МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

Информационная система «Туристическая фирма»

(наименование темы проекта или работы)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

Управление данными

(наименование дисциплины)

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Балашова Т.И.

(подпись) (фамилия, и., о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Усталкова В.С.

(подпись) (фамилия, и. О.)

17-ИТД

(шифр группы)

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород

2020

**Содержание**

Изм.

Лист

№ документа

Подпись

Дата

Лист

КР-ИРИТ-09.03.02-(17-ИТД)

Разработал

Усталкова В.С.

Проверил

Балашова ТИ

Реценз.

Н. Контр.

Утвердил

###### Система обслуживания клиентов туристической фирмы

Лит.

Листов

Кафедра «Информатика и

Системы Управления»

»

[Введение 4](#_Toc50570791)

[Анализ исходных данных и разработка ТЗ 5](#_Toc50570792)

[Основания и назначения разработки 5](#_Toc50570793)

[Назначение разработки 5](#_Toc50570794)

[Анализ технического задания 6](#_Toc50570795)

[Требования к структуре базы данных 6](#_Toc50570796)

[Выбор СУБД 7](#_Toc50570797)

[Основные функции СУБД 7](#_Toc50570798)

[Состав СУБД 7](#_Toc50570799)

[Классификации СУБД 8](#_Toc50570800)

[MongoDB 9](#_Toc50570801)

[PostgreSQL 11](#_Toc50570802)

[Выбор Back-End языка 13](#_Toc50570803)

[Проектирование базы данных 16](#_Toc50570804)

[Логическая модель 16](#_Toc50570805)

[Создание базы данных в среде PostgreSQL 17](#_Toc50570806)

[Front-End фреймворк VueJS 18](#_Toc50570807)

[Обзор среды разработки Visual Studio Code. 19](#_Toc50570808)

[Руководство пользователя 21](#_Toc50570809)

[Руководство программиста 25](#_Toc50570810)

[Минимальные требования к клиентскому ПК. 25](#_Toc50570811)

[Минимальные требования к серверу 25](#_Toc50570812)

[Заключение 26](#_Toc50570813)

[Список использованной литературы 27](#_Toc50570814)

Приложение

# Введение

Тема курсовой работы:” База знаний игры Герои меча и магии”.

Приложения создается для хранения информации о сущностях игры «Герои меча и магии». С помощью данного приложения можно познакомиться с расами, существами, героями, заклинаниями и их полными хакарктеристиками, а также добавлять, редактировать и удалять записи.

Пользователям должен быть представлен удобный интерфейс для визуализации хранимых данных и операций с ними.

Для достижения цели нужно решить следующие задачи:

1. Выбор СУБД
2. Проектирование структуры БД
3. Выбор фреймворка для Back-End и Front-End
4. Написание руководства пользователя
5. Написание руководства программиста

# Анализ исходных данных и разработка ТЗ

## Основания и назначения разработки

Разработать приложение “ База знаний игры Герои меча и магии ” , необходимого для работы с ифнормацией о сущностях игры: герои, существа, заклинания и т.д.

## Назначение разработки

Цель курсовой работы - закрепления знаний, полученных при изучении дисциплины, а также получение практических навыков проектирования информационных систем с использованием современных технологий.

Разрыбатываемое приложение позволит автоматизировать работу с данными игры.

**Требования к системе в целом**

1. Программа должна обеспечивать добавление данных об игре в базу данных.
2. Программа должна обеспечивать возможность просмотра, редактирования и изменения данных об игре.

**Требования к техническим средствам**

Требования к техническим средствам определяются установленной операционной системой. Программа не должна быть ресурсоёмкой и не предъявлять особых требований к техническим средствам.

**Требования к информационной и программной совместимости**

Программное обеспечение должно быть написано на языке JavaScript и выполнятся на сервере в среде NodeJS в ОС семейства Windows и Linux.

**Требования к пользовательскому интерфейсу**

Программа должна обладать удобным и интуитивным пользовательским интерфейсом с использованием форм, таблиц и переходам без перезагрузки страниц (Single Page Application).

## Анализ технического задания

Необходимо разработать информационную систему со следующим функционалом:

1. Добавление записей, возможность редактирования и удаления;
2. Проверка правильности ввода;
3. Обработка ошибок;
4. Наличие сложных запросов.

## Требования к структуре базы данных

Для функционирования базы данных были созданы следующие таблицы:

1. Расы;  
 2. Герои;  
 3. Существа;

4. Заклинания;

5. Армия;

6. Список\_Заклинаний.

**Расы:** имя, описание.

**Герои:** имя, описание, раса, очки здоровья и маны

**Существа:** имя, описание, раса, очки здоровья

**Заклинания:** название, описание, раса, очки маны

**Армия:** существо, герой, количество существ

**Список заклинаний:** заклинание, герой

# 

# Выбор СУБД

Система управления базами данных (СУБД)— ком­плекс про­грамм, по­зво­ляю­щих соз­дать ба­зу дан­ных (БД) и ма­ни­пу­ли­ро­вать дан­ны­ми (встав­лять, об­нов­лять, уда­лять и вы­би­рать). Система обес­пе­чи­ва­ет безо­пас­ность, на­дёж­ность хра­не­ния и це­ло­ст­ность дан­ных, а так­же пре­дос­тав­ля­ет сред­ст­ва для ад­ми­ни­ст­ри­ро­ва­ния БД

## Основные функции СУБД

* управление данными во внешней памяти (на дисках);
* управление данными в оперативной памяти с использованием дискового кэша;
* журнализация изменений, резервное копирование и восстановление базы данных после сбоев;
* поддержка языков БД (язык определения данных, язык манипулирования данными).

## Состав СУБД

* ядро, которое отвечает за управление данными во внешней и оперативной памяти и журнализацию;
* процессор языка базы данных, обеспечивающий оптимизацию запросов на извлечение и изменение данных и создание, как правило, машинно-независимого исполняемого внутреннего кода;
* подсистему поддержки времени исполнения, которая интерпретирует программы манипуляции данными, создающие пользовательский интерфейс с СУБД;
* сервисные программы (внешние утилиты), обеспечивающие ряд дополнительных возможностей по обслуживанию информационной системы.

## Классификации СУБД

**По модели данных**

* + Иерархические
  + Сетевые
  + Реляционные
  + Объектно-ориентированные

**По степени распределённости**

* Локальные СУБД (все части локальной СУБД размещаются на одном компьютере)
* Распределённые СУБД (части СУБД могут размещаться на двух и более компьютерах).

**По способу доступа к БД**

* Файл-серверные

В файл-серверных СУБД файлы данных располагаются централизованно на файл-сервере. СУБД располагается на каждом клиентском компьютере (рабочей станции). Доступ СУБД к данным осуществляется через локальную сеть. Синхронизация чтений и обновлений осуществляется посредством файловых блокировок. Преимуществом этой архитектуры является низкая нагрузка на ЦП сервера. Недостатки: потенциально высокая загрузка локальной сети; затруднённость централизованного управления; затруднённость обеспечения таких важных характеристик как высокая надёжность, высокая доступность и высокая безопасность. Применяются чаще всего в локальных приложениях, которые используют функции управления БД.

