Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Инженерно-экономический факультет

Кафедра экономической информатики

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

по дисциплине:

«Средства и технологии анализа и разработки информационных систем»

на тему:

**СЕРВИС УПРАВЛЕНИЯ БИЛЕТНО-КАССОВЫМИ ОПЕРАЦИЯМИ НА ЖЕЛЕЗНОДОРОЖНОЙ СТАНЦИИ**

Студент Тюрина К. Р.

гр. 572301

Руководитель Хомяков П. В.

Минск 2018

# ВВЕДЕНИЕ

В современном мире важность транспортных коммуникаций сложно оспорить. Железнодорожный транспорт является одним из самых важных направлений транспортного развития страны, и мы все чаще сталкиваемся с тем, что железная дорога становится жизненной необходимостью.

На сегодняшний день железные дороги существуют на всех материках, кроме Антарктиды. Рельсовая сеть покрыла всю планету. В мире не найти цивилизованного человека, который хоть раз в жизни не воспользовался бы этим видом транспорта. Однако сравнительно немного людей, даже из тех, кто часто использует железную дорогу, ясно осознает насколько это колоссальной мощности предприятие. Железнодорожный транспорт – это один из самых совершенных и могучих видов транспорта. Он является самым главным источником внутренней силы, производительности и богатства страны. От уровня развития железных дорог в большей степени зависит как военная, так и экономическая жизнь государства.

Особое назначение железнодорожного транспорта – возможность массовых перевозок пассажиров, а также грузов на большие расстояния, что делает эксплуатацию железных дорог крайне эффективным. После водного транспорта железнодорожный бесспорно является наиболее энергетически и экономически выгодным средством как передвижения, так и перевозки грузов.

После войны одним из основных направлений политики СССР стало восстановление железнодорожной отрасли. К концу 80-х годов железнодорожный транспорт достиг достаточно весомого развития: большая доля путей были электрифицированы к тому времени, что позволило в значительной мере увеличить скорость перевозок, а также их объем. Стоит отметить, что также к концу 80-х годов грузооборот увеличился в 6,5 раз, а пассажирооборот примерно в 5 раз, в сравнении с показателями послевоенного времени.

В современном мире услуги для населения, предоставляемые железнодорожным транспортом, должны идти в ногу со временем и быть качественными и способными в полной мере удовлетворять потребности, имеющиеся у населения, в том числе у грузоотправителей и получателей грузов при перевозках. Улучшению экономических показателей страны способствует в достаточном большом объеме развитие железнодорожного транспорта, так как от качественности и своевременности доставки пассажиров и грузов зависит ритмичность экономики, ее уровень сбалансированности, а также темпы ее роста. На сегодняшний день лишь железные дороги могут обеспечить достаточный объем пассажирский и грузоперевозок.

Значительную часть железнодорожного транспорта занимают именно пассажирские перевозки. Объем пассажиропотока на каждой отдельной станции огромен, и неустанно растет с каждым годом, с каждой новой инновацией в железнодорожном транспорте. В Республике Беларусь цифра перевозок пассажиров перевалила за 16091 тысячу пассажиров в отчете за январь-март 2018 года. Масштабы перевозок вынуждают работников железнодорожных станций оптимизировать взаимодействие непосредственно с пассажирами, а также работу по техническому обслуживанию составов и самих железнодорожных станций.

На данный момент проблемой большинства железнодорожных станций является необходимость вручную вести учет поездов, маршрутов и пассажиров, а также билетно-кассовых операций. Немалый объем бумажной работы не позволяет работникам станций сосредоточиться на качестве обслуживания пассажиров. Внедрение системы автоматизации управления билетно-кассовыми операциями на железнодорожной станции позволит работникам станций с легкостью регистрировать новые маршруты, проводить билетно-кассовые операции, а также больше времени уделять качеству обслуживания пассажиров.

Целью данной курсовой работы является оптимизация управления билетно-кассовыми операциями на железнодорожной станции, а также автоматизация работы самой станции с целью сокращения временных затрат на учет поездов, маршрутов и пассажиров, что позволит улучшить качество обслуживания граждан и гостей города, а также упростит дальнейшее расширение железнодорожной станции.

Для достижения поставленной цели необходимо решить следующие задачи:

* изучить работу железнодорожной станции;
* разработать логическую и физическую модели представления данных;
* выполнить функциональное моделирование процессов учета маршрутов и пассажиров школы иностранных языков на основе стандарта IDEF0;
* создать логическую и физическую модели представления данных;
* описать созданные модели с помощью UML;
* спроектировать и разработать базу данных;
* реализовать серверную часть приложения, которая позволит работать с базой данных и выполнить ряд функций, поставленных целью разработки курсового проекта;
* разработать клиентскую часть приложения, в том числе интерфейс приложения;
* протестировать программный продукт.

# ОПИСАНИЕ РАБОТЫ ЖЕЛЕЗНОДОРОЖНОЙ СТАНЦИИ

Железнодорожная станция – главная единица железнодорожного транспорта, один из видов раздельных пунктов с путевым развитием, которое позволяет поводить работы по приёму, отправке, скрещению и обгону поездов. Кроме того работы по перевозке грузов, багажа и грузобагажа и обслуживанию пассажиров, а также, при достаточном развитии путевых развязок, - маневровую работу по формированию и расформированию составов и техническое обслуживание поездов.

Каждая железнодорожная станция обязательно включает в себя путевое хозяйство, грузовое хозяйство и системы сигнализации и централизации.

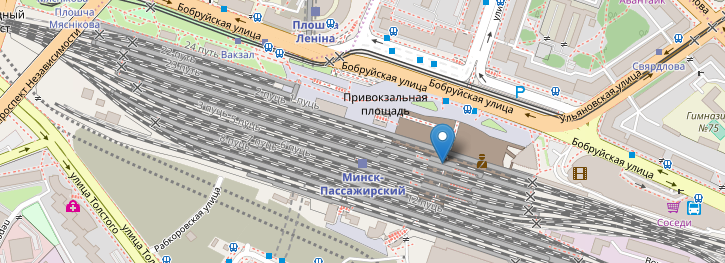


Рисунок 1.1 – Карта железнодорожного вокзала Минск-Пассажирский

На рисунке 1.1 представлена карта железнодорожного вокзала Минск-Пассажирский. На данном рисунке ясно видно, что вокзал оборудован всем необходимым для обслуживания пассажиров.

Путевое хозяйство представляет собой совокупность железнодорожных путей, объединенных в парки. Каждый парк, как и каждый путь в парке может иметь определенную специализацию. Например, сортировочный парк, приёмо-отправочный парк и так далее. На пассажирский станциях некоторые пути как правило секционированы. Это сделано для того, чтобы была возможность принимать на путь два небольших пригородных состава с разных сторон. Однако в путевом развитии станции такой путь рассматривается как одно целое.

Также на станции присутствуют тупиковые пути, имеющие с одной стороны тупиковый упор. Такие пути используются как правило в технических целях для простоя вагонов, локомотивов или даже целых составов.

Грузовые хозяйства, в отличие от путевых, используются для проведения грузовых работ. Они состоят из погрузочно-выгрузочных путей, складов, терминалов, сортировочных станций и тому подобное. Как правило грузовые хозяйства строятся отдельно от пассажирских станции, зачастую даже в нескольких километрах от нее.

Системы сигнализации и централизации, как уже было сказано ранее, обязательно присутствуют на всех железнодорожных станциях. Они представляют собой совокупность стрелок и светофоров, обеспечивающих безопасное управление движением поездов. На рисунке 1.2 представлен пример железнодорожной стрелки.



Рисунок 1.2 – Железнодорожная стрелка

По цели использования и характеру работы железнодорожные станции делятся на:

* промежуточные;
* участковые;
* сортировочные;
* грузовые;
* пассажирские.

Промежуточные станции выполняют функции небольших вокзалов для приема, отправления и пропуска поездов. Станции, на которых выполняется грузовая работа данного участка, называются опорными.

Участковые железнодорожные станции находятся на границах участков железнодорожных линий. Такие станции используются в основном для смены локомотива, локомотивной бригады, а также технического обслуживания подвижного состава.

Сортировочные станции имеют очень важное значение во всем железнодорожном транспортном сообщении в целом. На таких станциях выполняют работы по массовому формированию и расформированию грузовых поездов.

Железнодорожные станции и железнодорожный транспорт в целом требует постоянного развития. По официальным данным национального статистического комитета Республики Беларусь пассажирские перевозки по железнодорожному транспорту уступают только автомобильным (автобусным) перевозкам (рисунок 1.3).



Рисунок 1.3 – Пассажирские перевозки по видам транспорта за I квартал 2018 г.

На рисунке 1.4 представлена схема Белорусской железной дороги.



Рисунок 1.4 – Схема Белорусской железной дороги

Железнодорожный вокзал Минск-Пассажирский является самым крупным вокзалом в столице республики и одним из самых крупных в стране в целом. На 2015 год кроме рейсов внутри страны вокзал осуществляет провоз пассажиров до следующих стран:

* Австрия
* Болгария
* Венгрия
* Германия
* Казахстан
* Латвия
* Литва
* Молдавия
* Нидерланды
* Польша
* Россия
* Румыния
* Словакия
* Украина
* Франция
* Хорватия
* Чехия
* Швейцария.

# ОПИСАНИЕ ПРОЦЕССА ПОКУПКИ БИЛЕТА ПАССАЖИРОМ НА ЖЕЛЕЗНОДОРОЖНОЙ СТАНЦИИ

После проведенного анализа предметной области представим функциональную модель процесса покупки билета на поезд пассажиром.

Для начала представим модель «как есть», отражающую реальный, используемый на данный момент процесс покупки билета на рейс.

Входными данными являются:

* оплата за билет;
* заявка на покупку билета.

Ресурсом, который необходим для успешного завершения процесса, является администратор железнодорожной станции.

Основным процессом данного курсового проекта является покупка пассажиром билета на поезд. В соответствии с этим на рисунке 2.1 представлена контекстная диаграмма функциональной модели.

