ВВЕДЕНИЕ

В зависимости от наших потребностей финансы играют определенную роль в нашей жизни. За большую многовековую историю они стали хорошим инструментом, при помощи которого человек направляет и усиливает свою жизненную энергию, реализует свои планы и идеи. Человек правит миром при помощи финансов (денег) в своих, присущих ему, индивидуальных рамках. И чтобы это происходило, нужно соблюдать ряд правил, только тогда человек сможет жить в гармонии с собой и окружающим его миром.

Чтобы добиться этой гармонии существует ряд подходов, один из них – ведение бюджета. Бюджет – это своеобразные план использования и формирования денежных средств для реализации необходимых функций и задач.

Ведение бюджета помогает решить ряд проблем:

* навести порядок в голове, относительно того, сколько человек зарабатывает и тратит, в каком направлении расходуется его бюджет и в каком направлении он должен идти;
* закрыть множество дыр в бюджете, через которые финансы уходят в неизвестном направлении;
* узнать насколько для человека важно то, на что он сегодня потратил свои деньги;
* стать умнее, эффективнее и успешнее;
* реализовать мечты, и сделать это как можно раньше;
* упорядочить свою жизнь.

В настоящее время вести бюджет – такая же необходимость, как чистить зубы или вести здоровый образ жизни, или развиваться, учиться и стремиться вперед. Недаром Бертольд Авербах, немецкий писатель, сказал: «Нажить много денег – храбрость, сохранить их – мудрость, а умело расходовать – искусство»

Целью данного дипломного проекта является разработка и реализация приложения для видения бюджета, которая обеспечит полный контроль и проведение аналитики его состояния. В первую очередь данная система предназначена для людей делающих первые шаги в формировании и управлении своим бюджетом. Также система может использоваться уже опытными в данном вопросе людьми в качестве альтернативного варианта приложения для ведения бюджета.

В соответствии с поставленной целью были определены следующие задачи:

* выбор платформы для создания приложения;
* разработка пользовательской и административной частей приложения;
* реализация функциональности создания и управления бюджетом;
* построение системы создания отчетов;
* построение системы создания графиков;
* построение системы сбора и анализа данных;
* построение системы анализа голосовых команд;
* обеспечение высокой производительности и скорости работы приложения.

Проект будет состоять из веб- и мобильного приложения, которые будут предоставлять следующие функции:

* ввод данных с помощью голосовых команд;
* создание пользователей в системе;
* сбор данных о доходах и расходах пользователей;
* построение отчетов и графиков расходов пользователей;
* временное хранение данных при плохом интернет соединении;
* анализ и обработка данных пользователей.

**1** ОБЗОР ЛИТЕРАТУРЫ

## **1.1** Обзор существующих аналогов

На этапе проектирования системы были тщательно изучены существующие аналоги. Одним из наиболее приближенных примеров является приложение «Daily Budget» (рисунок 1.1).



Рисунок 1.1 – Главная страница приложения «Daily Budget»

Данный проект – это приложение для ведения бюджета. В нем реализована функциональность введения доходов и расходов. Все доходы и расходы разделяются на группы (рисунок 1.2), что удобно для группировки и представления данных. Это также удобно для отображения графиков ежемесячных затрат для каждой конкретной группы.

Также в приложении предусмотрено указание постоянного ежемесячного дохода, а также постоянных затрат, чтобы не вводить данную информацию ежемесячно вручную. Исходя из ежемесячного дохода и желаемой суммы для сохранения, высчитывается ежедневный бюджет, который доступен пользователю для достижения поставленной цели. Это сделано для того, чтобы наглядно на графике можно было убедиться, вкладывается ли пользователь в положенный ему ежедневный бюджет (рисунок 1.3).

Также, к основным недостаткам приложения относятся:

* разработано только под операционную систему (ОС) iOS;
* нет синхронизации данных для работы с разных устройств;
* все данные хранятся у пользователя в телефоне, что не безопасно;
* ограниченный набор графиков;
* жесткая привязка к работе только с одной валютой.



Рисунок 1.2 – Группы затрат приложения «Daily Budget»



Рисунок 1.3 – График попадания в бюджет приложения «Daily Budget»

Еще один аналог - приложение «ViZi Бюджет» (рисунок 1.4). Это приложения для ведения бюджета предоставляет больший набор функций в сравнении с предыдущим аналогом. Есть возможность синхронизации данных, введения собственных категорий затрат, а также возможность иметь несколько счетов. Приложение имеет бесплатную и платную версии, и представлено для двух мобильных платформ: iOS и Android.



Рисунок 1.4 –Приложение «ViZi Бюджет»

Данное приложение также не лишено недостатков:

* сложный пользовательский интерфейс;
* отсутствие веб-версии приложения;
* слишком большой набор функций для новичка.

## **1.2** Аналитический обзор

Веб-приложение – это клиент-серверное приложение, в котором клиентом выступает браузер, а сервером – веб-сервер. Логика веб-приложения распределена между сервером и клиентом, хранение данных осуществляется, преимущественно, на сервере, обмен информацией происходит по сети. Одним из преимуществ такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения являются кроссплатформенными сервисами. Книга Сэмма Руби «Гибкая разработка веб-приложений»[1] содержит информацию о том, как создать простое веб-приложение с нуля. В книгу включены описание простейшего рабочего процесса (с использованием текстового редактора и системы контроля версий Git), основы технологий клиентской стороны (HTML, CSS, jQuery, JavaScript), основы серверных технологий (HTTP, базы данных), основы облачного развертывания и несколько примеров правильной практики написания кода (функции, MVC). С ее помощью можно изучить фундаментальные основы языка, научиться программировать, используя объекты и массивы, а также ментальные модели, которые соответствуют этому типу разработки программного обеспечения (ПО).

Мобильное приложение – программное обеспечение, предназначенное для работы на смартфонах, планшетах и других мобильных устройствах [2]. Мобильные приложения ушли далеко вперед в связи с развитием сотовой связи и беспроводных технологий (3G, 4G или LTE, Wi-Fi, WiMax).

