я почищу this.cart = {items:[]} и скажу что items является пустым массивом. Далее я все сохраняю return this.save(). Очень простой метод. Тем не менее он позволяет полностью очистить корзину. После успешной очистки происходит перенаправление на страницу orders.hbs.

Теперь эти данные необходимо отобразить на странице «Заказы». И для этого я формирую шаблон. Заголовок страницы называю «Мои заказы». Все заказы относятся к какому – то определенному пользователю т.е. пользователь не может посмотреть заказы другого пользователя.

Но перед этим необходимо реализовать метод get в контроллере order.js. Внутри callback оборачиваю все в try catch. Далее получаю список всльзователя. Создаю переменную const orders и в ней обращаюсь к модели await Order и вызываю метод find() где в объекте прописываю условие. Условие такое что если user.userId совпадает с req.user.\_id то тогда это заказы данного пользователя. Через структуру user.userId я указываю в модели order.js на объект user.

user: {

name: String,

userId: {

type: Schema.Types.ObjectId,

ref: 'User',

required: true,

},

},

Т.е. я просто указываю где искать идентификатор пользователя. Ну а что касается req.user.\_id, то это наш текущий id. Далее обращаюсь к методу populate в параметры которого вношу user.userId.

Далее обращаюсь к функции res.render с помощью которого формирую заказы. Для этого в объекте второго параметра добавляю поле которое передаю на front-end массив orders который трансформирую с помощью orders.map() в параметры которой передаю callback функцию в которой возвращаю объект. Где по сути разворачиваю массив o.\_doc. Дальше считаю цену всего заказа price: o.courses.reduce() в параметры которой принимаю callback. Данная функция callback принимает два параметра, total – какое то тотальное значение, и значение c – какой – то конкретный курс.

Здесь единственно что немного отличается структура данных. return total += теперь у объекта **c** есть поле count, c.count где храниться количество курсов которые приобрел пользователь умноженное на **c** и дальше нужно перейти в объект course. и только после этого обратиться к полю .price.

Далее с помощью handlebars вывожу все на front-end.

«Пример страницы профиля пользователя представлен в приложении А»

Создаю страницу пользователя где будет возможность редактировать информацию о пользователе который сейчас находится в сессии а так же иметь возможность загружать картинки.

В папке views создаю файл profile.hbs. В контроллере создаю файл profile.js. В контроллер сразу подключаю промежуточное программное обеспечение для защиты маршрута const auth = require('../middleware/auth');

Создаю get запрос. Первым параметром создаю маршрут. Вторым передаю асинхронную callback функцию с параметрами req и res. И в теле данной функции отображаю страницу с помощью res.render первым параметром которая принимает добавленный маршрут profile. Вторым параметром принимаю объект в который входят поля: title: ‘Профиль’, isProfile:true. И здесь так же нужен объект пользователя что бы понимать какие данные есть у конкретного пользователя: user: req.user.toObject() что бы привести к объекту который понимает front-end.

В файле index.js подключаю данный контроллер const profileRoutes = require('./routes/profile');

Создаю страницу представления profile.hbs.

Далее обрабатываю post запрос. Для того что бы работать с файлами необходимо подключить дополнительный пакет.

* npm install multer

В папке middleware я создаю файл file.js. Данный middleware отвечает за загрузку файлов. Подключаю пакет const multer = require('multer'); Экспортирую  функцию module.exports = multer в которую ложу объект конфигурации где необходимо определить минимум два поля: const storage который будет определятся от пакета multer его метода diskStorage() т.е. здесь я определяю куда и как будут загружаться файлы на сервер.

Так же нужна еще одна переменная const fifleFilter которая присваивает callback функцию где происходит определенная валидация файлов. Данная функция принимает в себя параметры request, file и callback.

Далее регистрирую переменные storage и fileFilter.

module.exports = multer({

  storage,

  fileFilter,

});

Функция diskStorage принимает в себя объект в котором задаются определенные функции которые вызываются в процессе как файл начинает загружаться.

Первая функция которую я определю называется destination(){}. Данная функция так же принимает в себя три параметра (req, file, cb).