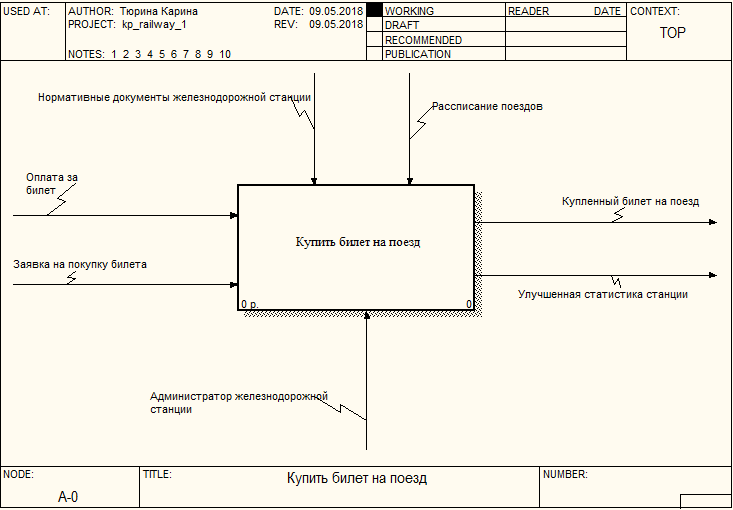


Рисунок 2.1 – Контекстная диаграмма верхнего уровня процесса «Купить билет на поезд» модели «как есть»

На рисунке 2.2 представлена декомпозиция процесса «Купить билет на поезд». Данный процесс включает в себя следующие этапы:

* подобрать направление;
* оплатить билет;
* скорректировать статистику станции.

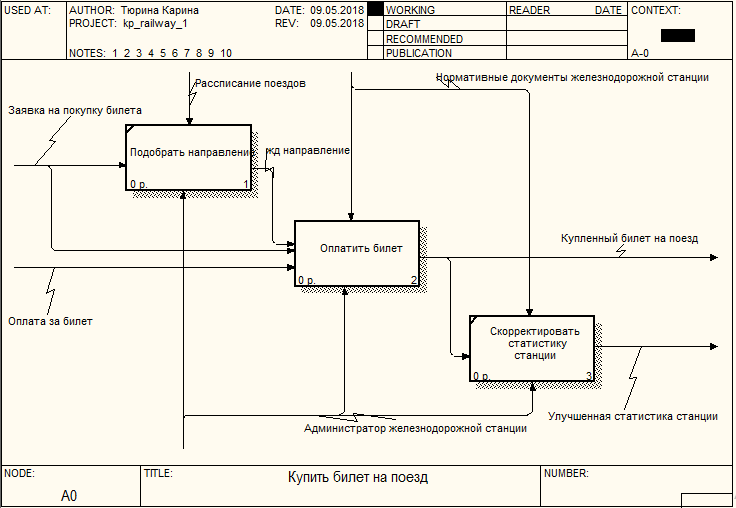


Рисунок 2.2 – Декомпозиция процесса «Купить билет на поезд»

Рисунок 2.4 представляет декомпозицию процесса «Оплатить билет». Для успешного завершения данного этапа необходимо выполнить следующие действия:

* забронировать билет на поезд;
* произвести оплату билета;
* выдать пассажиру билет на поезд.

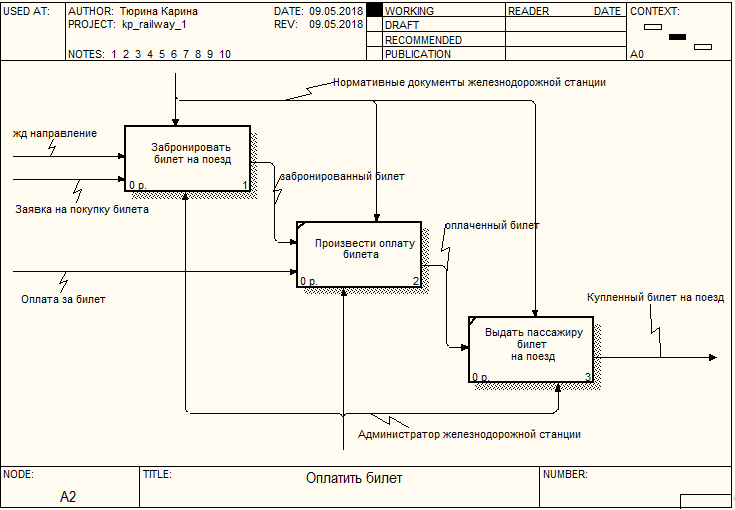


Рисунок 2.4 – Декомпозиция процесса «Оплатить билет»

На рисунке 2.5 представлена декомпозиция процесса «Забронировать билет на поезд». Данный процесс не может быть завершен без следующих этапов:

* отобрать поезда в соответствии с датой;
* выбрать поезд в соответствии с заявкой;
* произвести бронирование билета на поезд.

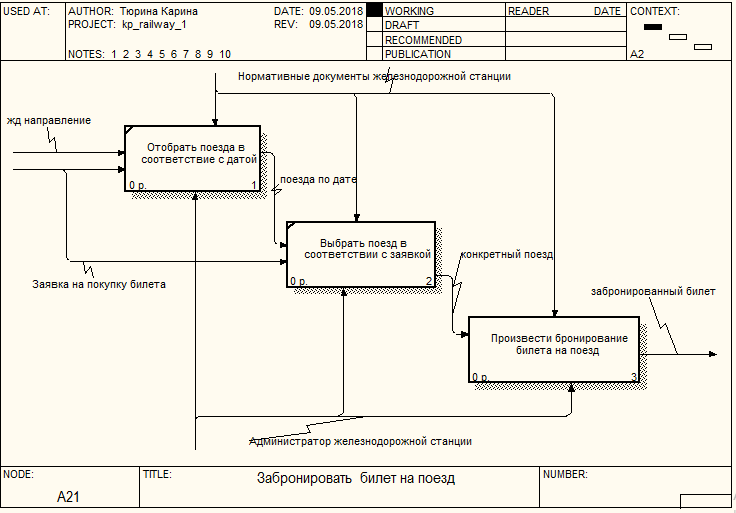


Рисунок 2.5 – Декомпозиция процесса «Забронировать билет на поезд»

Процесс бронирования билета на поезд, декомпозиция которого представлена на рисунке 2.6, состоит из следующих этапов:

* заполнить форму бронирования билета на поезд;
* отправить запрос в базу данных для сохранения бронирования билета на поезд;
* уведомить пассажира об успешном бронировании билета.

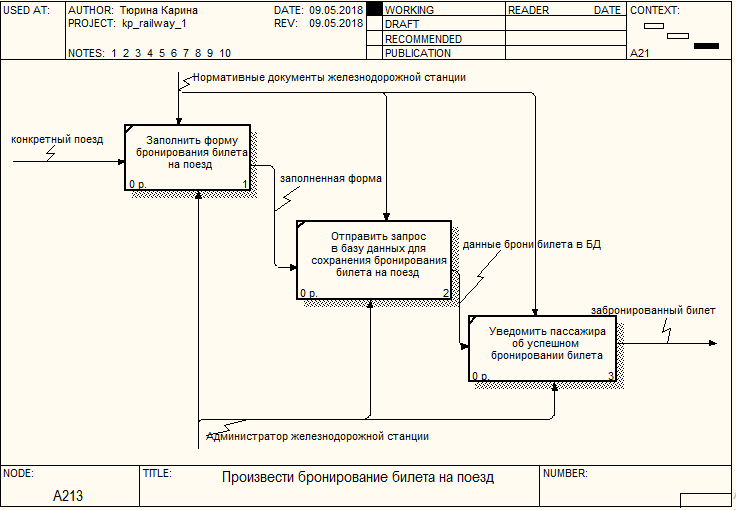


Рисунок 2.6 – Декомпозиция процесса «Произвести бронирование билета на поезд»

С помощью представленной диаграммы IDEF-0, отражающей модель «как есть», можно более подробно изучить процесс покупки пассажиром билета, в том числе и его подпроцессы [6].

Для более тщательного изучения процесса рассмотрим модель «как должно быть» процесса покупки билета на поезд, отражающую улучшения в текущем варианте процесса, описанном выше в модели «как есть».

На рисунке 2.7 представлена контекстная диаграмма верхнего уровня процесса «Купить билет на поезд».

Исходными данными для данного варианта диаграммы является:

* заявка на покупку билета.

В отличие от модели «как есть», ресурсом для успешного завершения данного процесса является пассажир.

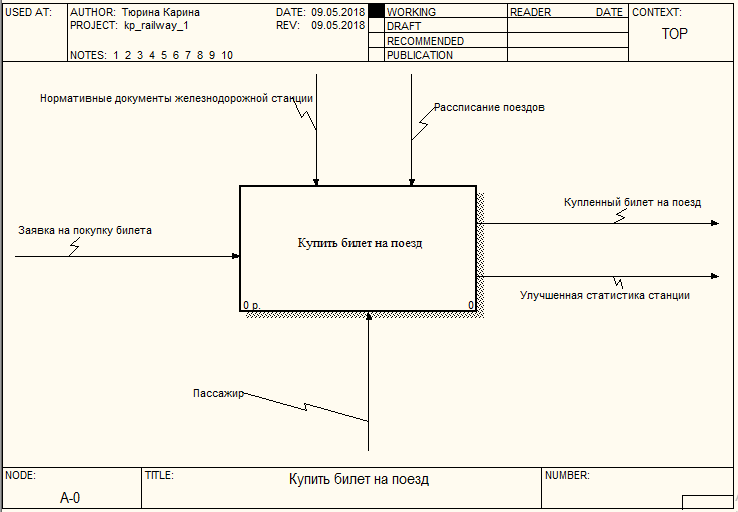


Рисунок 2.7 – Контекстная диаграмма верхнего уровня процесса «Купить билет на поезд» модели «как должно быть»

На рисунке 2.8 представлена декомпозиция процесса «Купить билет на поезд». Данный процесс в модели «как должно быть» состоит из следующих этапов:

* подобрать направление;
* оплатить билет.