Многие приложения могут быть предустановлены на устройство в процессе производства, загружены пользователем с помощью различных платформ для распространения ПО или существовать в формате веб-приложений.

Основные операционные системы, на базе которых строятся мобильные приложения – iOS, Android и WindowsMobile.

Существуют различные каналы распространения приложений: специализированный онлайн магазин – AppStore, AndroidMarket, MicrosoftStore, самостоятельный поиск или скачивание в Интернете.

## **1.3** Архитектура клиент-сервер

Важным вопросом в разработке веб-приложений является их архитектура. Наиболее эффективную работу приложений обеспечивает архитектура «клиент-сервер» [3].

Особенностью архитектуры является то, что само веб-приложение находится и выполняется на сервере, клиент при этом получает только результаты работы. Работа приложения основывается на получении запросов от пользователя (клиента), их обработке и выдачи результата. Вся передача запросов и результатов их обработки происходит через Интернет.

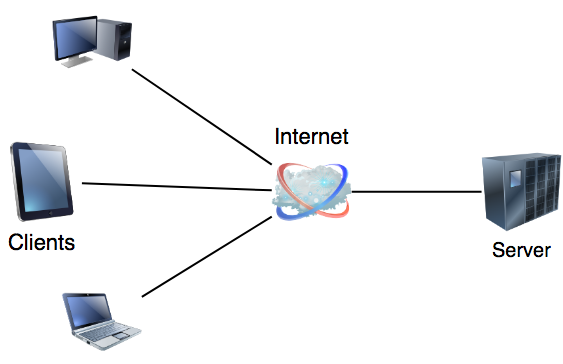


Рисунок 1.5 - Архитектура клиент-сервер

Приемом данных от клиента и их передачей на сервер, а также отображением результатов запросов обычно занимается специальное приложение – браузер (Microsoft Edge, Firefox, Safari, Google Chrome). Одной из функций браузера является отображение данных, полученных из Интернета, в виде страницы, описанной на языке HTML, следовательно, результат, передаваемый сервером клиенту, должен быть представлен на этом языке.

На стороне сервера веб-приложение выполняется специальным программным обеспечением (веб-сервером). Основные задачи, которые он решает – принимает запросы клиентов, обрабатывает их, формирует ответ в виде страницы, описанной на языке HTML, и передает его клиенту.

В процессе обработки запроса пользователя веб-приложение компонует ответ на основе исполнения программного кода, работающего на стороне сервера, веб-формы, страницы HTML, другого содержимого, включая графические файлы.

В результате, как уже было сказано, формируется HTML-страница, которая и отправляется клиенту. Получается, что результат работы веб-приложения идентичен результату запроса к традиционному веб-сайту, однако, в отличие от него, веб-приложение генерирует HTML-код в зависимости от запроса пользователя, а не просто передает его клиенту в том виде, в котором этот код хранится в файле на стороне сервера. То есть веб-приложение динамически формирует ответ с помощью исполняемого кода – так называемой исполняемой части. За счет наличия исполняемой части, веб-приложения способны выполнять практически те же операции, что и обычные Windows-приложения, с тем лишь ограничением, что код исполняется на сервере, в качестве интерфейса системы выступает браузер, а в качестве среды, посредством которой происходит обмен данными, – Интернет.

К наиболее типичным операциям веб-приложения относятся:

* прием данных от пользователя и сохранение их на сервере;
* аутентификация пользователя и отображение интерфейса системы, соответствующего данному пользователю;
* отображение постоянно изменяющейся информации;
* выполнение различных действий по запросу пользователя: извлечение данных из базы данных (БД), добавление, удаление, изменение данных в БД;
* проведение сложных вычислений.

К основным достоинствам архитектуры «клиент-сервер» можно отнести следующие:

* все данные хранятся на сервере, который защищен лучше большинства клиентов;
* использовать ресурсы сервера могут клиенты с разными аппаратными платформами и операционными системами;
* возможность распределить функции вычислительной системы между несколькими компьютерами в сети, что позволяет упростить обслуживание системы;
* на сервере проще организовать контроль прав и полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами;
* позволяет объединить различные клиенты.

Среди недостатков можно выделить необходимость квалифицированного администратора данной системы. В случае использования централизованной системы, неработоспособность основного сервера может сделать неработоспособным все приложение. Немаловажным фактором является высокая стоимость оборудования.

## **1.4** REST-сервис

REST – это набор архитектурных принципов и стиль проектирования приложений, ориентированный на создание сетевых систем, в основе которых лежат механизмы для описания и обращения к ресурсам [4].

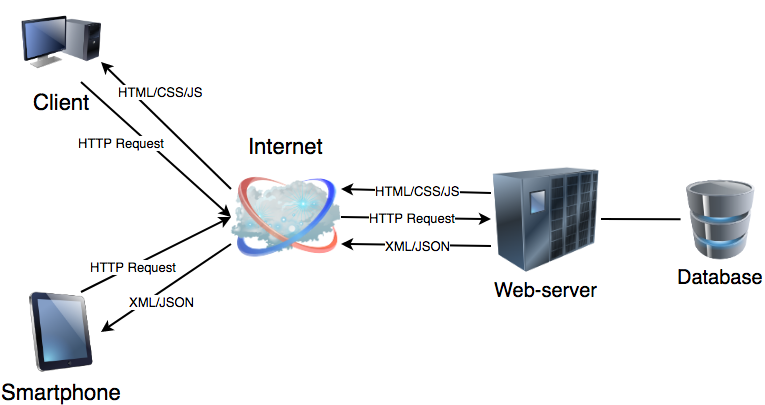


Рисунок 1.6 - REST-сервис

Примером такой системы служит World Wide Web – распределенная система, предоставляющая доступ к связанным между собой документам, расположенным на различных компьютерах, подключенных к сети Интернет [5].