Вторая функция которую определяю называется filename(){} она так же в себя принимает три параметра (req, file, cb). И эта функция задает название загружаемого файла. Ну а функция destination складывает загружаемые файлы в определенную директорию.

Для того что бы выполнить данную функцию необходимо обратиться к cb() куда первым параметром передаю ошибку если она есть. Но т.е. ошибок нет то первым параметром передаю null. Вторым параметром передаю путь до папки куда буду складывать все картинки, папку назову 'images'.

Теперь что касается функции filename. здесь я так же вызываю cb() для того что бы завершить данный метод, сам multer будет вызывать этот callback, поэтому cb() необходимо определить. Где первым параметром я так же передаю ошибку, но в моем случае это null. И после этого необходимо сфоримировать название файла. Здесь очень важно что бы названия этих файлов не повторяялись. Т.е. нужно выбрать какое - то уникальное название для файла, иначе будет какая то, ошибка и пользователь может потерять свой аватарку. Самое оптимальное это выбрать текущую дату т.е. она является уникальной. И допустим совместить ее с названием оригинального файла, который загружается. Для того что бы получить уникальный префикс вызываю new Date() и вызвать метод который называется .toISOString() Далее + '- ' +  и после обращаюсь к оригинальному названию файла file.originalname. Название originalname пишется не в маленьком регистре а пишется просто одной строкой.

Функция filefilter является некоторым валидатором для файлов с помощью которой я ограничиваю расширения загружаемых файлов.

Далее создаю массив  const allowedTypes = [''] Это  массив  содержащий в себе строки тех  MIME type которые разрешены для данного файла const allowedTypes = ['image/png', 'image/jpg', 'image/jpeg'].

Ну и дальше в условии if просто обращаюсь к данному массиву allowedTypes у массива есть метод который называется includes()  и если  file.mimetype  находится в данном массиве, то все отлично и валидация пройдена.

Подключаю middleware к приложению. Перехожу в index.js Импортирую const fileMiddleware = require('./middleware/file'); Далее после того как задаю сессию, но перед тем как я задаю параметр csrf я подключаю данный middleware app.use(fileMiddleware) и у него я должен  вызвать метод  single()  т.е. я говорю что загружаю всего лишь один файл. И в этот метод я должен передать название того поля куда этот файл попадет  app.use(fileMiddleware.single('avatar')).

Загрузка файлов.

Теперь необходимо реализовать роут для того, что бы file upload у нас работал. Перехожу в profile.js. И в первую очередь перед асинхронными функциями добавляю auth. Это делается для того, что бы у нас не авторизованные пользователи не могли отправлять post запросы и соответственно получать, и просматривать страницу профиля.  Теперь обрабатываю post запрос в первую очередь оборачиваем в блок try catch.  Далее,  необходимо получить объект пользователя, у которого мы будем изменять какие - то данные. Модель User кстати необходимо импортировать  const User = require('../models/user');

И теперь я могу обратиться к модели пользователя  const user = await User. вызываю метод findById() куда я передаю id текущего пользователя из объекта req.user.\\_id.

После получения пользователя я буду у него изменять определенные поля. В конце я сделаю Object.assign(user,) Вторым параметром я передаю объект, к примеру, который будет называться toChange который я сейчас создам. И после этого ждем когда пользователь сохранится в БД await user.save(). После того как пользователь сохранится делаю res.redirect('/profile')

Теперь что касается const toChange = {} поле \*\*name: которое я буду получать из объекта req.body.name. Т.е. оно береться из views profile.hbs с

И дальше пишу проверку. Если if в объекте req.file, то тогда объекту toChange. его полю avatarUrl  будем заносить значение  пока пустой строки.

Для начало делаю console.log(req.file) для того что бы убедится в том что он у нас существует и что я действительно подключил muter правильно. и в модели пользователя добавлю еще одно свойство avatarUrl потому что сейчас его нет. models user.js добавляю avatarUrl: String, он не обязательный.

Далее реализую представление profile.hbs.

* 1. Разработка интерфейса программного продукта

Скриншоты всех пользовательских представлений представлены в ПРИЛОЖЕНИИ С.

Интерфейс – часть некоторой системы, предназначенная для обеспечения взаимодействия с ней человека либо другой системы.