* Клиент-серверные

Клиент-серверная СУБД располагается на сервере вместе с БД и осуществляет доступ к БД непосредственно, в монопольном режиме. Все клиентские запросы на обработку данных обрабатываются клиент-серверной СУБД централизованно. Недостаток клиент-серверных СУБД состоит в повышенных требованиях к серверу. Достоинства: потенциально более низкая загрузка локальной сети; удобство централизованного управления; удобство обеспечения таких важных характеристик как высокая надёжность, высокая доступность и высокая безопасность.

* Встраиваемые

Встраиваемая СУБД — СУБД, которая может поставляться как составная часть некоторого программного продукта, не требуя процедуры самостоятельной установки. Встраиваемая СУБД предназначена для локального хранения данных своего приложения и не рассчитана на коллективное использование в сети. Физически встраиваемая СУБД чаще всего реализована в виде подключаемой библиотеки. Доступ к данным со стороны приложения может происходить через SQL либо через специальные программные интерфейсы.

## MongoDB

MongoDB реализует новый подход к построению баз данных, где нет таблиц, схем, запросов SQL, внешних ключей и многих других вещей, которые присущи объектно-реляционным базам данных.

В отличие от реляционных баз данных MongoDB предлагает документо-ориентированную модель данных, благодаря чему MongoDB работает быстрее, обладает лучшей масштабируемостью, ее легче использовать.

Но, даже учитывая все недостатки традиционных баз данных и достоинства MongoDB, важно понимать, что задачи бывают разные и методы их решения бывают разные. В какой-то ситуации MongoDB действительно улучшит производительность вашего приложения, например, если надо хранить сложные по структуре данные. В другой же ситуации лучше будет использовать традиционные реляционные базы данных. Кроме того, можно использовать смешенный подход: хранить один тип данных в MongoDB, а другой тип данных - в традиционных БД.

Вся система MongoDB может представлять не только одну базу данных, находящуюся на одном физическом сервере. Функциональность MongoDB позволяет расположить несколько баз данных на нескольких физических серверах, и эти базы данных смогут легко обмениваться данными и сохранять целостность.

*Формат данных в MongoDB*

Одним из популярных стандартов обмена данными и их хранения является JSON (JavaScript Object Notation). JSON эффективно описывает сложные по структуре данные. Способ хранения данных в MongoDB в этом плане похож на JSON, хотя формально JSON не используется. Для хранения в MongoDB применяется формат, который называется BSON (БиСон) или сокращение от binary JSON.

BSON позволяет работать с данными быстрее: быстрее выполняется поиск и обработка. Хотя надо отметить, что BSON в отличие от хранения данных в формате JSON имеет небольшой недостаток: в целом данные в JSON-формате занимают меньше места, чем в формате BSON, с другой стороны, данный недостаток с лихвой окупается скоростью.

*Кроссплатформенность*

MongoDB написана на C++, поэтому ее легко портировать на самые разные платформы. MongoDB может быть развернута на платформах Windows, Linux, MacOS, Solaris. Можно также загрузить исходный код и самому скомпилировать MongoDB, но рекомендуется использовать библиотеки с офсайта.

*Документы вместо строк*

Если реляционные базы данных хранят строки, то MongoDB хранит документы. В отличие от строк документы могут хранить сложную по структуре информацию. Документ можно представить, как хранилище ключей и значений.

## PostgreSQL

PostgreSQL является одной из наиболее популярных систем управления базами данных. Сам проект postgresql эволюционировал из другого проекта, который назывался Ingres. Формально развитие postgresql началось еще в 1986 году. Тогда он назывался POSTGRES. А в 1996 году проект был переименован в PostgreSQL, что отражало больший акцент на SQL. И собственно 8 июля 1996 года состоялся первый релиз продукта.

С тех пор вышло множество версий postgresql. Текущей версией является версия 12. Однако регулярно также выходят подверсии.

PostgreSQL поддерживается для всех основных операционных систем - Windows, Linux, MacOS.

Фундаментальная характеристика объектно-реляционной базы данных — это поддержка пользовательских объектов и их поведения, включая типы данных, функции, операции, домены и индексы. Это делает Postgres невероятно гибким и надежным. Среди прочего, он умеет создавать, хранить и извлекать сложные структуры данных.

*Преимущества PostgreSQL:*

* поддержка БД неограниченного размера;
* мощные и надёжные механизмы транзакций и репликации;
* расширяемая система встроенных языков программирования и поддержка загрузки C-совместимых модулей;
* наследование;
* легкая расширяемость.

*Текущие ограничения PostgreSQL:*

* Нет ограничений на максимальный размер базы данных
* Нет ограничений на количество записей в таблице
* Нет ограничений на количество индексов в таблице
* Максимальный размер таблицы — 32 Тбайт
* Максимальный размер записи — 1,6 Тбайт
* Максимальный размер поля — 1 Гбайт
* Максимум полей в записи 250—1600 (в зависимости от типов полей)

В качестве СУБД выбираем PostgreSQL, т.к. данные в приложении будут предствалены в реляционном виде, и будет удобнее работать с ними именно в Postgres.

# Выбор Back-End языка

Для разработки приложения был выбран язык JavaScript, т.к. у него есть много преимуществ перед другими языками программирования.

Изначально JavaScript был создан, чтобы «сделать веб-страницы живыми».

Программы на этом языке называются скриптами. Они могут встраиваться в HTML и выполняться автоматически при загрузке веб-страницы.

Скрипты распространяются и выполняются, как простой текст. Им не нужна специальная подготовка или компиляция для запуска.

Это отличает JavaScript от другого языка – Java.

Сегодня JavaScript может выполняться не только в браузере, но и на сервере или на любом другом устройстве, которое имеет специальную программу, называющуюся «движком» JavaScript.

Node.js — это опенсорсная кроссплатформенная среда выполнения для JavaScript, которая работает на серверах. С момента выпуска этой платформы в 2009 году она стала чрезвычайно популярной и в наши дни играет весьма важную роль в области веб-разработки.

*Скорость*

Одной из основных привлекательных особенностей Node.js является скорость. JavaScript-код, выполняемый в среде Node.js, может быть в два раза быстрее, чем код, написанный на компилируемых языках, вроде C или Java, и на порядки быстрее интерпретируемых языков наподобие Python или Ruby. Причиной подобного является неблокирующая архитектура платформы, а конкретные результаты зависят от используемых тестов производительности, но, в целом, Node.js — это очень быстрая платформа.

*Асинхронность*

В традиционных языках программирования (C, Java, Python, PHP) все инструкции, по умолчанию, являются блокирующими, если только разработчик явным образом не позаботится об асинхронном выполнении кода. В результате если, например, в такой среде, произвести сетевой запрос для загрузки некоего JSON-кода, выполнение потока, из которого сделан запрос, будет приостановлено до тех пор, пока не завершится получение и обработка ответа.

JavaScript значительно упрощает написание асинхронного и неблокирующего кода с использованием единственного потока, функций обратного вызова (коллбэков) и подхода к разработке, основанной на событиях. Каждый раз, когда нам нужно выполнить тяжёлую операцию, мы передаём соответствующему механизму коллбэк, который будет вызван сразу после завершения этой операции. В результате, для того чтобы программа продолжила работу, ждать результатов выполнения подобных операций не нужно.