REST определяет строгое разделение ответственности между компонентами клиент-серверной системы, облегчающее реализацию необходимых актеров. Еще одна цель REST – упрощение семантики взаимодействия компонентов сетевых систем для улучшения масштабируемости и повышения производительности. В REST заложен принцип автономности запросов, который означает, что запросы, обрабатываемые сервером или клиентом, обязательно должны включать всю информацию, необходимую для их понимания.

Для обмена данными стандартных медиа-типов в системах REST используется минимальное количество запросов. REST-системы используют URI (универсальные идентификаторы ресурсов) для поиска и получения доступа к представлениям необходимых ресурсов. В течение последних нескольких лет разработчики создавали REST-сервисы для своих приложений, используя самые разнообразные технологии. Также архитектура REST отличается своей простотой, требуя от приложений обеспечить только возможность приема сообщений с HTTP- заголовками.

## **1.5** Apache Cordova

Apache Cordova — это платформа разработки мобильных приложений с открытым исходным кодом. Она позволяет использовать стандартные веб-технологии, такие как HTML5, CSS3 и JavaScript для кроссплатформенной разработки, избегая родного языка разработки для каждой из мобильных платформ. Приложения выполняются внутри обертки нацеленной на каждую платформу и полагаются на стандартные API (программный интерфейс приложения) для доступа к датчикам устройства, данным и состоянию сети [6].

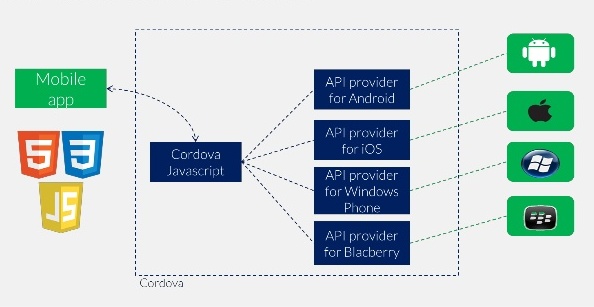


Рисунок 1.7 - Программный интерфейс приложения Cordova [7]

Основные причины использования Apache Cordova:

* нужно расширить приложение на более чем одну платформу, без необходимости повторно реализовать его для каждого языка разработки платформ и набора инструментов;
* нужно развернуть веб-приложение, которое упаковано для распространения в различных магазинах приложений;
* необходимо смешивание компонентов собственного приложения с WebView (специальный браузер окно).

Само приложение реализовано как веб-страницы, по умолчанию локальный файл под названием index.html, который ссылается на любой CSS, JavaScript, изображения, файлы мультимедиа или другие ресурсы необходимы для его запуска. Приложение выполняет как WebView в пределах оболочки приложения, которую вы распространяете в магазины приложений.

Таким образом, с помощью данной платформы есть возможность писать слабонагруженные приложения, который будут работать на любой мобильной платформе. При этом затраты на разработку будут минимальны, поскольку код будет написан один раз, а запускаться будет как в браузере, так и на любом мобильном устройстве.

**2** СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Изучив теоретические аспекты разрабатываемой системы и выработав список требований необходимых для разработки системы, приложение разбивается на функциональные блоки (модули). Это необходимо для обеспечения гибкой архитектуры. Такой подход позволяет изменять или заменять модули без изменения всей системы в целом.

В разрабатываемом веб-приложении можно выделить следующие блоки:

* блок пользовательского интерфейса;
* блок ядра клиентского приложения;
* блок для работы с клиентской базой данных;
* блок работы с клиентским приложением;
* блок регистрации;
* блок веб-сервиса;
* блок работы с реляционной базой данных;
* блок ядра серверного приложения.

Структурная схема, иллюстрирующая перечисленные блоки и связи между ними приведена на чертеже ГУИР.400201.216 C1.

Каждый модуль выполняет свою задачу. Чтобы система работала, каждый модуль взаимодействует с другими модулями путем обмена данными, используя различные форматы и протоколы.

Рассмотрим функциональные блоки веб-приложения.

*Блок пользовательского интерфейса* является клиентской частью веб-приложения. Данный блок представляет собой совокупность средств, при помощи которых пользователь взаимодействует с приложением через браузер. Для построения интерфейса используется технологии HTML, CSS, TypeScript.

TypeScript — язык программирования, представленный Microsoft в 2012 году и позиционируемый как средство разработки веб-приложений, расширяющий возможности JavaScript. Строгость и большая функциональность языка позволяет разработать библиотеку компонент, с помощью которой будут строиться основные компоненты клиентского интерфейса [8].

*Блок для работы с клиентской базой данных* предоставляет доступ к локальному хранилищу браузера, где хранятся авторизационные данные пользователя, а также данные, которые по тем или иным причинам не были отправлены на сервер.

Интернет-хранилище – это программные методы и протоколы веб-приложения, используемые для хранения данных в веб-браузере. Интернет-хранилище представляет собой постоянное хранилище данных, похожее на куки, но со значительно расширенной емкостью и без хранения информации в заголовке запроса HTTP. Существуют два основных типа веб-хранилища: локальное хранилище и сессионное хранилище, ведущие себя аналогично постоянным и сессионным кукам соответственно.

Таким образом, данный блок представляет собой временное хранилище данных пользователя до ближайшего сеанса синхронизации с сервером.

*Блок ядра клиентского приложения* представляет собой центр управления всеми частями клиентской части приложения. Данный блок знает обо всех блоках на клиентской стороне, организует совместную работу всех блоков, а также организует работу с блоком для работы с клиентским приложением.

*Блок работы с клиентским приложением* представляет собой центр для связи клиентской части с серверной. Через него проходят все операции синхронизации и обмена данными.

*Блок регистрации* служит для регистрации новых пользователей приложения. Поскольку регистрация пользователей приложения содержит логику отличную от основной логики работы приложения, имеет смысл вынести его реализацию в отдельный блок.

*Блок веб-сервиса* представляет встроенное приложение с использованием фреймворка ExpressJS, который предназначен для удобного построения API веб-сервисов.