В юзабилити как нигде кстати подходит выражение “Встречают по одежке а провожают по уму”. Заключается это в том что любого пользователя посещающего сайт, в первую очередь привлекает дизайн. Вторым аспектом привлекательности сайта является простота его в использовании. За это как раз таки отвечает качественно продуманная логика приложения в которую входят сообщения об ошибках, валидация формы и т.д.

Когда не зарегистрированный пользователь переходит по ссылке в «Информационную систему интернет магазина» он попадает на страницу логина.

Область навигации данной страницы состоит из четырех ссылок и картинки svg которая так же является ссылкой. При клике на ссылки происходит смена цвета ссылок с белого на желтый, тем самым оповещая пользователя о том, что он совершает действие перехода.

Переключения между формами регистрации и логина происходят посредствам двух tab «Войти в магазин» и «Регистрация пользователя». При нажатии на tabs, происходит смена стилей. Шрифт становится желтым, а цвет фона прозрачным, появляется рамка. Такие же стили применены к кнопке «Войти» и «Зарегистрировать».

При наведении курсора на input так же происходит внешнее подсвечивание данных областей ввода.

Рассмотрим пример регистрации пользователей.

При неправильном вводе пользователем необходимой информации в области ввода, он должен информироваться сообщением об ошибке. Если же это не будет происходить пользователь просто не сможет понять, что от него требует приложение.

Данные пользовательские представления рассмотрены в ПРИЛОЖЕНИИ С. Конкретно с примера С.3 по С.9.

Для повышения юзабилити упрощаю требования к паролям. Пароль должен вводится только в нижнем регистре и без пробелов, содержать символьно-буквенные значения и не иметь пробелов. Однако пользователь об этом не задумывается при вводе пароля и может допустить массу не точностей при вводе. Каждый раз выводить сообщение об ошибке просто отпугнет пользователя и как вариант он больше не будет посещать данный ресурс. Для того что бы это не происходило необходимо все буквенно-символьные значения в автоматическом режиме перевести в нижний регистр и убрать лишние пробелы. Что в принципе и происходит.

Не зарегистрированный пользователь

Подвал сайта (Footer) является областью контактной информации и расположена на всех страницах как для не зарегистрированных пользователей, так и зарегистрированных.

Большинство картинок в приложении SVG формата, кроме фоновых картинок. SVG – это векторная графика широко применяемая в полиграфии.

Преимущества юзабилити SVG :

* Масштабирование: в отличие от растровой графики, SVG не теряет в качестве при масштабировании, поэтому ее удобно использовать для разработки.
* Уменьшение HTTP-запросов при использовании: при использовании SVG сокращается количество обращений к серверу, соответственно увеличивается скорость загрузки сайта.
* Стайлинг и скриптинг: при помощи CSS можно менять параметры графики на сайте, например фон, прозрачность или границы.
* Анимация и редактирование при помощи JS.

Все картинки SVG помещены в один файл sprite размер которого составляет 35кб.

* 1. Разработка алгоритмов реализации вариантов использования

«Пример добавление курса представлен в приложении А»

* Добавить курс

Каждый зарегистрированный пользователь «Интернет магазина курсов» имеет возможность добавления своей собственной карточки курса, с последующей возможностью продажи данного курса.

Когда зарегистрированный и авторизованный пользователь проявляет желание добавить новый курс для продажи он переходит по ссылке «Добавить курс», тем самым совершая get запрос на перенаправление и отрисовку новой страницы в модели представления. Одной из таких страниц является страница «Добавить курс».

Код модели представления views/add.hbs представлен в ПРИЛОЖЕНИИ Б.

При создании формы в add.hbs прописываю необходимые атрибуты:

action =”/add” через данный атрибут указываю обработчику путь к форме которую он должен обработать. Далее указываю каким методом данная форма должна отправлять значения, введенные пользователем method = “POST”.