Подобный механизм возник в браузерах. Мы не можем позволить себе ждать, скажем, окончания выполнения запроса, не имея при этом возможности реагировать на действия пользователя, например, на щелчки по кнопкам. Для того чтобы пользователям было удобно работать с веб-страницами, всё, и загрузка данных из сети, и обработка нажатия на кнопки, должно происходить одновременно, в режиме реального времени.

Асинхронные механизмы позволяют единственному Node.js-серверу одновременно обрабатывать тысячи подключений, не нагружая при этом программиста задачами по управлению потоками и по организации параллельного выполнения кода. Подобные вещи часто являются источниками ошибок.

Node.js предоставляет разработчику неблокирующие базовые механизмы ввода вывода, и, в целом, библиотеки, использующиеся в среде Node.js, написаны с использованием неблокирующих парадигм. Это делает блокирующее поведение кода скорее исключением, чем нормой.

Когда Node.js нужно выполнить операцию ввода-вывода, вроде загрузки данных из сети, доступа к базе данных или к файловой системе, вместо того, чтобы заблокировать ожиданием результатов такой операции главный поток, Node.js инициирует её выполнение и продолжает заниматься другими делами до тех пор, пока результаты выполнения этой операции не будут получены.

# Проектирование базы данных

## Логическая модель

Общим способом представления логической модели БД является построение ER-диаграмм (Entity-Relationship – сущность-связь). В этой модели сущность определяется как дискретный объект, для которого сохраняются элементы данных, а связь описывает отношение между двумя объектами.

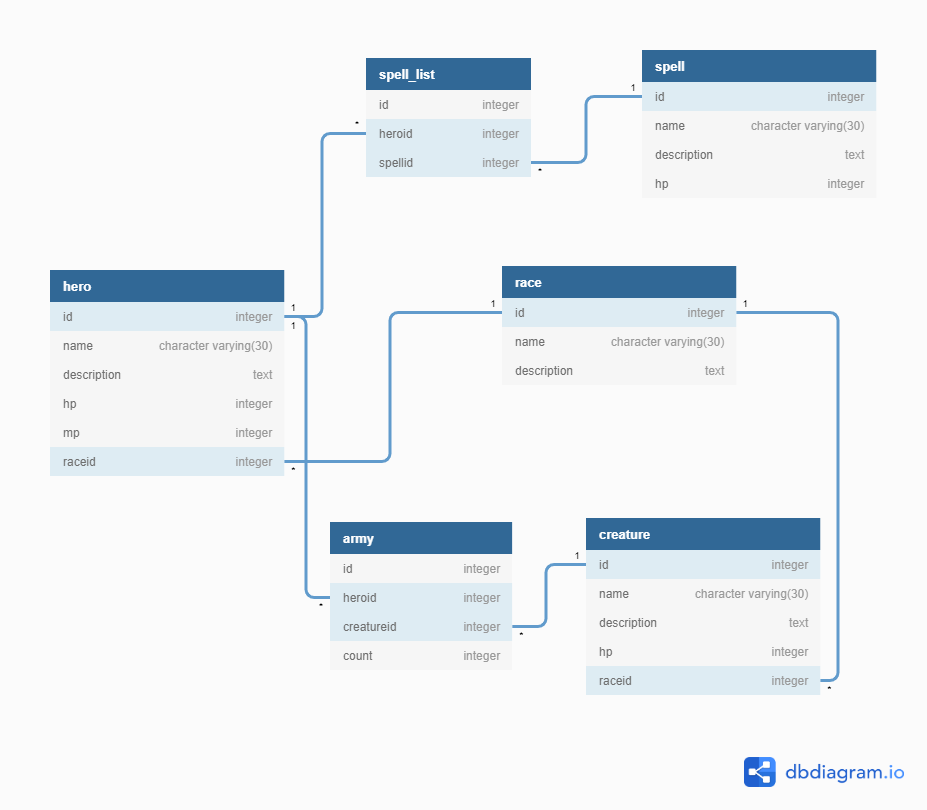


Рис.1 ER-диаграмма БД для приложения

## Создание базы данных в среде PostgreSQL

Для работы по созданию базы данных и таблиц будем использовать PostgreSQL.

Внешний вид окна программы PostgreSQL приведен на рис. 2.

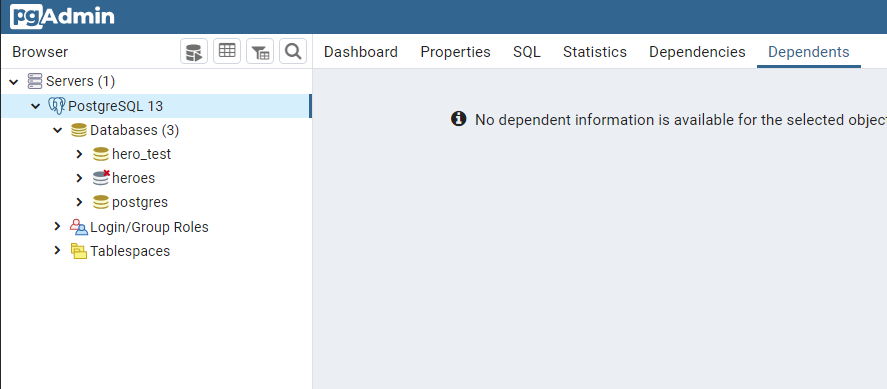


Рис. 2 Внешний вид окна PostgreSQL

Для создания базы данных необходимо кликнуть правой кнопкой мыши на пункте «Databases» и выбрать пункт меню «Create - Database». Окно создания БД представлено на рис. 3.

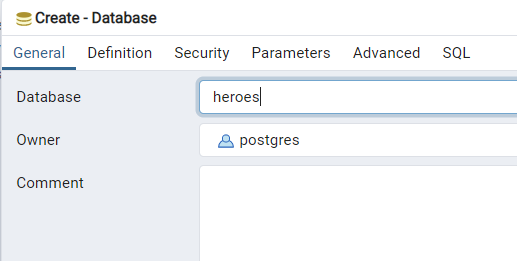


Рис.3 Окно создания БД

При разработке серверной части приложения мы будем использовать ORM, где опишем схемы данных, и впоследствии при первом запуске приложения таблицы и связи будут созданы автоматически (рис. 4).

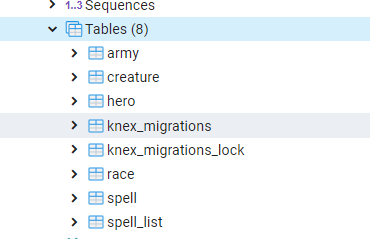


Рис.4 Сгенерированные таблицы.

# Front-End фреймворк VueJS

Для разработки клиентской части используется VueJS.

Vue.js или просто Vue – это прогрессивный JavaScript-фреймворк, который используется для создания пользовательских интерфейсов.

Фреймворк (framework) можно рассматривать как некий каркас для создания приложений на определенном языке программирования. Он включает в себя набор библиотек, которые позволяют значительно упростить и убыстрить разработку приложений. Также цель фреймворка – это предоставление такой среды разработки, которая позволит дорабатывать и расширять функционал проекта.