*Блок работы с реляционной базой данных* включает механизмы для взаимодействия с данными, используемыми приложением. Для ее реализации была выбрана база данных SQLite. SQLite – компактная встраиваемая реляционная база данных, запросы к которой можно осуществлять при помощи языка запросов SQL. База данных не поддерживает все особенности SQL и уступает в функциональности другим развитым СУБД, но вполне подходит для хранения и извлечения информации.

SQLite не использует парадигму клиент-сервер, то есть движок SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а предоставляет библиотеку, с которой программа компонуется и движок становится составной частью программы. Таким образом, в качестве протокола обмена используются функции (API) библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том компьютере, на котором исполняется программа. Простота реализации достигается за счет того, что перед началом исполнения транзакции записи весь файл, хранящий базу данных, блокируется; ACID-функции достигаются в том числе за счет создания файла журнала [9].

Несколько процессов или потоков могут одновременно без каких-либо проблем читать данные из одной базы. Запись в базу можно осуществить только в том случае, если никаких других запросов в данный момент не обслуживается; в противном случае попытка записи оканчивается неудачей, и в программу возвращается код ошибки. Другим вариантом развития событий является автоматическое повторение попыток записи в течение заданного интервала времени.

Благодаря архитектуре движка возможно использовать SQLite как на встраиваемых системах, так и на выделенных машинах с гигабайтными массивами данных.

*Блок ядра серверного приложения* представляет собой центр управления всеми частями серверной части приложения. Данный блок знает обо всех блоках на серверной стороне, организует совместную работу всех блоков.

Блок сервера работает на платформе NodeJS. NodeJS является платформой для создания современных веб-сервисов. Он позволяет писать асинхронный, неблокирующий код, используя стиль и шаблоны NodeJS. Позволяет использовать возможности баз данных, таких как Elasticsearch и Redis. Работать со многими протоколами, создавать веб-службы RESTful, клиенты и серверы сокетов TCP. В платформу встроена возможность проверять функциональность кода с помощью библиотеки Mocha и управлять его жизненным циклом с помощью менеджера пакетов NPM (Node Packet Manager). Используя ряд практических областей программирования, возможно использовать новейшие доступные функции ECMAScript, классы, и популярные модули, а также создавать богатые инструменты командной строки и веб-интерфейс с использованием современных технологий веб-разработки [10].

**3** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

## **3.1** Проектирование программ сверху вниз

Самое главное условие успешного создания как крупного, так и небольшого приложения заключается в применении надежных методов проектирования.

Самое широкое распространение при написании такого рода программ получили следующие методы:

* нисходящий (сверху-вниз);
* восходящий (снизу-вверх);
* специальный (на данный конкретный случай).

В случае нисходящего метода процесс разработки программы начинается с программы высокого уровня и спускается до программ низкого уровня.

В свою очередь восходящий метод работает в обратном направлении, то есть сначала идет разработка отдельных специальных подпрограмм, на основании которых в последствии строятся более сложные, которые заканчиваются самым верхним уровнем приложения.

Специальный подход не имеет какого-то заранее установленного способа разработки, то есть решения в выборе метода происходят во время процесса разработки и могут меняться в любой момент времени.

JavaScript является, в одно время, как структурированным так и не структурированным языком. Именно поэтому в качестве разработки был выбран язык TypeScript. Он представляет собой строгий и структурированный JavaScript и лучше всего подходит для нисходящего программирования. Нисходящий подход позволяет писать ясный, легко читаемый код, что в дальнейшем не вызовет проблем с сопровождением приложения. К тому же данный метод отлично подходит для прояснения и создания всей структуры программы в целом до кодирования более низких уровней приложения.

Выбранный метод позволяет уменьшить потери времени, обусловленные неудачными и ошибочными решениями на начальном этапе. А в условиях небольшого срока разработки программы – это делает его основным методом разработки и позволяет перейти к проектированию программы.

## **3.2** Структурирование программы

Как описывалось выше, для любой общей схемы при применении метода «сверху-вниз» разработку начинают с общего описания приложения, а после этого двигаются в направлении проработки ее конкретных деталей, компонентов и прочего.

При разработке любой программы лучше всего точно определить, что и как программа будет делать на самом высоком уровне, а затем уже погружаться в детали, касающиеся каждого ее действия. На рисунке 3.1 представлена схема нисходящего метода разработки.



Рисунок 3.1 – Схема нисходящего метода разработки программы

Исходя из изображения выше можно заметить, что эта схема очень легко ложиться на структурную схему разрабатываемого приложения, что говорит о правильном выборе метода разработки. Согласно схеме, можно легко определить и выделить два основных модуля программы. Модулей два потому как приложение содержит клиентскую и серверную части, каждая из которых служит обособленным модулем высокого уровня. К модулям высокого уровня относятся:

* блок ядра клиентского приложения;
* блок ядра серверного приложения.

Теперь можно приступать к составлению перечня действий, которые будет выполнять каждый отдельный модуль.

## **3.3** Блок ядра серверного приложения

Разработка начинается с ядра серверного приложения потому как разрабатывать клиентское ядро без минимально функционирующей серверной части не имеет смысла.

Первая и самая основная цель серверной стороны – это запускать сервер. Так как сервер будет работать на платформе NodeJS, то самый удобный и безопасный способ работать с сервером – это подключить библиотеку ExpressJS.

Это уже давно зарекомендовавший себя модуль, который имеет большой и гибкий набор конфигураций.

Необходимо добавить данную библиотеку в проект с помощью NPM-менеджера. Для этого нужно использовать команду: «nap install express --save». После установки нужно подключить модуль к основному файлу сервера и произвести пробный запуск сервера на 1337 порту. По умолчанию, сервер будет запущен на локальном адресе компьютера без каких-либо дополнительных настроек. Пробный запуск сервера представлен на рисунке 3.2.



Рисунок 3.2 – Тестовый запуск сервера на ExpressJS модуле

После успешного тестового запуска сервера можно приступить к настройке его конфигураций.