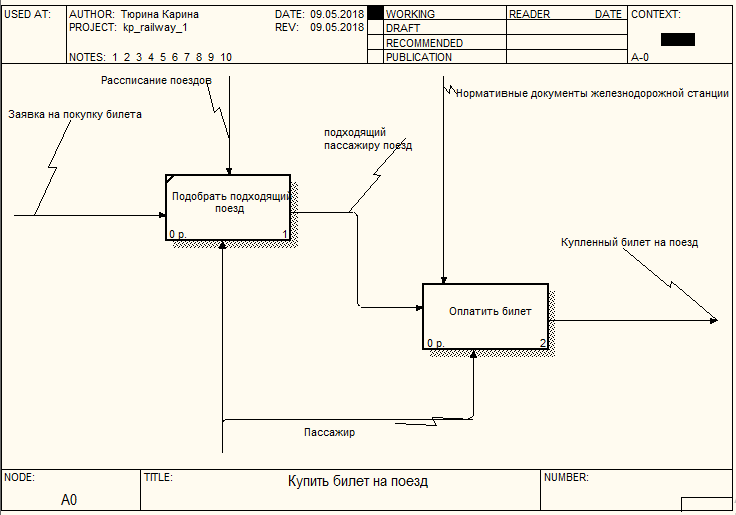


Рисунок 2.8 – Декомпозиция процесса «Купить билет на поезд»

Процесс «Оплатить билет» в модели «как должно быть», декомпозиция которого представлена на рисунке 2.9, не может быть успешно завершен без следующих этапов:

* произвести оплату билета;
* распечатать билет на поезд.

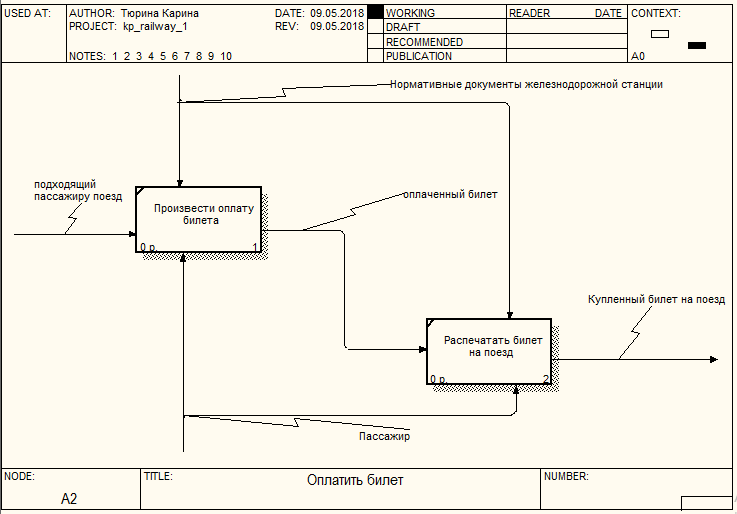


Рисунок 2.9 – Декомпозиция процесса «Оплатить билет» в модели «как должно быть»

Рисунок 2.10 отражает декомпозицию процесса «Произвести оплату билета в модели «как должно быть». Данный процесс включает в себя следующие этапы:

* произвести оплату билета;
* распечатать билет на поезд.

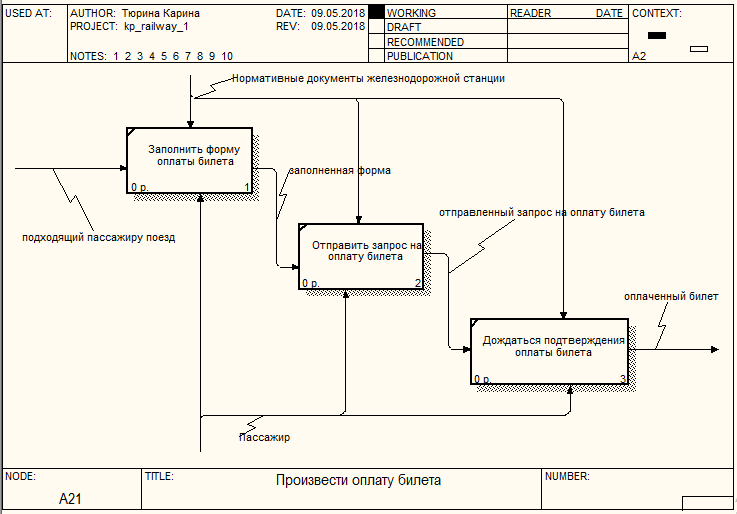


Рисунок 2.10 – Декомпозиция процесса «Произвести оплату билета» модели «как должно быть»

С помощью представленной диаграммы IDEF-0, отражающей модель «как должно быть», можно более подробно изучить процесс покупки пассажиром билета, а также его подпроцессы [6].

Проведенное функциональное моделирование показывает, что, позволив пассажиру самому управлять процессом покупки билета, можно значительно уменьшить затраты на процесс покупки билета на поезд и освободить время работников железнодорожной станции для улучшения качества обслуживание пассажиров по ходу следования поезда.

# СПЕЦИФИКАЦИЯ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ СИСТЕМЫ

Для более подробного изучения спецификации системы управления билетно-кассовыми операциями на железнодорожной станции рассмотрим диаграмму вариантов использования, представленную на рисунке 3.1 и отражающую отношения между актерами и вариантами использования [7].

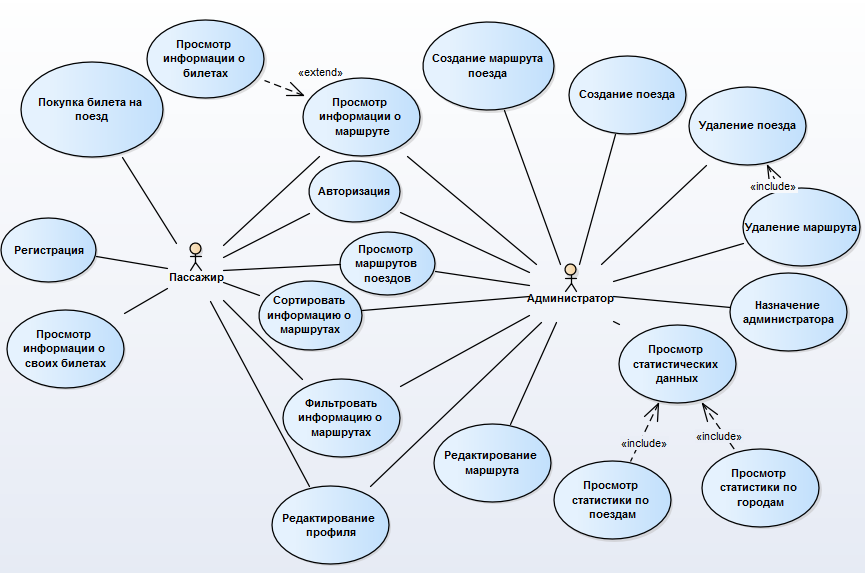


Рисунок 3.1 – Диаграмма вариантов использования системы управления билетно-кассовыми операциями на железнодорожной станции

На диаграмме представлены два актера: пассажир и администратор.

Пассажир имеет следующие варианты использования:

* регистрация;
* покупка билета на поезд;
* просмотр информации о маршруте с возможностью просмотра информации о билетах;
* авторизация;
* просмотр маршрутов поездов;
* сортировка информации о маршрутах;
* фильтрация информации о маршрутах;
* просмотр информации о своих билетах;
* редактирование профиля;
* удаление профиля.

Роль администратора позволяет осуществлять следующие варианты использования:

* авторизация;
* редактирование профиля;
* создание маршрута поезда;
* создание поезда;
* назначение администратора;
* просмотр статистических данных по городам и поездам;
* просмотр маршрутов поездов;
* редактирование маршрутов поездов;
* удаление профиля;
* удаление поезда;
* удаление маршрута;
* сортировка информации о маршрутах;
* фильтрация информации о маршрутах;
* просмотр информации о маршруте, в том числе информации о билетах.

# ИНФОРМАЦИОННАЯ МОДЕЛЬ СИСТЕМЫ И ЕЕ ОПИСАНИЕ

В результате проектирования системы управления билетно-кассовыми операциями на железнодорожной станции мною были выделены следующие сущности:

* Пассажир;
* Администратор;
* Поезд;
* Маршрут;
* Билет.

Так как пассажир и администратор отличаются только правами управления системы, их можно объединить в одну сущность Пользователь и добавить поле «роль».

В результате получим следующие сущности:

* Пользователь;
* Поезд;
* Маршрут;
* Билет.

Информационная модель системы представлена на рисунке 4.1.

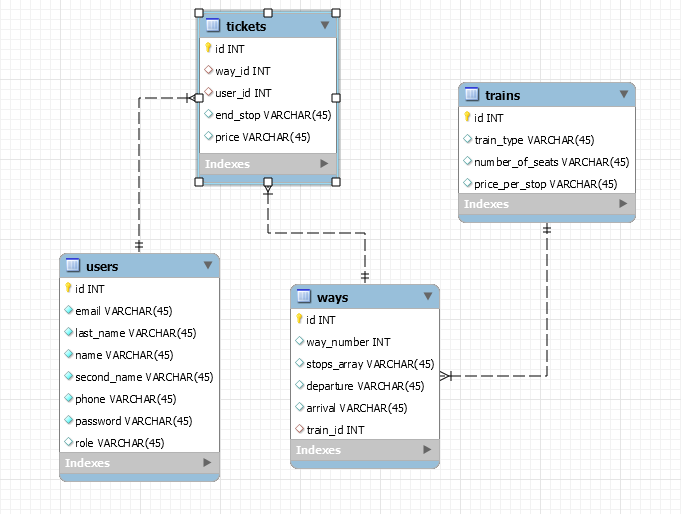


Рисунок 4.1 – Информационная модель системы управления билетно-кассовыми операциями на железнодорожной станции

Сущность Пользователь, которой соответствует таблица users в информационной модели системы, необходима для хранения информации о пассажирах, а также администраторах системы. Данная сущность содержит следующие атрибуты:

* id – уникальный идентификатор пользователя;
* email – электронная почта пользователя;
* last\_name – фамилия пользователя;
* name – имя пользователя;
* second\_name – отчество пользователя;
* phone – телефон пользователя;
* password – пароль для входа в систему;
* role – роль пользователя.