Vue реактивен, производителен и прост в освоении. Фреймворк не зависит от jQuery. Компоненты Vue легко использовать в уже существующем проекте.

# Обзор среды разработки Visual Studio Code.

Visual Studio Code — это сервис, который позиционируется как «легкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений.

Особенности Visual Studio Code:

* VS Code позволяет разрабатывать как консольные приложения, так и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms, а также веб-сайты, веб-приложения, веб-службы как в родном, так и в управляемом кодах для всех платформ.
* В редакторе присутствуют встроенный отладчик, инструменты для работы с Git и средства рефакторинга, навигации по коду, автодополнения типовых конструкций и контекстной подсказки.
* Продукт поддерживает разработку для платформ ASP.NET и Node.js, и считается легковесным решение, которое позволяет обойтись без полной интегрированной среды разработки.
* Большим плюсом редактора является поддержка большого количества языков, таких как C++, C#, Python, PHP, JavaScript и других.

Возможности Visual Studio Code:

* встроенные инструменты интеграции с GitHub, GIT, а также Visual Studio Team Services для быстрого тестирования, сборки, упаковки и развертывания разных типов приложений;
* удобство работы с Unity-проектами;
* работа с Mono и Node.js с помощью встроенного отладчика;
* поддержка TypeScript и JavaScript;
* публикация созданных приложений в Microsoft Azure через сервис Visual Studio Team Services;
* поддержка практически всех языков программирования;
* написание кода для конкретной задачи с его последующей интеграцией в проект (с надстройкой или напрямую);
* обширная библиотека шаблонов, готовых фрагментов кода и сниппетов с возможностью добавления своих элементов;
* одновременная работа с несколькими проектами (в нескольких окнах);
* интерфейс можно разделить на две панели для сравнения кода;
* функция отладки.

Структура нашего проекта в редакторе представлена на рис. 5.

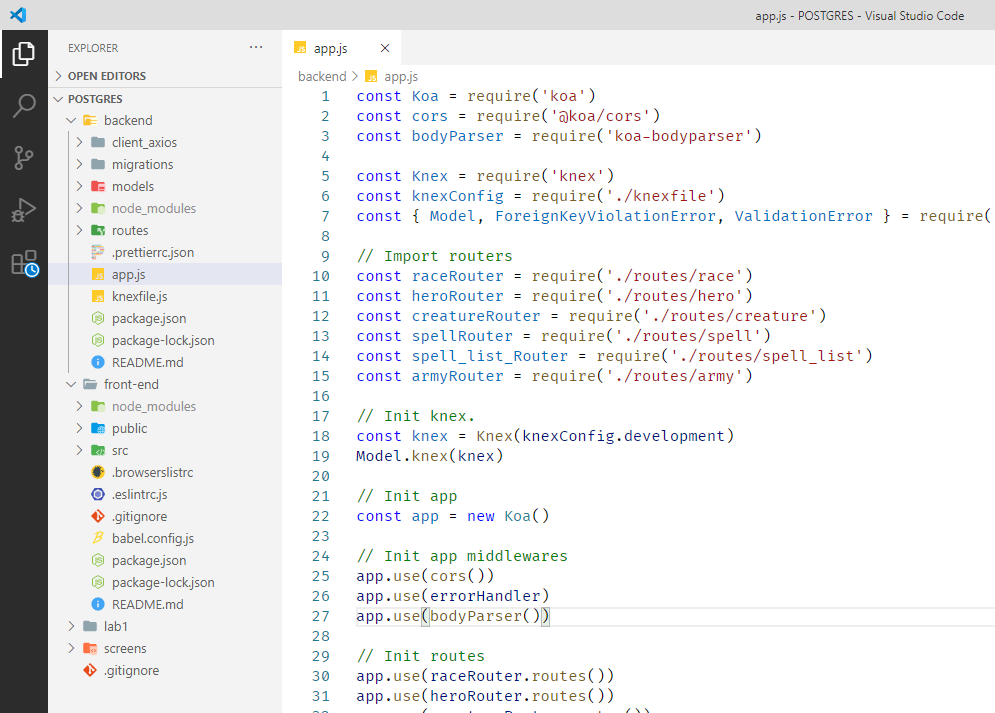


Рис.5 Структура проекта в VS Code.

# Руководство пользователя

При запуске программы пользователю будет представлена главная таблица приложения со списком героев. С помощью кнопок «Инфо», «Редактирование» и «Удаление» можно выыполнять соответствующие действия.



Рис.6. Таблица Hero

Формы добаления/редактирования героев.

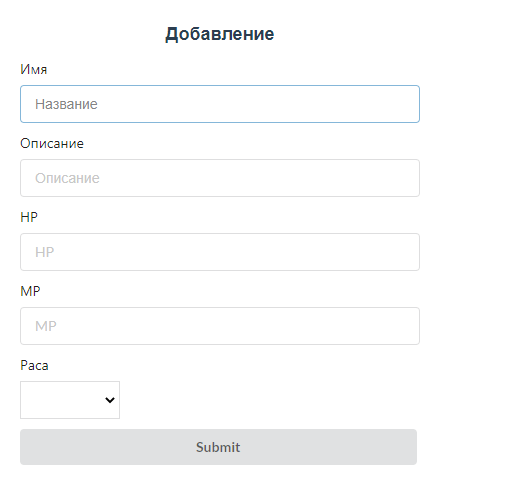
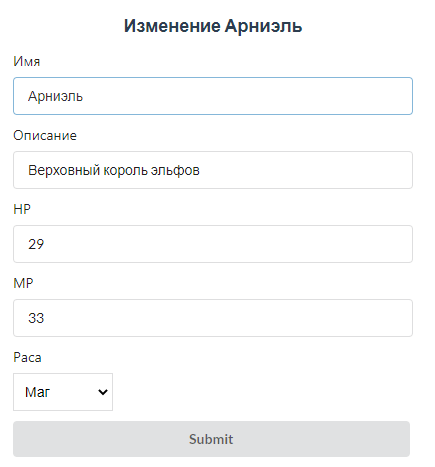
 

Рис.7 Добавления или изменение героя.

Таблица расы содержит список с названием и описанием возможных рас (с возможностью добавления, редактирования и удаления).

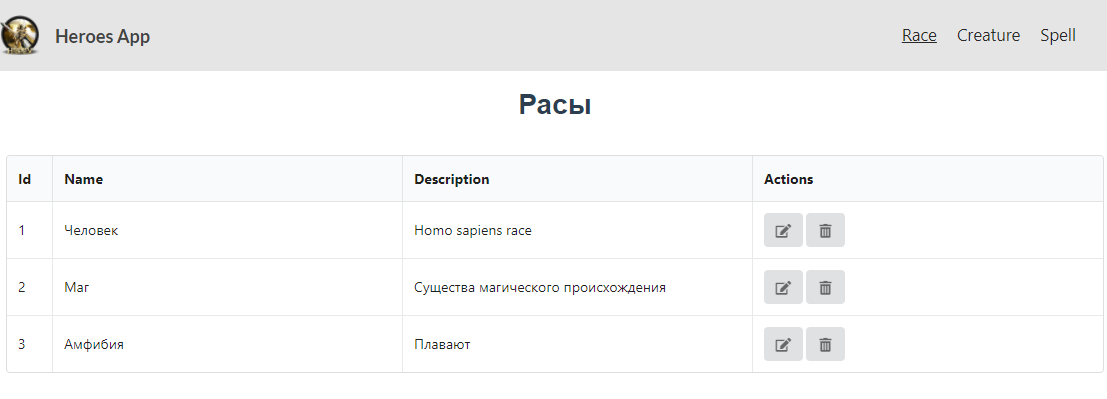


Рис.8 Таблица Race

Наподобии героям, существа также имеют отдельную таблицу, и каждое существо принадлжеит к какой-либо расе.