Имеет смысл вынести конфигурацию и настройки сервера в отдельный файл. Это упростит работу с проектом и дальнейшее ориентирование в файлах проекта. Сервер должен быть достаточно безопасным, так как будет хранить данные бюджета пользователей. Здесь будет использоваться подход, при котором к серверу будет открыто как можно меньше путей. Именно поэтому в конфигурации сервера нужно закрыть доступ ко всем скрытым файлам, начинающимся с точки. Также в конфигурации слудет указать, чтобы по умолчанию открывались только файлы с расширением html и не разрешалась никакая переадресация.

Сервер будет содержать и статический данные, такие как скрипты, файлы стилей и скриптов. Имеет смысл перенести статические файлы в одну папку и запасть сервер для работы именно с ней в качестве корневого каталога. Таким образом доступ к серверу будет строго ограничен статическими файлами.

Для обмена данными между клиентом и сервером будет использоваться формат JSON – это текстовый формат обмена данными, основанный на JavaScript. По умолчанию данный модуль не включен в конфигурацию сервера, поэтому его необходимо добавить через NPM-менеджер, а затем добавить в конфигурацию.

Постоянно загружать статические файлы по запросу клиента не имеет смысла, именно поэтому в конфигурацию также включается настройка Кеша. В настройке нудно указать максимальный срок кеширования файлов, в данном случае это будет один день. А также необходимо включить использование Etan – часть HTTP протокола. Данная часть HTTP является одним из механизмов, который помогает обеспечивать веб-проверку кэша, а также помогает пользователю делать условный запрос к серверу. Данный механизм позволяет делать кэш более эффективным, что экономит пропускную способность, так как серверу не нужно каждый раз отправлять полные ответ, ведь содержимое не было изменено с последнего обращения. Он также может быть использован для рационального управления много поточностью в качестве способа предотвращения одновременного обновления и перезаписи ресурса.

Etag является закрытым идентификатором, который присеваются сервером на определенную версию ресурса, найденного по адресу. И если же содержание ресурса по данному адресу меняется на новое, то назначается новый номер Etag. Использование данного механизма аналогично работе сканера отпечатка пальцев, ведь можно быстро сравнить и определить версии ресурса на уникальность. Сравнивать Etag можно и нужно только с Etag того же адреса, так как идентификаторы от разных ресурсов могут быть как одинаковыми так и разными, вне зависимости от ресурсов, так как сравнение не имеет какого-либо смысла.

Также для кеширования будет использоваться свойство заголовка «Last modified». Данный заголовок также служит в качестве элемента кеширования данных. В нем содержится дата последнего изменения файла и если при запросе, клиентские браузер видит, что дата изменения не менялась, то подгружает файл из кэша браузера.

На данном этапе настройка конфигурации сервера заканчивается. Теперь сервер будет раздавать статический файлы, а значит можно заняться разработкой статических файлов на клиентской стороне.

## **3.4** Блок ядра клиентского приложения

Основное ядро клиентской части приложения будет содержаться в JavaScript файле. Однако его нужно загрузить в HTML файл. HTML файл будет статически загружаться с сервера, автоматически загружать статические стили и основной JavaScript файл, после чего начнется построение всех частей приложения из основного ядра.

В HTML файлe указываются пути для загрузки скриптов и стилей, также указываются мета-теги для оптимизации страницы браузера под мобильные устройства. Как описывалось ранее, в дальнейшем приложение будет обернуто в приложение Cordova. Следовательно функции зумирования страницы должны быть отключены.

В теле страницы описываются основные элементы страницы, на основании которых будут строиться остальные части программы. К данным элементам относятся:

* блок заголовка;
* блок меню;
* блок основных элементов.

Блок заголовка постоянно виден на экране и должен содержать только элементы которые всегда должны быть на виду. В данный момент это иконка для открывания и закрывания блока меню.

Блок меню, как и блок заголовка, представляют из себя сруб-модули. Все, что нужно блоку меню для работа – это список страниц и ссылок на них, а также элемент меню, который находится в статическом HTML файле. На основании полученных данных, строится список всех элементов меню, после чего, он добавляется к элементу меню. На этом работа модулю заканчивается. Даже несмотря на то, что модуль получается небольшим, его стоит отделять в качество независимого модулю на случай разрастания сложности приложения.

Как можно понять из абзаца выше, приложение будет иметь несколько страниц. Но каждый раз загружать новую страницу, как это делается на сайтах, не выгодно. Именно поэтому приложением будет представлять из себя одностраничное веб-приложение.

Одностраничное веб-приложение – это приложение, все функции которого отображаются без перехода на другие страницы, то есть загрузка нового контента происходит через запросы. Eсил контент погружается через запросы, то стоит сразу поговорить о преимуществе данного подхода:

* снижение нагрузки на сервер – запрашивается только нужная часть информации, то есть данные, а не вся страница;
* сервер передает именно данные (обычно через JSON), а не обрабатывает и отрисовывает всю страницу по-новому.

В следствии этого, при разработке сильно разделяется серверная и клиентская части. Сервер, при этом, обрабатывает лишь API запросы, а за дизайн и отображение информации отвечает клиентская сторона. Сразу видно, что при таком подходе имеет смысл разделить программистов на две команды – клиентскую и серверную части. Как видно, одностраничные приложения – это не только способ отображения информации, но и совершенно иной организационный поход к разработке.

Область применения одностраничным веб-приложений довольно широка. Помимо снижения нагрузки на сервер и экономии траффика между клиентом и сервером одна из центровых причин для такого подхода к разработке была создание веб-приложений с максимально приближенным к наивному поведению. То есть если открыть тот же фотошоп – он не прогружается заново в тот момент, как открывается новый документ. Исходя из того, что данное приложение будет обернуто в мобильное приложение и должно быть похоже на нативное приложение, и была выбрана разработка одностраничного веб-приложения.