Сущности Поезд соответствует таблица trains в информационной модели системы. Данная сущность используется для хранения информации о созданных составах. Сущность Поезд содержит следующие атрибуты:

* id – уникальный идентификатор поезда;
* train\_type – тип состава;
* number\_of\_seats – количество мест для пассажиров;
* price\_per\_stop – цена билета за остановку.

Сущности Маршрут, которой соответствует таблица ways в информационной модели, используется для хранения информации о маршрутах: их времени отправления и прибытия, а также остановках на пути следования. Сущность Поезд состоит из следующих атрибутов:

* id – уникальный идентификатор маршрута;
* way\_number – номер маршрута;
* stops\_array – массив остановок на пути следования поезда;
* departure – время и дата отправления поезда;
* arrival – время и дата прибытия поезда в конечный маршрут;
* train\_id – ссылка на объект сущности Поезд.

Сущность Билет самая важная среди остальных сущностей, так как содержит в себе информацию о купленных билетах и связывает между собой сущности Пользователь и Маршрут. Данная сущность соответствует таблице tickets в информационной модели и содержит следующие атрибуты::

* id – уникальный идентификатор билета;
* way\_id – ссылка на объект сущности Маршрут;
* user\_id – ссылка на объект сущности Пользователь;
* end\_stop – конечная остановка;
* price – цена билета.

Данная информационная модель приведена к третьей нормальной форме. Об этом свидетельствует то, что не ключевой атрибут сущности функционально зависит только от всего первичного ключа и ни от чего другого [1].

# ОБОСНОВАНИЕ ВЫБОРА КОМПОНЕНТОВ И ТЕХНОЛОГИЙ ДЛЯ РЕАЛИЗАЦИИ КУРСОВОГО ПРОЕКТА

Для реализации поставленной задачи курсового проекта, мною были выбраны следующие технологии:

* язык Ruby и фреймворк Ruby on Rails (RoR) для реализации серверной части приложения;
* HTML, CSS, JavaScript, фреймворк Bootstrap, а также формат erb с возможностью вставки ruby кода на страницу для реализации клиентской части приложения;
* в качестве сервера базы данных была выбрана СУБД PostgreSQL.

Ruby – динамический, интерпретируемый, высокоуровневый язык для удобного и быстрого объектно-ориентированного программирования [3]. Ruby on Rails – многоуровневый, полнофункциональный фреймворк для создания качественных веб-приложений [4]. Данный фреймворк основан на архитектуре Модель-Представление Контроллер (MVC).

Основное преимущество разработки на фреймворке RoR - это скорость с высоким качеством реализуемого приложения. Кроме того, язык Ruby является легко расширяемым языком. Все библиотеки языка Ruby с открытым исходным кодом и каждый разработчик может внести свой вклад в развитие любого из них. Репозитории библиотек (именуемых гемами, с англ. gems) можно найти на сайте github.com, а список всех гемов на сайте rubygems.org [2].

Как представлено на рисунке 5.1, Ruby on Rails является одним из самых популярных веб-фреймворков в мире.

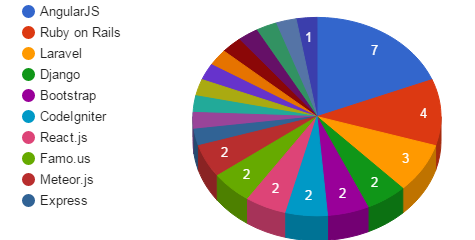


Рисунок 5.1 – Круговая диаграмма по популярности веб-фреймворков

Язык Ruby также не уступает в количестве разрабатываемых приложений. Как показано на рисунке 5.2 данный язык программирования входит в десятку самых популярных языков мира.

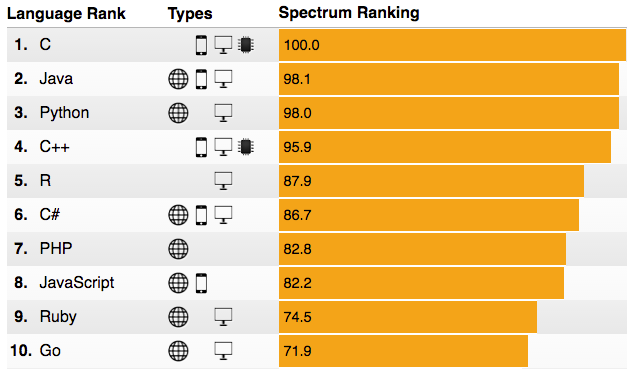


Рисунок 5.2 – График популярности языков программирования

Языки HTML, CSS и JavaScript на сегодняшний день стали уже неотъемлемой частью любого веб-приложения. Данные языки позволяют разрабатывать современные динамические интерфейсы, реализующие все необходимое для приложений любой сложности. Кроме того такие приложения получаются кроссплатформенными и требуют только наличия современного браузера у пользователя.

В качестве сервера базы данных я выбрала свободную объектно-реляционную систему управления базами данных PostgreSQL, основанную на языке SQL. Тот факт, что данная СУБД является не просто реляционной, а объектно-реляционной, дает ей значительное преимущество перед конкурентными СУБД. Благодаря этому, в PostgreSQL есть поддержка пользовательских объектов, а также их поведения, типов данных, функций и тому подобное. Кроме того, именно данная СУБД поддерживает все возможные типы данных, в том числе сетевых адресов, битовых строк, xml, json, массивов, диапазонов и многих других. Также БД в PostgreSQL гарантированно дает уверенность в обеспечении целостности данных: без сомнений сохранены будут только корректные данные [1].

На рисунке 5.3 представлена диаграмма последовательности процесса покупки билета на поезд. На диаграмме можно более подробно взаимодействие пассажира и администратора железнодорожной станции при покупке пассажиром билета на поезд. При оптимизации данного процесса роль администратора может выполнять разработанное приложение.

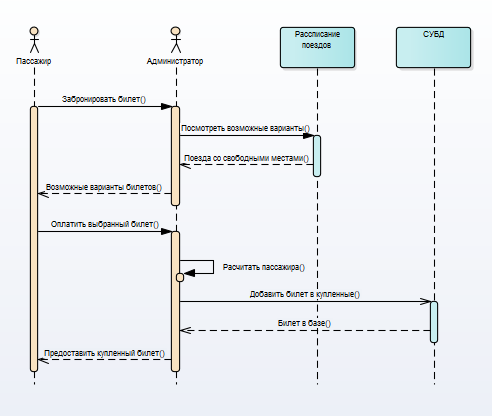


Рисунок 5.3 – Диаграмма последовательности процесса покупки пассажиром билета на поезд

На рисунке 5.4 представлена диаграмма компонентов системы управления билетно-кассовыми операциями на железнодорожной станции.

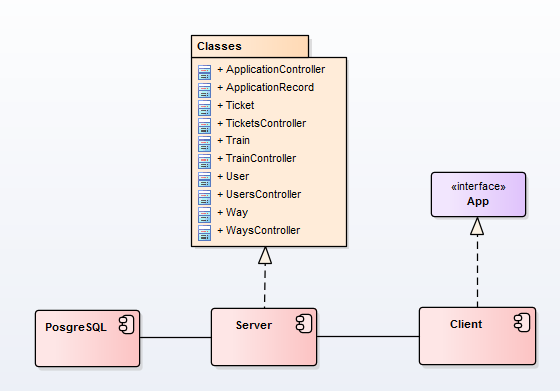


Рисунок 5.4 – Диаграмма компонентов системы управления билетно-кассовыми операциями на железнодорожной станции

На представленной диаграмме компонентов системы можно увидеть разбиение программного продукта на структурные компоненты, а также как они связаны между собой.



Рисунок 5.5 – Диаграмма развертывания системы управления билетно-кассовыми операциями на железнодорожной станции

На рисунке 5.5 представлена диаграмма развертывания системы управления билетно-кассовыми операциями на железнодорожной станции. Данная диаграмма позволяет наглядно представить взаимодействие компонентов в распределенной среде.

# МОДЕЛИ ПРЕДСТАВЛЕНИЯ СИСТЕМЫ И ИХ ОПИСАНИЕ

Для более подробного описания системы необходимо также рассмотреть следующие элементы:

* диаграмма состояний;
* блок схема алгоритма расчета аналитических данных для графика популярности направлений;
* блок схема работы всей программы.

На рисунке 6.1 представлена диаграмма состояний при покупке билета пассажиром.

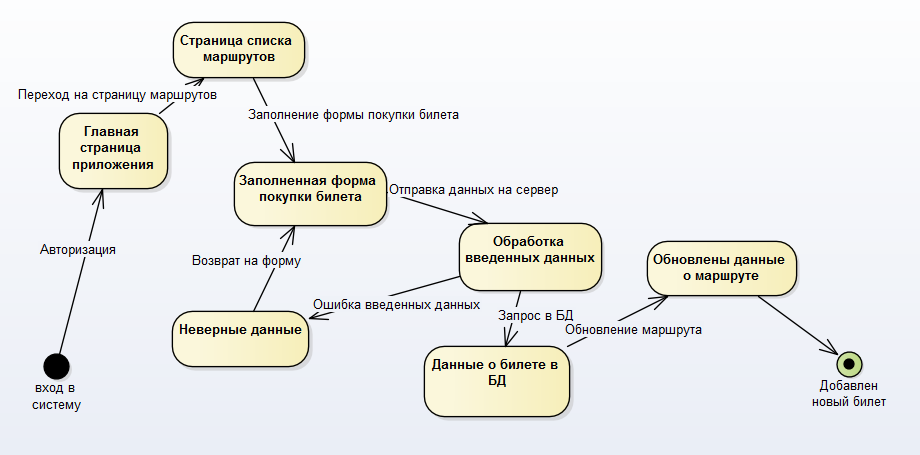


Рисунок 6.1 – Диаграмма состояний при покупке билета пассажиром

Пассажир авторизуется на сайте и попадает на главную страницу приложения. Затем пользователю необходимо перейти на страницу со списком возможных маршрутов и найти подходящий поезд. После этого пользователь должен заполнить форму покупки билета, которая отправиться на сервер для проверки. В случае неправильных данных с сервера вернется ошибка и приложение вернется на страницу заполнения формы. Если данные введены верно в базу данных будет отправлен запрос на добавление данных о билете, а также запрос на обновление данных маршрута.