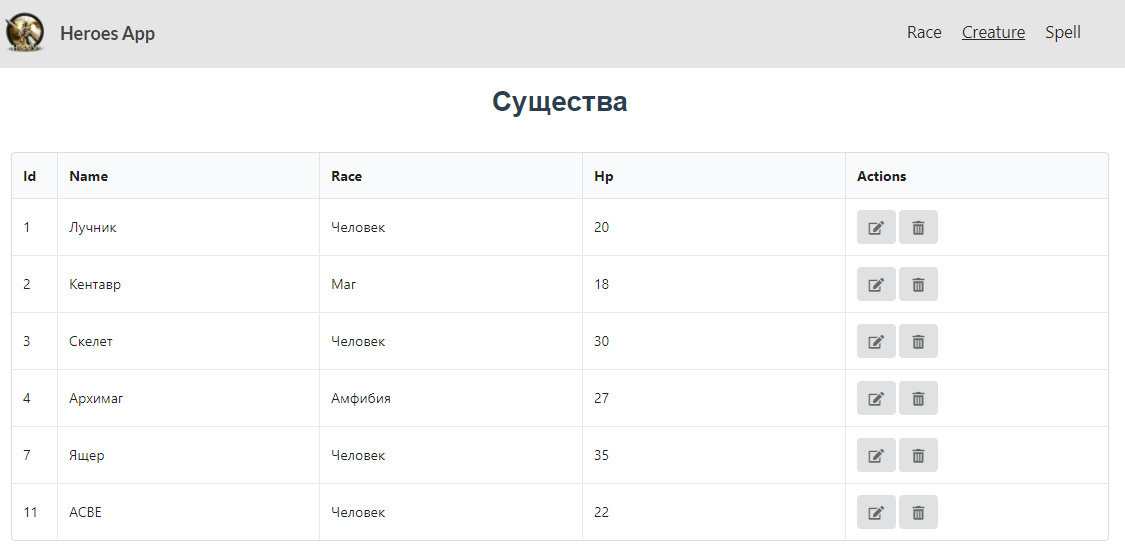


Рис.9 Таблица Creature

У каждого героя есть своя страница со всеми характеристиками, на которой представлены список заклинаний (для данного героя) и армия (набор существ с указанием количества). Здесь же можно внести изменения в список зиклинаний или армию (рис. 10).

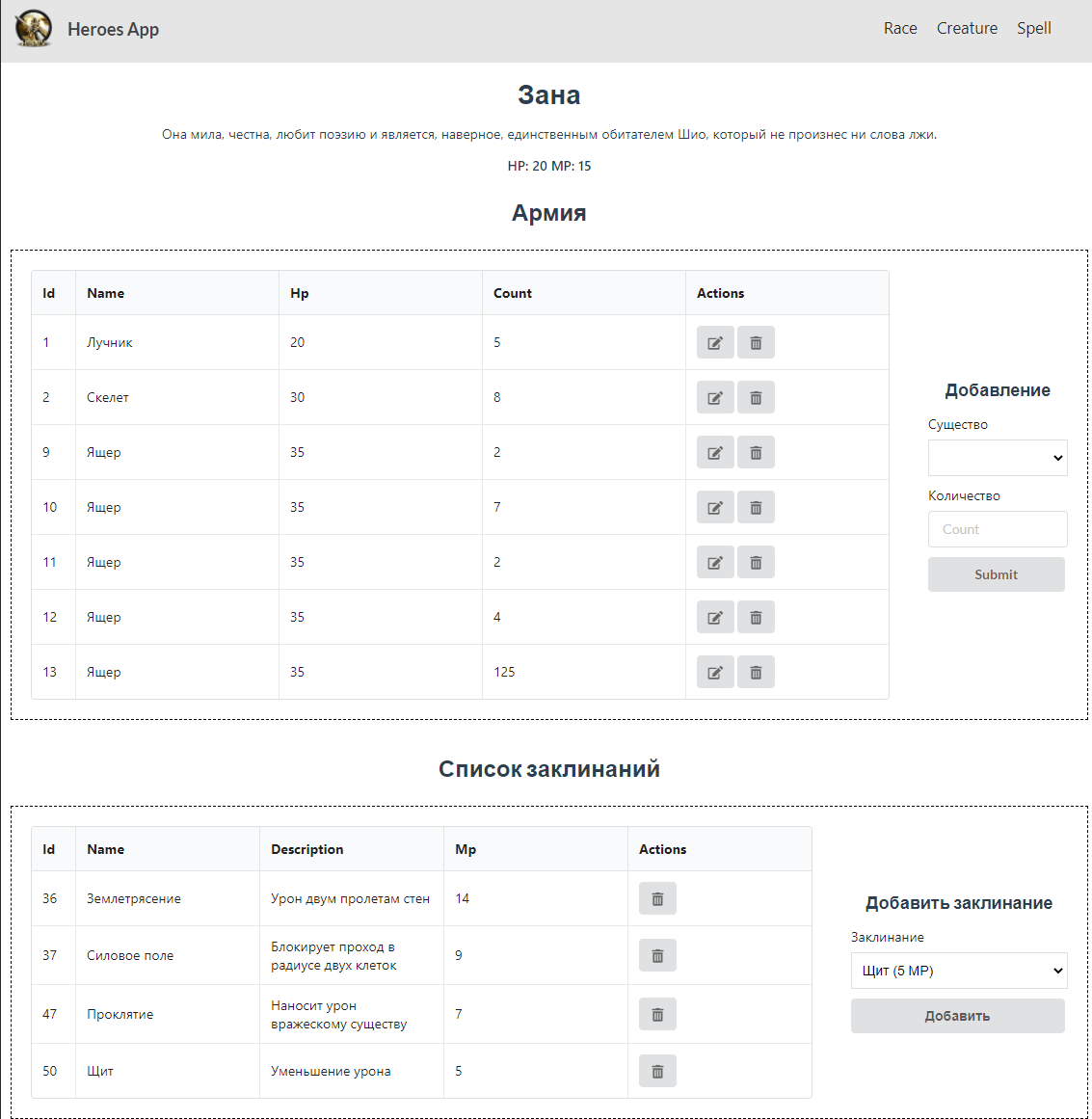


Рис.10. Страница героя с армией и списком заклинаний.

**Запросы**

Для работы с некоторыми сущностями используются сложные запросы. Например, чтобы получить армию героя, нужно выполнить запрос используя JOIN’ы (так как army – промежуточная таболица для связи многие ко многим между существами и героями).