Для работы с адресами нужен отдельный модуль, потому как в адресе странице будет указывать не только текущая страница, но необходимые параметры для ее правильной работы. Основная работа данного модуля – это уметь обрабатывать строку адреса в удобный для работы JSON формат. Это действие должно работать и в обратную строну, то есть при получении объекта в формате JSON, он должен быть преобразован в строку для дальнейшей вставки в адресную строку браузера. Другой неотъемлемой частью данного модуля является прослушивание события браузера при любом изменении адреса строки – это необходимо для распознавания перехода на новую страницу пользователя. После нажатия пользователя на другую страницу или при изменении параметров данной страницы срабатывает событие об изменении адрес. Далее модуль берет новое значение адреса преобразует его в формат JSON и информирует вышестоящий модуль об изменении в состоянии приложения. Вышестоящий модуль обновляет состояние приложения исходя из нового состояния адреса.

Для более понятной работы с адресами будет использоваться следующий формат адреса: имя страницы будет сопровождаться знаком «#», а далее через запятую будут идти все остальные параметры так, как это представлено на рисунке нижу.



Рисунок 3.3 – Формат адреса страницы

Так в приложении будет несколько страниц, то нужен сруб-модуль для работы со страницей. Основные функции данного модуля:

* хранение объектов всех страницы приложения;
* хранение текущей активной страницы;
* создание страниц, если они не были созданы до этого;
* получение объекта новой страницы, исходя из нового состояния приложения;
* замена текущей активной страницы новой;
* передача новой странице нового состояния приложения.

Перейдем к созданию страницы – это будет еще один суб-модуль. Точнее это будет несколько модулей, каждый из которых представляет из себя отдельную страницу приложения. Это очень удобный подход, поскольку каждая страница обладает своей логикой и нет смысла описывать все в одном месте, к тому же это упрощает дальнейшую работы с каждой отдельной станицей, поскольку сразу понятно в каком модуле необходимо работать.

Несмотря на отдельную логику каждой страницы, все они должны иметь стандартный набор методов, чтобы модуль работы со страницами мог с ними коммуницировать. Каждая страница должна иметь следующие методы:

* initialize;
* focus;
* blur;
* pageName.

Метод «initialize» – это метод, который используется для инициализации страницы. То есть он вызывается только при первом создании страницы и носит конфигурационный характер. Во время выполнения данного метода созданются все объекты страницы, элементы и начальные состаяния.

Далее идет метод «focus». Данный метод вызывается каждый раз, когда происходит изменение состояния приложения, а именно, когда пользователь переходит на страницу данного модуля или когда он меняет какие-либо параметры для данной страницы. Исходя из данного описания можно понять, что данный метод служит отправной точкой для работы со страницей.

Метод «blur» – это метод, служащий для завершения работы страницы. Данный метод вызывается для активной страницы, когда происходит переключению на другую страницу. Во время вызова данного метода, текущая страница, своего рода, ставится на паузу, при этом она отменяет все текущие операции и запросы, закрывает все диалоговые окна, если таковые имеются. Этот метод очень необходим, так как позволяет освободить память под другую страницу, что значительно увеличивает производительность всего приложения.

Метод «pageName» – это метод, служащий для возвращения имени страницы. Он используется модулем работы со страницами для поиска страниц.

Станицы могут также содержать и другие методы, касающиеся только конкретной страницы. Эти методы должны быть закрыты для вышестоящих модулей и никак не должны зависеть от других модулей приложения.

Вернемся к модулю работы со странницами. Без описания структуры каждой отдельной страницы нельзя было обойтись для долившего описания структуры данного модуля.

При изменении состояния приложения данный модуль должен как-то найти необходимую ему страницу. Для этого он хранит в себе хеш-таблицу всех известных ему проинициализированных страниц.



Рисунок 3.4 – Диаграмма деятельности изменения страницы

Каждый раз когда происходи изменение состояния, модуль берет имя новой страницы и ищет его хеш-таблице. Если страница не была найдена, значит на нее заходят впервые, модуль инициализирует объект данной станицы и выполняет повторный поиск. Как только страница будет найдена, модуль проверяет не является ли данная страница активной, ведь возможен вариант, что пользователь уже находиться на данной станице, но он лишь поменял конфигурации данной страницы. Если страница тоже самая, то все, что нужно сделать, это лишь вызвать метод «focus» у текущей страницы и передать в нее новое состояние. Если же новая страница не является текущей, то вызывается метод «blur» для текучей страницы, затем новая страницы становится текущей для данного модуля, и затем для уже новой страницы вызывается метод «focus» с новым состоянием. Схема работы модулю отражена на рисунке 3.4.

Может возникнуть ситуация, при которой в адресной строке будет указана страница, которая не известна модулю. В таком случае модуль всегда имеет страницу по умолчанию, на которую перейдет, если подобное случится.

Исходя из описанных методов видно, что дальнейшая разработка клиентской стороны заключается в написании каждой отдельной станицы и ее внутренней логики, однако в конечном итоге все упреться в совместную работу клиентской и серверной сторон. Для этого необходима разработка блока работы с клиентским приложением.

## **3.5** Блок работы с клиентским приложением

Блок работы с клиентским приложением представляет из себя API для работы сервером. Как уже описывалось ранее, сервер должен быть максимально закрытым для всем подключений и запросов, поэтому API должен быть доступен строго по одному адресу на сервере, а остальные его действия будут зависеть от параметров переданных в запросе. Для большей безопасности будет использоваться POST-запрос.

POST-запрос является одним из многих методов запроса, которые поддерживает проток HTTP. Данный метод предназначен для создания запроса, при котором веб-сервер получает данные от клиента, которые хранятся в теле запроса для хранения. Чаще всего он используется для загрузки файлов либо представления заполненной формы.

В отличии от POST-запроса, другой похожий метод GET предназначен для получения информации от сервера. В рамках данного запроса некоторые данные имеют возможность быть переданными в строке адреса, и могут указывать, например, условия поиска или диапазоны дат, другую информацию, определяющую адрес. В рамках POST-запроса может быть отправлен произвольный объем данных любого типа на сервер в теле запроса. Заголовки в данном запросе чаще всего указывают на тип содержимого запроса.