На рисунке 6.2 представлена блок-схема алгоритма расчета аналитических данных для графика популярности направлений.

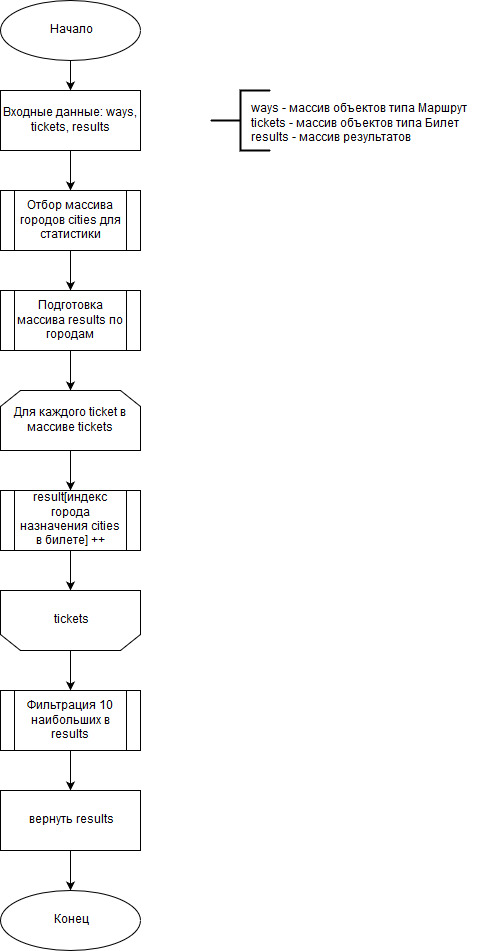


Рисунок 6.2 – Блок-схема алгоритма расчета аналитических данных для графика популярности направлений

На рисунке 6.3 представлена блок-схема алгоритма работы всей программы.

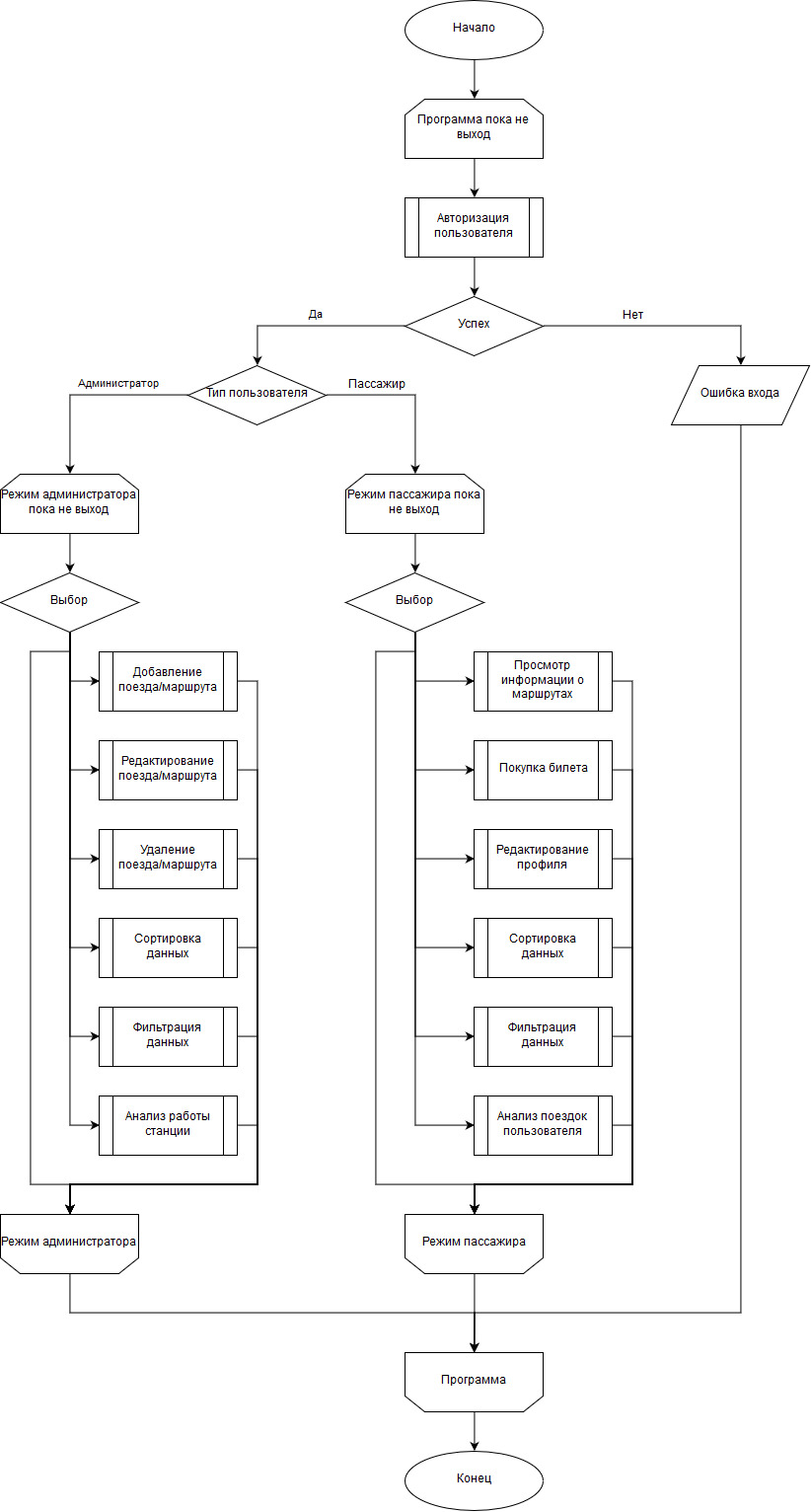


Рисунок 6.3 – Блок-схема алгоритма работы всей программы

# ОПИСАНИЕ ПРИМЕНЕНИЯ ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

В целях оптимизации разрабатываемого кода мною были выбраны следующие паттерны проектирования:

* Модель-Представление-Контроллер (MVC) для оптимизации структуры проекта в целом;
* Заместитель (Proxy) для улучшенного контроля доступа в БД;
* Посредник (Mediator) для связи серверной части приложения с клиентом.

Шаблон Модель-Представление-Контроллер представляет собой разделение данных приложения, пользовательского интерфейса и логики приложения между собой. Суть данного паттерна в том, чтобы каждый из компонентов программы: модель, представление и контроллер могли существовать независимо друг от друга. Более того каждый компонент может изменяться не затрагивая другие компоненты и не меняя при этом общей структуры проекта. На рисунке 7.1 представлена общая структура паттерна «Модель-Представление-Контроллер».



Рисунок 7.1 – Структура шаблона «Модель-Представление-Контроллер»

Данный паттерн очень популярен среди веб-разработчиков. Многие фреймворки, в том числе и Ruby on Rails, построены на основе данного шаблона, что позволяет веб-разработчикам строить распределенные приложения быстро и качественно.

Паттерн Заместитель, с английского Proxy, - это структурный шаблон проектирования, который необходим для контроля доступа к другому объекту. В разрабатываемом приложении данный паттерн используется для контроля доступа к объектам базы данных. Класс ApplicationRecord и его наследники Train, Way, Ticket и User представляют собой так называемые Proxy-классы и перехватывают все попытки доступа к базе. Такой подход позволяет разместить базу на удаленном сервере, а также контролировать запросы и отправлять непосредственно в базу только заведомо корректные [5].

Достоинства шаблона Заместитель:

* удаленный доступ;
* виртуальный заместитель;
* защищает доступ к объектам.

Недостатки шаблона Заместитель:

* увеличенное время доступа к объектам.

Поведенческий паттерн Посредник, или Mediator, необходим для связи компонентов приложения, при этом не вынуждая их явно ссылаться друг на друга. Так паттерн медиатор инкапсулирует взаимодействие объектов, выступая при этом посредником между ними [5].

На рисунке 7.2 представлена структура паттерна Посредник.

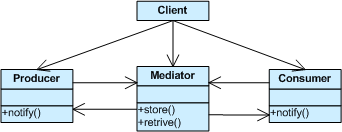


Рисунок 7.2 – Структура паттерна Посредник

На рисунке 7.3 представлена диаграмма классов системы управления билетно-кассовыми операциями на железнодорожной станции, демонстрирующая применение описанных выше паттернов проектирования.

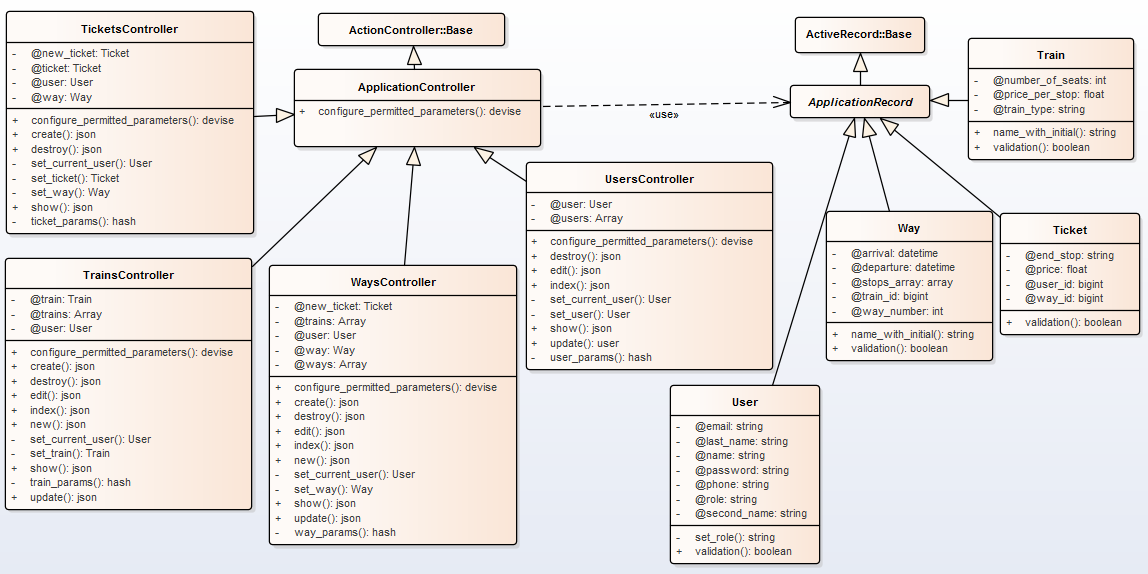


Рисунок 7.1 – Диаграмма классов системы управления билетно-кассовыми операциями на железнодорожной станции

# РУКОВОДСТВО ПО РАЗВЕРТЫВАНИЮ СИСТЕМЫ

Ниже приведенная инструкция требует операционной системы Linux, например Ubuntu 16.04.