Для упрощения такой задачи используется ORM ObjectionJS, в которой мы описываем модели данных и их взамосвязи. Затем при выполнении запросом мы указываем, какие данные нужно «присоединить» и получаем готовый набор данных.

Например, так выглядит RelationMapping сущности Hero:

{

      race: {

        relation: Model.BelongsToOneRelation,

        join: {

          from: 'hero.raceId',

          to: 'race.id',

        },

      },

      army: {

        relation: Model.ManyToManyRelation,

        modelClass: Creature,

        join: {

          from: 'hero.id',

          through: {

            from: 'army.heroId',

            to: 'army.creatureId',

          },

          to: 'creature.id',

        },

      },

      spell: {

        relation: Model.ManyToManyRelation,

        modelClass: Spell,

        join: {

          from: 'hero.id',

          through: {

            from: 'spell\_list.heroId',

            to: 'spell\_list.spellId',

          },

          to: 'spell.id',

        },

      },

    }

# Руководство программиста

## Минимальные требования к клиентскому ПК.

* компьютер с процессором Intel Pentium и выше или совместимый
* 1 ГБ RAM
* Монитор
* Клавиатура
* Мышь
* операционная система MS Windows® 7 и выше
* Google Chrome 55+ или Firefox 24+

## Минимальные требования к серверу

* операционная система Linux Ubuntu 18.0.4
* 512 МБ RAM

Программное обеспечение должно быть написано на прототипно-ориентированном языке JavaScript.

Программное обеспечение должно функционировать на операционной системе Windows версии 7 или новее.

Программа должна функционировать независимо от наличия в системе установленных средств разработки.

# Заключение

Все задачи, поставленные в начале работы, достигнуты. Разработанная информационная система удовлетворяет всем установленным ранее требованиям.

В результате мы получили автоматизированную систему для чтения, изменения, добавления и удаления данных об игре «Герои меча и магии».

# Список использованной литературы

1. Эрик Фримен, Элизабет Робсон – Head First. Паттерны проектирования.
2. Руководство PostgreSQL - https://metanit.com/sql/postgresql/
3. Документация NodeJS - <https://nodejs.org/en/docs/>
4. Документация VueJS - <https://vuejs.org/v2/guide/>
5. Документация ObjectionJS <https://vincit.github.io/objection.js/guide/>

**ПРИЛОЖЕНИЕ**

**initMigration.js**

exports.up = function (knex) {

  return knex.schema

    .createTable('race', (table) => {

      table.increments('id').primary()

      table.string('name')

      table.string('description')

    })

    .createTable('hero', (table) => {

      table.increments('id').primary()

      table

        .integer('raceId')

        .unsigned()

        .references('id')

        .inTable('race')

        .onDelete('SET NULL')

        .index()

      table.string('name')

      table.string('description')

      table.integer('hp')

      table.integer('mp')

    })

    .createTable('creature', (table) => {

      table.increments('id').primary()

      table

        .integer('raceId')

        .unsigned()

        .references('id')

        .inTable('race')

        .onDelete('SET NULL')

        .index()

      table.string('name')

      table.string('description')

      table.integer('hp')

    })

    .createTable('army', (table) => {

      table.increments('id').primary()

      table

        .integer('heroId')

        .unsigned()

        .references('id')

        .inTable('hero')

        .onDelete('CASCADE')

        .index()

      table

        .integer('creatureId')

        .unsigned()

        .references('id')

        .inTable('creature')

        .onDelete('CASCADE')

        .index()

      table.integer('count')

    })

    .createTable('spell', (table) => {

      table.increments('id').primary()

      table.string('name')

      table.string('description')

      table.integer('mp')

    })

    .createTable('spell\_list', (table) => {

      table.increments('id').primary()

      table

        .integer('heroId')

        .unsigned()

        .references('id')

        .inTable('hero')

        .onDelete('CASCADE')

        .index()

      table

        .integer('spellId')

        .unsigned()

        .references('id')

        .inTable('spell')

        .onDelete('CASCADE')

        .index()

    })

}

exports.down = function (knex) {

  return knex.schema

    .dropTableIfExists('hero')

    .dropTableIfExists('race')

    .dropTableIfExists('spell')

    .dropTableIfExists('army')

    .dropTableIfExists('spell\_list')

    .dropTableIfExists('creature')

}

**Army.js**

'use strict'

const { Model } = require('objection')

class Army extends Model {

  // Table name is the only required property.

  static get tableName() {

    return 'army'

  }

  static get jsonSchema() {

    return {

      type: 'object',

      required: ['heroId', 'creatureId', 'count'],

      properties: {

        id: { type: 'integer' },

        heroId: { type: ['integer', 'null'] },

        creatureId: { type: ['integer', 'null'] },

        count: { type: 'integer' },

      },

    }

  }

  // This object defines the relations to other models.

  static get relationMappings() {

    // One way to prevent circular references

    // is to require the model classes here.

    const Hero = require('./Hero')

    const Creature = require('./Creature')

    return {

      hero: {

        relation: Model.HasManyRelation,

        modelClass: Hero,

        join: {

          from: 'army.heroId',

          to: 'hero.id',

        },

      },

      creature: {

        relation: Model.HasManyRelation,

        modelClass: Creature,

        join: {

          from: 'army.creatureId',

          to: 'creature.id',

        },

      },

    }

  }

}

module.exports = Army

**Creature.js**

'use strict'

const { Model } = require('objection')

class Creature extends Model {

  // Table name is the only required property.

  static get tableName() {

    return 'creature'

  }

  static get jsonSchema() {

    return {

      type: 'object',

      required: ['name'],

      properties: {

        id: { type: 'integer' },

        name: { type: 'string', minLength: 1, maxLength: 255 },

        description: { type: 'string', maxLength: 255 },

        hp: { type: 'integer' },

        raceId: { type: ['integer', 'null'] },

      },

    }

  }

  // This object defines the relations to other models.

  static get relationMappings() {

    // One way to prevent circular references

    // is to require the model classes here.

    const Race = require('./Race')

    return {

      race: {

        relation: Model.BelongsToOneRelation,

        // The related model. This can be either a Model subclass constructor or an

        // absolute file path to a module that exports one.

        modelClass: Race,

        join: {

          from: 'creature.raceId',

          to: 'race.id',

        },

      },

    }

  }

}

module.exports = Creature

**Hero.js**

'use strict'

const { Model } = require('objection')

class Hero extends Model {

  // Table name is the only required property.

  static get tableName() {

    return 'hero'

  }

  static get jsonSchema() {

    return {

      type: 'object',

      required: ['name'],

      properties: {

        id: { type: 'integer' },

        name: { type: 'string', minLength: 1, maxLength: 255 },

        description: { type: 'string', maxLength: 255 },

        hp: { type: 'integer' },

        mp: { type: 'integer' },

        raceId: { type: ['integer', 'null'] },

      },

    }

  }

  // This object defines the relations to other models.

  static get relationMappings() {

    // One way to prevent circular references

    // is to require the model classes here.