Для упрощения работы с API он должен иметь строгую, но при этом удобную для понимания и работы структуру. Поэтому его архитектура должна быть схожа с архитектурой модуля для работы со страницами, работа которого описывалась в предыдущем разделе. То есть должен быть суб-модуль, который знает все известные и возможные API-запросы. К основным функциям главное API-модуля можно отнести:

* хранение объектов всех возможных API-сущностей;
* расшифровка запросов от клиента;
* вызов методов API-сущностей в соответствии с запросом;
* предотвращение вызовов несуществующих API-запросов.

API-сущность – это отдельный модуль для работы с конкретным элементов базы данных. То есть для каждой таблицы базы данных должна существовать своя сущность. Сделать это следует для того, чтобы каждая сущность выполняла операции только с одной таблицей и никак не затрагивала другие. Это делает архитектуру более прозрачной и помогает при дальнейшем развитии проекта, ведь каждая таблица имеет свой свойства и конфигурацию.

Однако для работы модуля API все сущности должны иметь общий интерфейс – методы, которые будет вызывать API-модуль. Для этого каждая сущность должна содержать следующие методы:

* add;
* remove;
* get;
* set.

Метод «add» служит для создания новой записи в таблицу. Он также имеет свой интерфейс входных параметров, который описывает чуть выше с использованием языка TypeScript. Это очень удобная вишь, так как данный интерфейс в дальнейшем будет использоваться на клиенте при написании кода, то есть программист всегда может увидеть набор необходим параметров для данного вызова метода.

Использование интерфейсов делает разработку одновременно строгой и, в тоже время, удобной и безопасной, так как при малейшем несовпадении интерфейса TypeScript выдаст ошибку о несовпадении структуры данных или типов данных. Как метод «add», так и остальные методы сущности должны возвращать в качестве результата Promise.

Promise (другое название «промис») – предоставляют удобный способ организации асинхронного кода. Но представляет собой специальный объект, который содержит свое состояние. Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).

Объект Promise очень важная в разработке вещь, так как делает работу сервера и клиента асинхронной.

Без данной возможности страничка браузера бы просто непросто замирала на время обработки операции.

Кратко описать работу Promise можно следящим образом:

* код, которому нужно выполниться асинхронно, создает объект Promise и возвращает его;
* внешняя функция, получает объект Promise и слушает его обработчики;
* при завершении процесса асинхронный код переводит Promise в одно из описанных выше состояний, то есть fulfilled или rejected, при этом автоматически вызываются обработчики во внешней функции.



Рисунок 3.5 – Состояния объекта Promise

Метод «remove» служит для удаления элемента из таблицы. Обязательным для него параметром служит идентификационный номер строки для удаления.

Метод «get» служит для получения данных из таблицы. Параметр при вызове данного метода должен быть опциональным. То есть при вызове метода без параметра должны возвращаться все данные из таблицы, но при наличии параметра, поиск должен осуществляться сугубо по полученным параметрам. Для каждой API-сущности должен быть описан свой интерфейс для поиска – это также будет помогать разработку клиентской стороны делать необходимые и правильные запросы.

Метод «set» предназначен для обновления записи таблицы. Он также имеет обязательный параметр с интерфейсом всех полей таблицы, чтобы программист не потерял в процессе разработки одно из полей.

Таким образом описывается каждая сущность API. В результате чего все сущности хранятся в виде хеш-таблицы в API-модуле для быстрого нахождения и работы.

И все же, основным методом данного модуля является метод обработки запросов клиентской стороны. После получения запроса от клиента в формате JSON, модуль должен убедиться, что данный запрос возможен, поэтому первым делом к какой именно сущности направлен запрос.



Рисунок 3.6 – Диаграмма деятельности обработки API-запроса

Если сущность существует, то проверяется, какой метод данной сущности будет вызван – и только после данной проверки API-модуль берет необходимую сущность из хеш-таблицы и вызывает необходимый метод с нужными параметрами. Если же какой-то из этапов работы API-модуля не удался, то он возвращает ошибку, которая в итоге отправляется в качестве ответа клиенту.

Для увеличения надежности API-модуль так провеет заголовки каждого запроса, а именно:

* метод запроса;
* хост запроса;
* тип данных запроса.

Метод запроса всегда должен быть POST, а запроса должны приходить искочительно от того же хоста, на котором запушен сервер, чтобы другие ресурсы не могли получить доступ к информации. Так как сервер настроен на работу с данными в формате JSON, то имеет смысл делать проверку на тип данных запроса, он должен быть всегда «application/json». При несовпадении одного из заголовком API-модуль так же вернет в качестве ответа ошибку. На рисунке 3.6 представлена диаграмма деятельности по обработке API-запроса.

На этом работа над функциональностью API-модуля окончена, теперь имеет смысл вернуться на клиентскую сторону и разработать модуль для работы с API.

## **3.6** Клиентский блок для работы с API

Для удобной работы с API необходим модуль, который будет основную работу по созданию запроса на север, заполнению нужных заголовков, а также проверке результата запроса.

Основное, что делает данный модуль – это создает Fetch-запрос. Это XMLHttpRequest-запрос нового поколения. Он предоставляет улучшенный интерфейс для осуществления запросов к серверу: как по части возможностей и контроля над происходящим, так и по синтаксису, так как построен на Promise.

При вызове Fetch возвращает промис, который, когда получен ответ, выполняет коллбэки с объектом Response или с ошибкой, если запрос не удался.

Данный метод принимает два параметра: адрес, куда пойдет запрос, и опции:

* метод запроса;
* заголовки запроса;
* тело запроса;
* мод, который указывает режим кросс-домашности, в котором должен быть выполнен запрос;
* кэш, который указывает как кешировать запроса и нужно ли это делать;
* переадресация, которая указывает как должен вести запрос при возникновении ошибки с кодами 300.

Как видно, всевозможных настроек у данного типа запросов очень много, поэтому именно он используется в качестве основного метода для отправки запросов на сервер.

При отправке запроса заполняется необходимый набор параметров и заголовков, а так же тело запроса.



Рисунок 3.7 – Диаграмма деятельности модуля клиентского API

После получения ответа от сервера, модуль должен проверить соответствуют ли заголовки ответа ожидаемым и, если все проверки пройдены, то модуль преобразует данные в JSON формат и отправляет в вышестоящий модуль для дальнейшей обработки.