Так как разрабатываемый программный продукт представляет собой веб-приложение, необходимо развернуть программу на хостинге. Для этого необходимо зарегистрироваться на одном из таких хостингов, например DigitalOcean, и заказать виртуальную машину с операционной системой Linux. Данная виртуальная машина получит ip адрес, что позволит пользователям интернета заходить на сайт. Затем при желании можно купить доменное имя, чтобы пользователям удобнее было запомнить как заходить на портал, не запоминая при этом неудобный для него формат ip адреса. Также, при желании, можно заказать разные виртуальные машины для каждого отдельного компонента системы: сервера базы данных, сервера приложения и клиентской части приложения [8].

Для корректной работы сервера базы данных на предоставленной хостингом виртуальной машине необходимо выполнить следующие шаги в терминале:

* sudo apt-get update – для обновления пакетов;
* sudo apt-get install postgresq postgresql-contrib – установка необходимых пакетов;
* psql – переход в консоль базы postgres;
* запустить выполнение sql-скрипта для создания БД.

Корректная работы сервера приложения требует следующих шагов:

* sudo apt-get update – для обновления пакетов;
* установить rbenv, ruby, rails;
* настроить nginx – сервер приложений для корректного доступа к серверу приложения из сети интернет;
* развернуть приложение;
* отредактировать файл config/database.yml для корректного доступа к базе;
* запустить сервер nginx.

Развертывание клиентской части приложения во многом похоже на развертывание сервера с небольшим отличием: не нужно настраивать доступ к базе данных, а также необходимо установить только те пакеты, которые требует сервер клиента. Развертывание клиента написаного на Angular.js требует следующих этапов:

* sudo apt-get install npm – установка менеджера пакетов для Node.js;
* npm install angular – установка Angular.js;
* npm start.

Также в коде проекта необходимо указать ip адрес сервера приложения, чтобы все запросы, отправленные на сервер по протоколу http приходили на правильный хост.

Выполнение описанных выше шагов позволит развернуть приложение в распределенной среде, и каждый пассажир сможет найти необходимый ему поезд, используя приложение с любого устройства, имеющего доступ в интернет.

# РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ РАЗРАБОТАННОЙ СИСТЕМЫ УПРАВЛЕНИЯ БИЛЕТНО-КАССОВЫМИ ОПЕРАЦИЯМИ НА ЖЕЛЕЗНОДОРОЖНОЙ СТАНЦИИ И ОЦЕНКА ВЫПОЛНЕНИЯ ЗАДАЧ

Разработанное веб-приложение представляет собой систему управления билетно-кассовыми операциями на железнодорожной станции. Приложение позволяет войти в систему под двумя типами пользователя: администратора и пассажира. При входе на сайт главная страница сайта (рисунок 9.1).

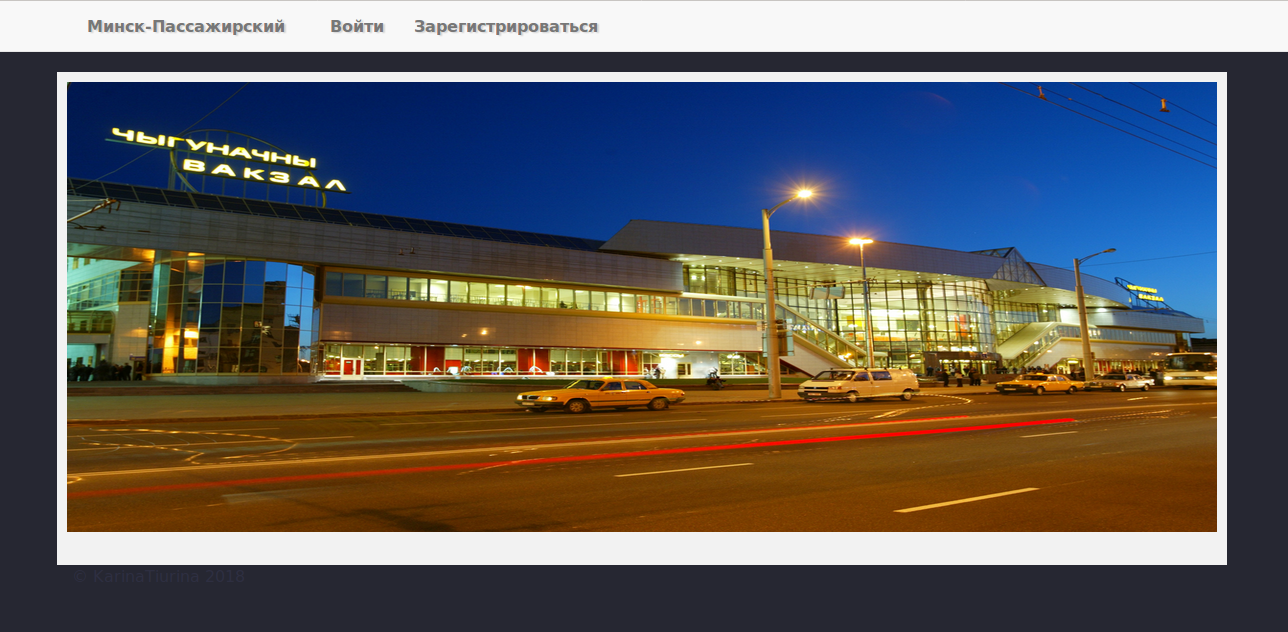


Рисунок 9.1 – Главная страница веб-приложения

Для входа в систему необходимо заполнить зарегистрироваться, и затем войти в систему. На рисунках 9.2, 9.3 изображены соответствующие формы. На рисунке 9.4 изображена ошибка валидации данных при регистрации.

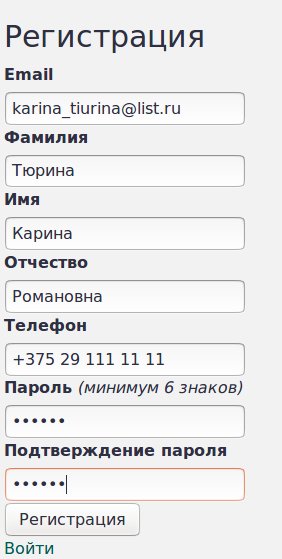


Рисунок 9.2 – Форма регистрации на сайте

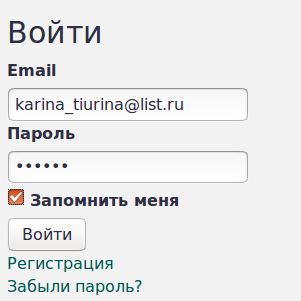


Рисунок 9.3 – Форма входа на сайт

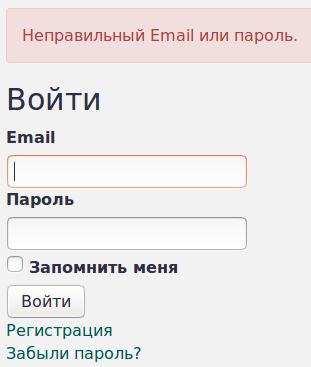


Рисунок 9.4 – Ошибка ввода данных при регистрации

При регистрации система автоматически даст пользователю права пассажира, назначить администратора может только другой администратор. Также при регистрации пользователь увидит приветственное сообщение (рисунок 9.5).

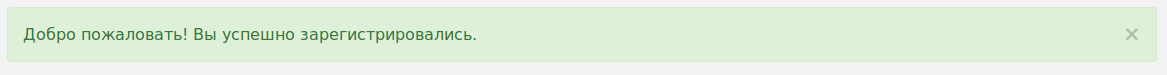


Рисунок 9.5 – Приветственное сообщение при регистрации

После входа в систему пользователь может просматривать, а также редактировать свой профиль (рисунок 9.6, 9.7).

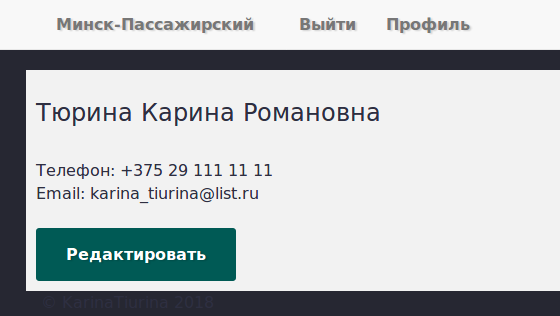


Рисунок 9.6 – Профиль пользователя

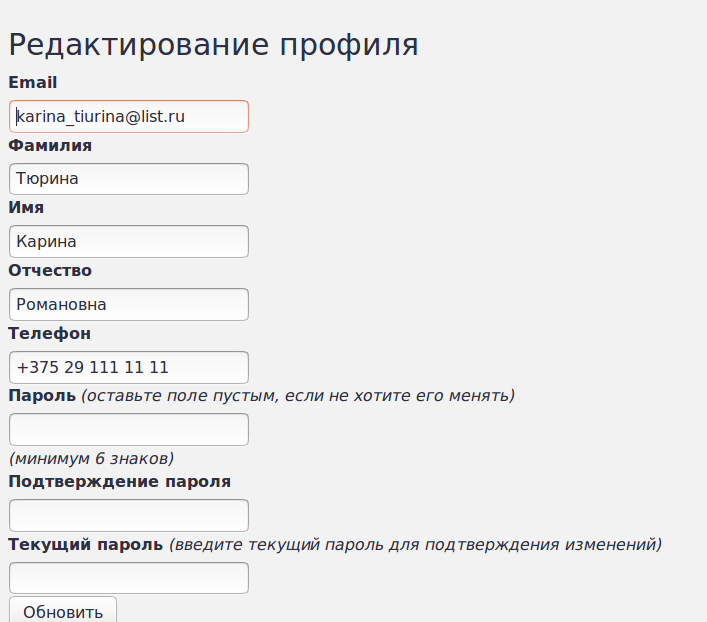


Рисунок 9.7 –Редактирование профиля

Роль администратора позволяет добавлять, редактировать, удалять поезда. На рисунке 9.8 представлена реализация данных функций.