Форма состоит из трех input посредством которых происходит отправка введенных пользователем данных на сервер. Однако для того что бы пользователь понял за какой аспект отвечает та или иная часть ввода input, ее необходимо обозначить текстовым сообщением. За это в теге input отвечает атрибут placeholder. Данный атрибут выводит в поле input то значение которое ему было присвоено:

* Введите название курса
* Введите цену курса
* Введите url картинки

Так же на данной странице имеется текстовая область textarea посредством которой происходит ввод html шаблона описания курса. Однако данная область не является обязательной для ввода.

Обязательное поле для ввода задается в models/course.js путем прописывания нужному объекту поля required : true.

Введенные данные пользователем попадают в атрибут value после чего происходит отправка данных значений в модель контроллер model/add.js.

В контроллере данные значения попадают в обработчик формы post. Для сохранения передаваемых значений в обработчике формы используется поле data которая принимает в себя объект конфигурации передаваемых полей, которые в свою очередь сохраняются в объекте req.body.

 data: {

        title: req.body.title,

        price: req.body.price,

        img: req.body.img,

        description: req.body.description,

      },

После сохранения данных полей происходит запись и сохранение данных полей в базу данных тем самым формируя новую модель курса. При формировании модели курса генерируется и добавляется новое поле userId. Данное поле присваивает уникальный id пользователю, создавшему данный курс.

  const course = new Course({

    title: req.body.title,

    price: req.body.price,

    img: req.body.img,

    description: req.body.description,

    userId: req.user,

  });

Для того что бы курс отобразился в модели представления со всеми внесенными полями в БД, необходимо подождать пока все поля должным образом сохраняться в БД и лишь после этого сделать перенаправление на страницу /courses.

 try {

    await course.save();

    res.redirect('/courses');

  } catch (e) {

    console.log(e);

  }

При перенаправлении, контроллер courses.js, должен заново произвести итерацию по средствам которой произведется новый поиск всех идентификаторов (id) курсов и так как количество id курсов увеличивается отображается новая созданная карточка курса.

router.get('/', async (req, res) => {

  try {

    const courses = await Course.find()

      .populate('userId', 'email name')

      .select('price title img');

    res.render('courses', {

      title: 'Курсы',

      isCourses: true,

      userId: req.user ? req.user.\_id.toString() : null,

      courses,

    });

  } catch (e) {

    console.log(e);

  }

});

* Редактировать курс

«Пример редактирования курса представлен в приложении А»

Когда пользователь, который является владельцем курса, решается внести какие-либо изменения он переходит на страницу редактирования путем нажатия на кнопку, в карточке курса, «Редактировать». Данная кнопка доступна лишь пользователю создавшего данный курс. От других пользователей данная кнопка скрыта.

Пример кода get обработки на страницу «Редактировать» представлен в ПРИЛОЖЕНИИ Б

При запросе на страницу ‘/:id/edit’ происходит проверка аутентификации пользователя. Если пользователь не проходит аутентификацию, то происходит перенаправление на главную страницу. Если же пользователь успешно проходит аутентификацию, то следующим шагом через модель Course происходит поиск курса c помощью вызываемого метода findById который принимает req.params.id т.е делает запрос к БД по параметрам id и если же данный id найден, то производится проверка пользователя на то, является ли он владельцем данного курса. Если же пользователь не проходит данную проверку, то он перенаправляется на страницу курсов. Если же пользователь проверку прошел успешно, он перенаправляется на страницу «Редактировать» ‘course-edit’.

Код модели представления views/course-edit.hbs представлен в ПРИЛОЖЕНИИ Б.

Как и в примере «Добавить курс» при создании формы указываю какое действие ей нужно обработать action=”/courses/edit”, и каким методом ее нужно обработать method=”POST”. Форма так же состоит из трех input в которых содержится атрибут placeholder оповещающий пользователя какие данные нужно вводить в тот или иной input.

Так же присутствует текстовая область textarea с атрибутом placeholder, оповещающего пользователя о том, что нужно ввести html шаблон. Однако данная область не является обязательной для ввода. Обязательное поле для ввода задается в models/course.js путем прописывания нужному объекту поля required : true.

Введенные данные пользователем попадают в атрибут value после чего происходит отправка данных значений в модель контроллер model/courses.js.

После нажатия пользователем кнопки «Редактировать» происходит post отправка формы.