    const Race = require('./Race')

    const Army = require('./Army')

    const Creature = require('./Creature')

    const Spell = require('./Spell')

    return {

      race: {

        relation: Model.BelongsToOneRelation,

        join: {

          from: 'hero.raceId',

          to: 'race.id',

        },

      },

      army: {

        relation: Model.ManyToManyRelation,

        modelClass: Creature,

        join: {

          from: 'hero.id',

          through: {

            from: 'army.heroId',

            to: 'army.creatureId',

          },

          to: 'creature.id',

        },

      },

      spell: {

        relation: Model.ManyToManyRelation,

        modelClass: Spell,

        join: {

          from: 'hero.id',

          through: {

            from: 'spell\_list.heroId',

            to: 'spell\_list.spellId',

          },

          to: 'spell.id',

        },

      },

    }

  }

}

module.exports = Hero

**Race.js**

'use strict'

const { Model } = require('objection')

class Race extends Model {

  // Table name is the only required property.

  static get tableName() {

    return 'race'

  }

  static get jsonSchema() {

    return {

      type: 'object',

      required: ['name'],

      properties: {

        id: { type: 'integer' },

        name: { type: 'string', minLength: 1, maxLength: 255 },

        description: { type: ['string', 'null'], maxLength: 255 },

      },

    }

  }

}

module.exports = Race

**Spell.js**

'use strict'

const { Model } = require('objection')

class Spell extends Model {

  // Table name is the only required property.

  static get tableName() {

    return 'spell'

  }

  static get jsonSchema() {

    return {

      type: 'object',

      required: ['name'],

      properties: {

        id: { type: 'integer' },

        name: { type: 'string', minLength: 1, maxLength: 255 },

        description: { type: ['string', 'null'], maxLength: 255 },

        hp: { type: 'integer' },

      },

    }

  }

}

module.exports = Spell

**SpellList.js**

'use strict'

const { Model } = require('objection')

class SpellList extends Model {

  // Table name is the only required property.

  static get tableName() {

    return 'spell\_list'

  }

  static get jsonSchema() {

    return {

      type: 'object',

      required: ['heroId', 'spellId'],

      properties: {

        id: { type: 'integer' },

        heroId: { type: ['integer', 'null'] },

        spellId: { type: ['integer', 'null'] },

      },

    }

  }

  // This object defines the relations to other models.

  static get relationMappings() {

    // One way to prevent circular references

    // is to require the model classes here.

    const Hero = require('./Hero')

    const Spell = require('./Spell')

    return {

      hero: {

        relation: Model.HasManyRelation,

        modelClass: Hero,

        join: {

          from: 'spell\_list.heroId',

          to: 'hero.id',

        },

      },

      spell: {

        relation: Model.HasManyRelation,

        modelClass: Spell,

        join: {

          from: 'spell\_list.spellId',

          to: 'spell.id',

        },

      },

    }

  }

}

module.exports = SpellList

**App.js**

const Koa = require('koa')

const cors = require('@koa/cors')

const bodyParser = require('koa-bodyparser')

const Knex = require('knex')

const knexConfig = require('./knexfile')

const { Model, ForeignKeyViolationError, ValidationError } = require('objection')

// Import routers

const raceRouter = require('./routes/race')

const heroRouter = require('./routes/hero')

const creatureRouter = require('./routes/creature')

const spellRouter = require('./routes/spell')

const spell\_list\_Router = require('./routes/spell\_list')

const armyRouter = require('./routes/army')

// Init knex.

const knex = Knex(knexConfig.development)

Model.knex(knex)

// Init app

const app = new Koa()

// Init app middlewares

app.use(cors())

app.use(errorHandler)

app.use(bodyParser())

// Init routes

app.use(raceRouter.routes())

app.use(heroRouter.routes())

app.use(creatureRouter.routes())

app.use(spellRouter.routes())

app.use(armyRouter.routes())

app.use(spell\_list\_Router.routes())

// Start app server

const server = app.listen(8641, () => {

  console.log('Example app listening at port %s', server.address().port)

})

// Error handling

async function errorHandler(ctx, next) {

  try {

    await next()

  } catch (err) {

    if (err instanceof ValidationError) {

      ctx.status = 400

      ctx.body = {

        error: 'ValidationError',

        errors: err.data,

      }

    } else if (err instanceof ForeignKeyViolationError) {

      ctx.status = 409

      ctx.body = {

        error: 'ForeignKeyViolationError',

      }

    } else {

      ctx.status = 500

      ctx.body = {

        error: 'InternalServerError',

        message: err.message || {},

      }

    }

  }

}

**Hero.route.js**

const Router = require('koa-router')

const router = new Router({ prefix: '/hero' })

const Hero = require('../models/Hero')

router.get('/', async (ctx) => {

  const hero = await Hero.query().orderBy('id')

  ctx.body = hero

})

router.get('/:id', async (ctx) => {

  const hero = await Hero.query().findById(ctx.params.id)

  if (!hero) ctx.status = 404

  ctx.body = hero

})

router.get('/:id/army', async (ctx) => {

  const hero = await Hero.relatedQuery('army')

    .select(

      'army.id',

      'creature.id as creatureId',

      'creature.name',

      'creature.raceId',

      'creature.hp',

      'army.count'

    )

    .for(ctx.params.id)

    .orderBy('id')

  ctx.body = hero

})

router.get('/:id/spells', async (ctx) => {

  const hero = await Hero.relatedQuery('spell')

    .select('spell\_list.id', 'spell.name', 'spell.description', 'spell.mp')

    .for(ctx.params.id)

    .orderBy('id')

  ctx.body = hero

})

router.post('/', async (ctx) => {

  const hero = await Hero.query().insert(ctx.request.body)

  ctx.body = hero

})

router.patch('/:id', async (ctx) => {

  const numUpdated = await Hero.query().findById(ctx.params.id).patch(ctx.request.body)

  ctx.body = {

    success: numUpdated == 1,

  }

})

router.delete('/:id', async (ctx) => {

  const numDeleted = await Hero.query().findById(ctx.params.id).delete()

  ctx.body = {

    success: numDeleted == 1,

  }

})

module.exports = router

**Creature.route.js**

const Router = require('koa-router')

const router = new Router({ prefix: '/creature' })

const Creature = require('../models/Creature')

router.get('/', async (ctx) => {

  const creature = await Creature.query()

    .select(

      'creature.id',

      'creature.name',

      'creature.description',

      'creature.hp',

      'creature.raceId',

      'race.name as race\_name'

    )

    .fullOuterJoin('race', 'race.id', 'creature.raceId')

    .from('creature')