Рисунок 9.8 – Таблица поездов

На рисунке 9.9 представлена таблица маршрутов для роли администратор. Данный вид таблицы позволяет администратору управлять маршрутами на станции, а также искать маршруты по городу.

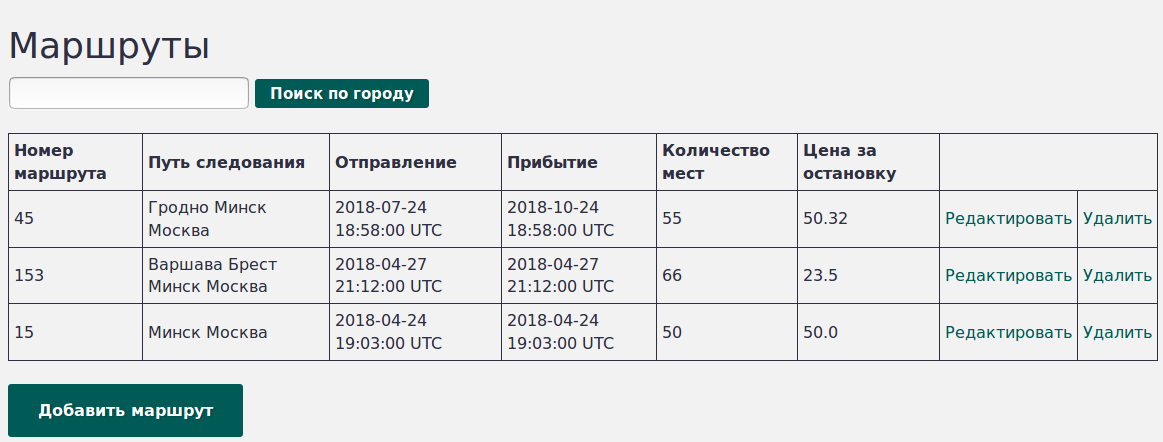


Рисунок 9.9 – Таблица маршрутов для администратора

Для покупки билета пассажиру необходимо перейти на страницу маршрута и заполнить форму покупки билета (рисунок 9.10).

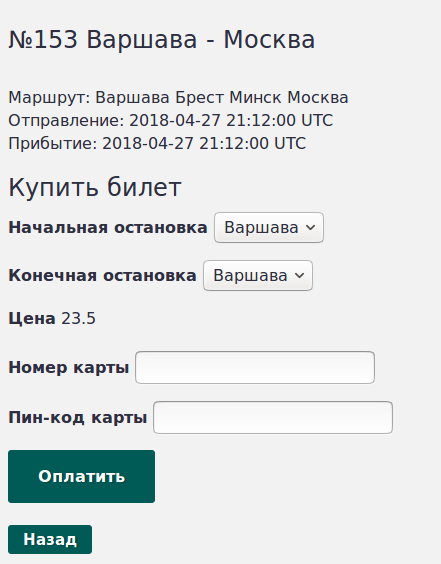


Рисунок 9.10 – Форма покупки билета

Также на странице маршрутов можно посмотреть график отправления маршрутов по дням недели (рисунок 9.11). Данный график позволит пассажирам спланировать поезду с учетом популярности марщрутов.

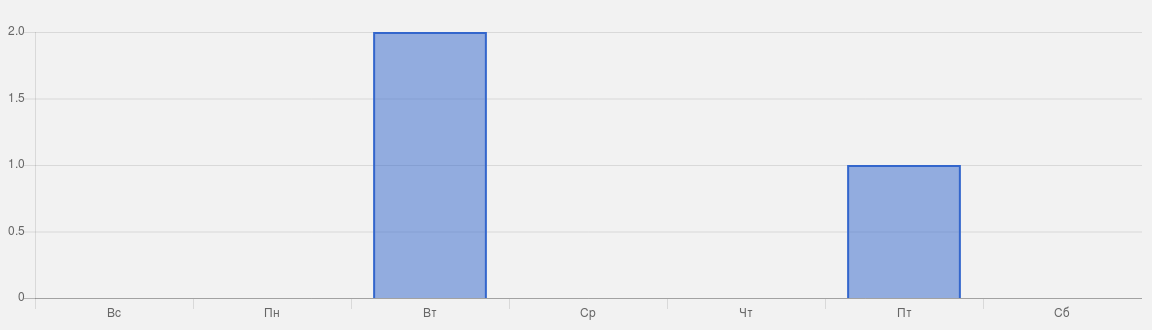


Рисунок 9.11 – График отправлений маршрутов по дням недели

Также сайт позволяет пользователям анализировать популярность направлений с помощью кругового графика, пример которого представлен на рисунке 9.12.

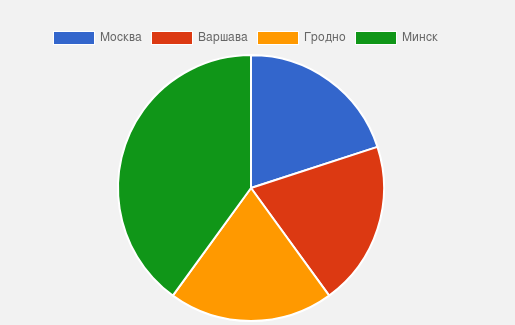


Рисунок 9.12 – График популярности направлений

Разработанное веб-приложение позволяет пассажирам выбрать следить за маршрутами, а также покупать билеты прямо на сайте. Администраторам станции данное приложение позволит сэкономить время на бумажной работе и уделять больше времени качеству обслуживания пассажиров по пути следования поездов.

# ЗАКЛЮЧЕНИЕ

В результате выполнения курсового проекта был реализован программный продукт для автоматизации билетно-кассовых операций на железнодорожной станции. Разработанное приложение избавить администраторов от большого объема бумажной работы и позволит больше времени уделять непосредственно качеству обслуживания пассажиров. Кроме того приложение позволяет анализировать популярность направлений на железнодорожной станции.

При разработке приложения были использованы следующие паттерны проектирования:

* модель-представление контроллер;
* заместитель;
* медиатор.

Использование шаблонов проектирования позволяет при необходимости легко расширить или изменить функционал приложения, в зависимости от потребностей пользователей. Кроме того паттерны позволяют упростить код приложения и сосредоточится на более качественном построении распределенного программного продукта.

Необходимая для работы приложения информация хранится в базе данных. Используемая СУБД – PostgreSQL. База данных генерируется sql-скриптом. Для запуска приложения необходимо только корректно настроить серверы каждого из компонентов.

В ходе работы над проектом было выполнено функциональное моделирование в соответствии со стандартом IDEF0, информационное моделирование (IDEF1X). Также для описания работы приложения были составлены следующие диаграммы с помощью UML.

Полученное приложение позволяет хранить и обрабатывать данные билетно-кассовых операций железнодорожной станции, работать под двумя типами пользователя (администратором и пассажиром), а также анализировать популярность направлений на железнодорожной станции.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Моргунов Е. П. Язык SQL. Базовый курс. Москва: Компания Postgres Professional, 2017.
2. Джей Макгаврен.Head First. Изучаем Ruby. O’Reilly, 2016.
3. Крис Пайн. Учись Программировать на Ruby. Санкт-Петербург, 2014.
4. Сэм Руби, Дэйв Томас, Дэвид Хэнссон. Rails 4. Гибкая разработка веб-приложений. Питер, 2014.
5. Г. Буч Объектно-ориентированный анализ и проектирование с примерами приложений. Вильями, 2010.
6. Дж. Рамбо, М. Блаха UML 2.0. Объектно-ориентированное моделирование и разработка. 2007
7. Хассан Гома UML-проектирование систем реального времени параллельных и распределенных приложений.
8. Установка Ruby on Rails на хостинг [Электронный ресурс]. Электронные данные. – Режим доступа: https:/www.digitalocean.com/community/tutorials/how-to-install-ruby-on-rails-with-rbenv-on-ubuntu-16-04

# ПРИЛОЖЕНИЕ

Скрипт генерации базы данных

/\*!40014 SET @OLD\_FOREIGN\_KEY\_CHECKS = @@FOREIGN\_KEY\_CHECKS, FOREIGN\_KEY\_CHECKS = 0 \*/;

SET NAMES 'utf8';

CREATE DATABASE railway\_tiurina\_k\_r;

USE railway\_tiurina\_k\_r;

--

-- Описание для таблицы users

--

CREATE TABLE users(

id INT (11) NOT NULL AUTO\_INCREMENT,

email VARCHAR (20) NOT NULL,

last\_name VARCHAR (20) DEFAULT NULL,

name VARCHAR (20) DEFAULT NULL,

second\_name VARCHAR (45) DEFAULT NULL,

phone VARCHAR (45) DEFAULT NULL,

password VARCHAR (45) DEFAULT NULL,

role VARCHAR (20) DEFAULT 'USER',

PRIMARY KEY (id),

UNIQUE INDEX id\_UNIQUE USING BTREE (id)

)

ENGINE = INNODB

AUTO\_INCREMENT = 3

CHARACTER SET utf8

COLLATE utf8\_general\_ci;

--

-- Описание для таблицы trains

--

CREATE TABLE trains(

id INT (11) NOT NULL AUTO\_INCREMENT,

train\_type VARCHAR (50) NOT NULL,

number\_of\_seats VARCHAR (50) NOT NULL,

price\_per\_stop FLOAT(11) NOT NULL,

PRIMARY KEY (id),

UNIQUE INDEX id\_UNIQUE USING BTREE (id)

)

ENGINE = INNODB

AUTO\_INCREMENT = 41

CHARACTER SET utf8

COLLATE utf8\_general\_ci;