Пример кода post запроса отправки формы кнопки «Редактировать»

представлен в ПРИЛОЖЕНИИ Б.

При запросе на редактирование пользователь проходит аутентификацию. Так же каждое поле проверяется на валидацию и, если какой-нибудь из input не соответствует требованиям, пользователь уведомляется сообщением об ошибке. Далее, что бы упростить себе работу, заменяю \_id из БД на просто id.

После чего провожу проверку является ли пользователь владельцем курса. Если же нет, присваиваю статус ошибки 422 и делаю перенаправление на страницу курсов.

Если же пользователь является владельцем данного курса, то код выполняется дальше. А именно через модель пользователя происходит поиск findById(id) курса в БД. После нахождения искомого курса снова происходит проверка является ли пользователь владельцем найденного курса. Если пользователь не является владельцем происходит перенаправление на страницу курсов. Однако если пользователь проходит проверку код выполняется дальше.

Далее через функцию Object.assign происходит копирование отправленных полей в первый параметр course. Вторым параметром указываю куда должны быть помещены данные скопированные поля, но так как в req.body уже имеются поля, то данные поля должны быть перезаписаны значениями новых.

Для того что бы перенаправление на страницу ‘/courses’ не произошло преждевременно прописываю await course.save(), тем самым говорю что перенаправление не должно быть осуществлено, пока все значения полей в БД не будут перезаписаны на новые и лишь только после сохранения всех полей должно произойти перенаправление на страницу ‘/courses’. Где должна произойти новая итерация поиска всех курсов c последующей отрисовкой всех имеющихся курсов.

Код перенаправления на страницу ‘/courses’ представлен в ПРИЛОЖЕНИИ Б

* Удалить курс

«Пример удаления курса представлен в приложении А»

В форме редактирования курса добавляю еще одну форму. Действие данной формы обозначаю в атрибуте action =”/courses/remove”. Далее в атрибуте method=”POST” указываю каким методом будет выполняться отправка данных с данной формы. И в форме содержится одна кнопка «Удалить курс». Однако мне нужно каким то способом передавать id курса который я хочу удалить. Для этого я использую input. Данный input я скрываю через type=”hidden”. Для того что бы определить какой id курса используется на данный момент передаю атрибут name=”id” т.е. какой курс я сейчас удаляю и в атрибут value=”{{course.id}}”, использую интерполяцию от handlebars, заношу значение id удаляемого курса.

Далее прописываю еще один скрытый input в атрибут name которого передаю используемый на данный момент ключ csrf. Данные ключи уникальны только для определенного клиента находящегося в сессии.

Пример кода формы представления удаления курса представлен в ПРИЛОЖЕНИИ Б.

При нажатии на кнопку удалить происходит обработка формы методом post т.е. в контроллере routes/courses прописываю логику которая позволяет удалить данный курс.

Пример кода post обработки формы удаления курса представлен в ПРИЛОЖЕНИИ Б

Первым параметром в обработчик события принимается обрабатываемый путь ‘/remove’. Вторым параметром принимается функция аутентификации. Третьим параметром указываю асинхронную callback функцию, принимающую в себя request и response. И в теле данной функции прописываю логику, которая позволяет удалить данный курс. Для этого через оператор await обращаюсь к модели Course у которой вызываю метод deleteOne. Данный метод в качестве параметров принимает объект, в котором указываю условие \_id: req.body.id, т.е. \_id из БД должен совпадать с req.body.id который передается с клиента. А так же userId должен совпадать с req.user.\_id. Для того что бы отслеживать ошибки в данном блоке кода, всю логику оборачиваю в блок try catch. После удаления должно происходить перенаправление на страницу ‘/courses’. Где должна произойти новая итерация поиска всех курсов c последующей отрисовкой всех имеющихся курсов.

* 1. Модульное тестирование алгоритмов реализации вариантов использования

1. Системное тестирование
   1. Функциональное тестирование
   2. Оценка безопасности
   3. Тестирование производительности
2. Экономическое обоснование эффективности внедрения
3. Охрана труда(применительно к области внедрения)
4. Промышленная экология
5. Ресурсосбережение

Заключение

Список использованной литературы

Приложения