    .orderBy('id')

  ctx.body = creature

})

router.get('/:id', async (ctx) => {

  const creature = await Creature.query().findById(ctx.params.id)

  ctx.body = creature

})

router.post('/', async (ctx) => {

  const creature = await Creature.query().insert(ctx.request.body)

  ctx.body = creature

})

router.patch('/:id', async (ctx) => {

  const numUpdated = await Creature.query().findById(ctx.params.id).patch(ctx.request.body)

  ctx.body = {

    success: numUpdated == 1,

  }

})

router.delete('/:id', async (ctx) => {

  const numDeleted = await Creature.query().findById(ctx.params.id).delete()

  ctx.body = {

    success: numDeleted == 1,

  }

})

module.exports = router

**Spell.route.js**

const Router = require('koa-router')

const router = new Router({ prefix: '/spell' })

const Spell = require('../models/Spell')

router.get('/', async (ctx) => {

  const spell = await Spell.query().orderBy('id')

  ctx.body = spell

})

router.post('/', async (ctx) => {

  const spell = await Spell.query().insert(ctx.request.body)

  ctx.body = spell

})

router.patch('/:id', async (ctx) => {

  const numUpdated = await Spell.query().findById(ctx.params.id).patch(ctx.request.body)

  ctx.body = {

    success: numUpdated == 1,

  }

})

router.delete('/:id', async (ctx) => {

  const numDeleted = await Spell.query().findById(ctx.params.id).delete()

  ctx.body = {

    success: numDeleted == 1,

  }

})

module.exports = router

**Race.route.js**

const Router = require('koa-router')

const router = new Router({ prefix: '/race' })

const Race = require('../models/Race')

router.get('/', async (ctx) => {

  const races = await Race.query().orderBy('id')

  ctx.body = races

})

router.get('/:id', async (ctx) => {

  const race = await Race.query().findById(ctx.params.id)

  ctx.body = race

})

router.post('/', async (ctx) => {

  const race = await Race.query().insert(ctx.request.body)

  ctx.body = race

})

router.patch('/:id', async (ctx) => {

  const numUpdated = await Race.query().findById(ctx.params.id).patch(ctx.request.body)

  ctx.body = {

    success: numUpdated == 1,

  }

})

router.delete('/:id', async (ctx) => {

  const numDeleted = await Race.query().findById(ctx.params.id).delete()

  ctx.body = {

    success: numDeleted == 1,

  }

})

module.exports = router

**App.vue**

<template>

  <div id="app">

    <header>

      <div v-on:click="gohome" class="logo">

        <img src="./assets/homm\_logo.png" />

        <h3>Heroes App</h3>

      </div>

      <nav id="nav">

        <!-- <router-link to="/">Hero</router-link> -->

        <router-link to="/race">Race</router-link>

        <router-link to="/creature">Creature</router-link>

        <router-link to="/spell">Spell</router-link>

      </nav>

    </header>

    <router-view class="router-container" />

  </div>

</template>

<script>

export default {

  methods: {

    gohome: function () {

      this.$router.push("/");

    },

  },

};

</script>

<style lang="scss">

body {

  margin: 0;

  padding: 0;

}

#app {

  display: flex;

  flex-flow: column wrap;

  font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Oxygen,

    Ubuntu, Cantarell, "Open Sans", "Helvetica Neue", sans-serif;

  text-align: center;

  color: #2c3e50;

  margin-bottom: 10px;

}

header {

  display: flex;

  flex-flow: row nowrap;

  justify-content: space-between;

  align-items: center;

  background-color: #e5e5e5;

  .logo {

    display: flex;

    flex-flow: row nowrap;

    justify-content: center;

    align-items: center;

    margin-left: 1rem;

    color: #434343;

    cursor: pointer;

    &:hover {

      color: rgb(116, 116, 116);

    }

    h3 {

      margin-bottom: 1.6rem;

    }

    img {

      width: 40px;

      height: 40px;

      margin-right: 15px;

    }

  }

  nav {

    margin: 10px;

    margin-right: 2rem;

    a {

      text-decoration: none;

      font-size: 1.2rem;

      margin-right: 20px;

      color: rgb(41, 41, 41);

      &.router-link-active {

        text-decoration: underline;

      }

      &:hover {

        color: rgb(116, 116, 116);

      }

    }

  }

}

.router-container {

  margin-top: 1rem;

}

</style>

**Hero.vue**

<template>

  <main class="view">

    <h1>Герои</h1>

    <vuetable

      ref="vuetable"

      :fields="['id', 'name', 'description', 'mp', 'hp', 'actions']"

      :api-mode="false"

      :data="data"

      :css="css.table"

      class="table-wrapper"

    >

      <div slot="actions" slot-scope="props">

        <div class="actions-btn">

          <button class="ui small blue button" @click="$router.push(`/hero/${props.rowData.id}`)">

            Info

          </button>

          <button class="ui icon button" @click="switchActionToEdit('view-item', props.rowData)">

            <i class="edit icon"></i>

          </button>

          <button class="ui icon red button" @click="onActionDelete('view-item', props.rowData)">

            <i class="trash alternate icon"></i>

          </button>

        </div>

      </div>

    </vuetable>

    <div class="action-form">

      <h3>{{action.title}} {{action.item && action.item.name}}</h3>

      <EditHeroForm

        :item\_name="action.item ? action.item.name : ''"

        :item\_description="action.item ? action.item.description : ''"

        :item\_hp="action.item && action.item.hp"

        :item\_mp="action.item && action.item.mp"

        :item\_race="action.item && action.item.raceId"

        @submit="handleSubmit"

      />

    </div>

  </main>

</template>

<script>

import Vuetable from "vuetable-2";

import EditHeroForm from "../components/forms/EditHero";

import cssTable from "../utils/css-table";

import http from "../utils/http";

import { CREATE\_ACTION, EDIT\_ACTION } from "../utils/actions";

export default {

  components: {

    Vuetable,

    EditHeroForm,

  },

  data() {

    return {

      css: cssTable,

      data: [],

      action: {

        title: CREATE\_ACTION,

        currentId: null,

        item: null,

      },

    };

  },

  mounted() {

    this.updateTable();

  },

  methods: {

    onActionDelete(action, data) {

      http.delete(`/hero/${data.id}`).then(() => this.updateTable());

    },

    handleSubmit(data) {

      if (this.action.title === CREATE\_ACTION) {

        this.addItem(data);

      } else {

        this.editItem(data);

      }

    },

    switchActionToEdit(event, data) {

      if (this.action.currentId === data.id) {

        this.switchActionToCreate();

        return;

      }

      this.action.title = EDIT\_ACTION;

      this.action.currentId = data.id;

      this.action.item = data;

    },

    switchActionToCreate() {

      this.action.title = CREATE\_ACTION;

      this.action.currentId = null;

      this.action.item = null;

    },

    addItem(data) {

      http.post("/hero", data).then(() => this.updateTable());

    },

    editItem(data) {

      http

        .patch(`/hero/${this.action.currentId}`, data)

        .then(() => this.updateTable());

    },

    updateTable() {

      http

        .get(`/hero`)

        .then((res) => {

          this.data = res.data;

        })

        .catch((err) => console.log(err));

    },

  },

};

</script>

<style lang="scss">

.view {

  display: flex;

  flex-flow: column wrap;

  justify-content: center;

  align-items: center;

}

.table-wrapper {

  display: flex;

  flex-flow: column wrap;

  align-items: center;

  margin: 10px;

  padding: 10px;

  max-width: 1200px;

  .ui.celled.table tr td:first-child,

  .ui.celled.table tr th:first-child {

    width: 45px;

  }

}

.actions-btn {

  display: flex;

  flex-flow: row nowrap;

  .button {

    display: flex;

    flex-flow: row nowrap;

    justify-content: center;

    align-items: center;

  }

}

.action-form {

  margin: 20px;

  max-width: 400px;

  width: 400px;

}

</style>