--

-- Описание для таблицы ways

--

CREATE TABLE ways(

id INT (11) NOT NULL AUTO\_INCREMENT,

way\_number INT (11) NOT NULL,

stops\_array ARRAY (45) NOT NULL,

train\_id INT (11) NOT NULL,

departure DATETIME NOT NULL,

arrival DATETIME NOT NULL,

PRIMARY KEY (id),

UNIQUE INDEX id\_UNIQUE USING BTREE (id)

)

ENGINE = INNODB

AUTO\_INCREMENT = 14

CHARACTER SET utf8

COLLATE utf8\_general\_ci;

--

-- Описание для таблицы tickets

--

CREATE TABLE tickets(

id INT (11) NOT NULL AUTO\_INCREMENT,

way\_id INT (11) NOT NULL,

user\_id INT (11) NOT NULL,

end\_stop VARCHAR (45) NOT NULL,

price FLOAT (11) NOT NULL,

PRIMARY KEY (id),

UNIQUE INDEX id\_UNIQUE USING BTREE (id)

)

ENGINE = INNODB

AUTO\_INCREMENT = 14

CHARACTER SET utf8

COLLATE utf8\_general\_ci;

/\*!40014 SET FOREIGN\_KEY\_CHECKS = @OLD\_FOREIGN\_KEY\_CHECKS \*/;

Листинг кода

ticket.rb

class Ticket < ApplicationRecord

belongs\_to :way

belongs\_to :user

validates :way, presence: true

validates :user, presence: true

validates :user, uniqueness: { scope: :way\_id }

end

train.rb

class Train < ApplicationRecord

has\_many :ways, dependent: :destroy

validates :train\_type, presence: true

validates :number\_of\_seats, presence: true

validates :price\_per\_stop, presence: true

def name\_with\_initial

"#{train\_type}/#{number\_of\_seats}/#{price\_per\_stop}руб."

end

end

user.rb

class User < ApplicationRecord

# Include default devise modules. Others available are:

# :confirmable, :lockable, :timeoutable and :omniauthable

devise :database\_authenticatable, :registerable,

:recoverable, :rememberable, :validatable

has\_many :ways

validates :name, presence: true

validates :last\_name, presence: true

validates :second\_name, presence: true

validates :phone, presence: true

validates :email, presence: true

before\_validation :set\_role, on: :create

def full\_name

last\_name + " " + name + " " + second\_name

end

private

def set\_role

self.role = "\_USER\_" if self.role.blank?

end

end

way.rb

class Way < ApplicationRecord

belongs\_to :train

has\_many :tickets

has\_many :passengers, through: :tickets, source: :user

validates :way\_number, presence: true, uniqueness: true

validates :stops\_array, presence: true

validates :departure, presence: true

validates :arrival, presence: true

def name\_with\_initial

first\_stop = stops\_array.first

last\_stop = stops\_array.last

"№#{way\_number} #{first\_stop} - #{last\_stop}"

end

def stops\_array\_str

result = ""

stops\_array.each do |item|

result += item + " "

end

result

end

end

application\_controller.rb

class ApplicationController < ActionController::Base

protect\_from\_forgery with: :exception

before\_action :configure\_permitted\_parameters, if: :devise\_controller?

def configure\_permitted\_parameters

devise\_parameter\_sanitizer.permit(:account\_update) { |u|

u.permit(:role, :name, :last\_name, :second\_name, :phone, :password, :password\_confirmation, :current\_password)

}

devise\_parameter\_sanitizer.permit(:sign\_up) { |u|

u.permit(:role, :email, :name, :last\_name, :second\_name, :phone, :password, :password\_confirmation, :current\_password)

}

end

def index

end

end

tickets\_controller.rb

class TicketsController < ApplicationController

before\_action :authenticate\_user!

before\_action :set\_current\_user

before\_action :set\_way, only: [:create, :destroy]

before\_action :set\_ticket, only: [:destroy]

# GET /tickets/1

# GET /tickets/1.json

def show

end

def create

@new\_ticket = @way.tickets.build(ticket\_params)

@new\_ticket.user = @user

@new\_ticket.price = 100 # change to count method!!!

respond\_to do |format|

if @new\_ticket.save

format.html { redirect\_to way\_path(@way), notice: 'Билет успешно оплачен' }

format.json { render :show, status: :ok, location: @way }

else

format.html { render 'ways/show', alert: "Ошибка покупки билета" }

format.json { render json: @new\_ticket.errors, status: :unprocessable\_entity }

end

end

end

# DELETE /tickets/1

# DELETE /tickets/1.json

def destroy

@ticket.destroy

respond\_to do |format|

format.html { redirect\_to way\_url(@way), notice: 'Билет успешно аннулирован' }

format.json { head :no\_content }

end

end

private

# Use callbacks to share common setup or constraints between actions.

def set\_ticket

@ticket = @way.tickets.find(params[:id])

end

# Never trust parameters from the scary internet, only allow the white list through.

def ticket\_params

params.require(:ticket).permit(:price, :end\_stop)

end

def set\_way

@way = Way.find(params[:way\_id])

end

def set\_current\_user

@user = current\_user

end

end

trains\_controller.rb

class TrainsController < ApplicationController

before\_action :authenticate\_user!

before\_action :set\_current\_user

before\_action :set\_train, only: [:show, :edit, :update, :destroy]

# GET /trains

# GET /trains.json

def index

@trains = Train.all

if params[:train\_type\_filter].present?

@trains = Train.where(train\_type: params[:train\_type\_filter])

end

end

# GET /trains/1

# GET /trains/1.json

def show

end

def new

@train = Train.new

end

def create

@train = Train.new(train\_params)

respond\_to do |format|

if @train.save

format.html { redirect\_to trains\_path, notice: 'Поезд успешно создан.' }

format.json { render :index, status: :ok, location: @train }

else

format.html { render :new }

format.json { render json: @train.errors, status: :unprocessable\_entity }

end

end

end

# GET /trains/1/edit

def edit

end

# PATCH/PUT /trains/1

# PATCH/PUT /trains/1.json

def update

respond\_to do |format|

if @train.update(train\_params)

format.html { redirect\_to @train, notice: 'Данные поезда успешно обновлены.' }

format.json { render :show, status: :ok, location: @train }

else

format.html { render :edit }

format.json { render json: @train.errors, status: :unprocessable\_entity }

end

end

end

# DELETE /trains/1

# DELETE /trains/1.json

def destroy

@train.destroy

respond\_to do |format|

format.html { redirect\_to trains\_url, notice: 'Поезд успешно удален.' }

format.json { head :no\_content }

end

end

private

# Use callbacks to share common setup or constraints between actions.

def set\_train

@train = Train.find(params[:id])

end

# Never trust parameters from the scary internet, only allow the white list through.

def train\_params

params.require(:train).permit(:train\_type, :number\_of\_seats, :price\_per\_stop)

end

def set\_current\_user

@user = current\_user

end

end

users\_controller.rb

class UsersController < ApplicationController

before\_action :authenticate\_user!, except: [:index]

before\_action :set\_current\_user, except: [:index]

# GET /users

# GET /users.json

def index

@users = User.all

end

# GET /users/1

# GET /users/1.json

def show

end

# GET /users/1/edit

def edit

end

# PATCH/PUT /users/1

# PATCH/PUT /users/1.json

def update

respond\_to do |format|

if @user.update(user\_params)

format.html { redirect\_to @user, notice: 'Данные профиля успешно обновлены.' }

format.json { render :show, status: :ok, location: @user }

else

format.html { render :edit }

format.json { render json: @user.errors, status: :unprocessable\_entity }

end

end

end

# DELETE /users/1

# DELETE /users/1.json

def destroy

@user.destroy

respond\_to do |format|

format.html { redirect\_to users\_url, notice: 'Пользователь успешно удален.' }

format.json { head :no\_content }

end

end

private

# Use callbacks to share common setup or constraints between actions.

def set\_user

@user = User.find(params[:id])

end

# Never trust parameters from the scary internet, only allow the white list through.

def user\_params

params.require(:user).permit(:role, :name, :email, :last\_name, :second\_name, :phone, :password)

end

def set\_current\_user

@user = current\_user

end

end

ways\_controller.rb

class WaysController < ApplicationController

before\_action :authenticate\_user!

before\_action :set\_current\_user

before\_action :set\_way, only: [:show, :edit, :update, :destroy]

# GET /ways

# GET /ways.json

def index

@ways = Way.all

if params[:stop\_filter].present?

@ways = Way.where("'" + params[:stop\_filter] + "' = ANY (stops\_array)")

end

end

# GET /ways/1

# GET /ways/1.json

def show

@new\_ticket = @way.tickets.build(params[:ticket])

end

def new

@way = Way.new

@trains = Train.all

@cities = %w(Минск Гомель Гродно Могилев Пинск)

end

def create

@way = Way.new(way\_params)

@way.stops\_array = params[:stops\_array]

@way.stops\_array.pop

respond\_to do |format|

if @way.save

format.html { redirect\_to ways\_path, notice: 'Маршрут успешно создан.' }

format.json { render :index, status: :ok, location: @way }

else

format.html { render :new }

format.json { render json: @way.errors, status: :unprocessable\_entity }

end

end

end

# GET /ways/1/edit

def edit

end

# PATCH/PUT /ways/1

# PATCH/PUT /ways/1.json

def update

@way.stops\_array = params[:stops\_array]

@way.stops\_array.pop

respond\_to do |format|

if @way.update(way\_params)

format.html { redirect\_to ways\_path, notice: 'Данные маршрута успешно обновлены.' }

format.json { render :index, status: :ok, location: @way }

else

format.html { render :edit }

format.json { render json: @way.errors, status: :unprocessable\_entity }

end

end

end

# DELETE /ways/1

# DELETE /ways/1.json

def destroy

@way.destroy

respond\_to do |format|

format.html { redirect\_to ways\_url, notice: 'Маршрут успешно удален.' }

format.json { head :no\_content }

end

end

private

# Use callbacks to share common setup or constraints between actions.

def set\_way

@way = Way.find(params[:id])

end

# Never trust parameters from the scary internet, only allow the white list through.

def way\_params

params.require(:way).permit(:way\_number, :stops\_array, :train\_id, :departure, :arrival)

end

def set\_current\_user

@user = current\_user

end

end