Если же в процессе выполнения запроса произошла ошибка или ответ не соответствует ожиданиям, то модуль возвращает в вышестоящий модель ошибку. На рисунке 3.7 изображена работа модуля по отправке и обработке запроса к серверу.

Теперь связь клиента и сервера налажена и из любой части клиентского приложения можно запросить необходимые данные. Однако каждый раз вызывать метод клиентского API и вписывать в него необходимый набор параметров очень накладно и небезопасно, так как при любом изменении в API нужно будет делать правки по всему проекту. Поэтому имеет смысл сделать, своего рода, обертку над клиентским API.

Данная обертка будет содержать в себе все известные типы запросов API, а также все их возможные методы, как это показано на рисунке 3.8. То есть разработчику будет нужно лишь пройти по цепочке методов модуля и в вызванном методе лишь передать параметр соответствующий нужному интерфейсу вызова.



Рисунок 3.8 – Обертка методов для работы с клиентским API

Данная модель значительно упрощает работу разработчика, а также увеличивает надежность кода. Это также сказывается и на дальнейшем сопровождении продукта, так как при малейшем изменений в API или добавлении новых типов запросов, все правки нужно будет делать лишь в одном файле, что является большим плюсом. На рисунке 3.9 представлен пример удобного использования обертки над клиентским API модулем.



Рисунок 3.9 – Пример вызова методов обертки над клиентским API

Теперь клиентская и серверная части полностью готовы к взаимодействию друг с другом. И следующим этапом будет создание блока для работы с базой данных.

## **3.7** Серверный блок для работы с базой данных

Основной упор в разрабатываемом программном комплексе делается на обработку хранимых в БД данных. Доступ к этим данным обеспечивает СУБД SQLite. Рассмотрим подробнее структуру используемой базы данных.

Базы данных создаются для доступа к данным и их хранения, содержат данные о некой предметной области, то есть к некоторой области деятельности человека или области реального мира. База данных представляет из себя систему данных о конкретной предметной области. Базы данных, которые относятся к одной и той же области, содержат более или менее детализированную и полную информацию о ней. Уровень полноты определяется перечнем факторов, прежде всего, целью использования данных из базы данных и трудностью производственных процессов, имеющихся в пределах данной области и конкретных условий.

Базы данных представляют собой связанную цепь или совокупность таблиц баз данных, связь между которыми может находиться в структуре данных, а может только предполагаться, то есть иметь смысл на неформальном уровне.

Каждая таблица базы данных представляется совокупностью столбцов и строк, столбцы которой соответствуют атрибутам объектов, либо событиям или явления, а стоки – экземплярам объекта.

Рассмотрим подробную структуру используемой в дипломном проекте базы данных. Необходимо отметить, что в данном проекте будет использоваться не одна база данных. Будет существовать одна общая база, в которой будут находиться данные о пользователях системы, активных сессиях, а также о общих доходах пользователей за весь прошлый период. Другие базы данных будут иметь одинаковую структуру и будут служить для работы только с одним пользователем. Это делается из соображений безопасности и приватности данных.

База данных пользователя представляет собой 6 таблиц. Проведем описание сущностей.

Сущность «currency» содержит в себе информацию о видах валют, в которых могут вноситься данные. Данная таблица представляет из себя два поля, одно из которых уникальный идентификационный номер, второй – имя валюты. Описание ее атрибутов приведено в таблице 3.1.

Таблица 3.1 – Атрибуты сущности «currency»

|  |  |
| --- | --- |
| Имя поля | Формат поля |
| id | into |
| name | text |

В данной сущности индекс: «id» – первичный уникальный ключ.

Сущность «incoming» содержит в себе информацию о всех затратах и прибылях пользователя, то есть все платежи совершаемые пользователем будут храниться в данной таблице. В данной таблице большое количество атрибутов. Поле «value» хранить в себе сумму затраты пользователя, поле «date» – дату совершения платежа, поле «comment» содержит описание затраты, поле «typeId» указывает к какому типу затрат относится данная запись, поле «currencyId» указывает в какой валюте свершалась данная затрата, поле «tags» носит опциональный характер и также хранит комментарий пользователя, поле «is Active» показывает является ли данная запись активной. Описание данной сущности представлено в таблице 3.2.

Таблица 3.2 – Атрибуты сущности «incoming»

|  |  |
| --- | --- |
| Имя поля | Формат поля |
| id | int |
| date | int |
| value | real |
| isActive | int |
| typeId | int |
| currencyId | int |
| comment | text |
| tags | text |

В данной сущности индекс: «id» – первичный уникальный ключ, а поля «typeId» и «currencyId» – внешние ключи к другим таблица. Внешний ключ – ключ, используемый для объединения двух таблиц. Реже его также называют ссылочным ключом. Данный ключ — это столбец или комбинация столбцов, значения которых соответствуют первичному ключу в другой таблице.

Связь между двумя таблицами задается через соответствие первичного ключа в одной из таблиц внешнему ключу во второй. Если для таблицы первичный ключ задан в определенном поле, то в этом поле не может содержаться двух записей с одинаковыми значениями.

Сущность «stable\_income» содержит в себе информацию о стабильных доходах пользователя, другими словами, те доходы, которые имеет постоянный характер должны храниться в данной таблице – это может быть, например, заработная плата.

В данной таблице большое количество атрибутов. Поле «value» хранить в себе сумму прибыли пользователя, поле «date» – дату совершения платежа, поле «name» содержит описание прибыли, поле «currencyId» указывает в какой валюте храниться данная прибыль. Описание данной сущности представлено в таблице 3.3.

Таблица 3.3 – Атрибуты сущности «stable\_income»

|  |  |
| --- | --- |
| Имя поля | Формат поля |
| id | int |
| date | int |
| value | real |
| currencyId | int |
| name | text |

В данной сущности индекс: «id» – первичный уникальный ключ, а поле «currencyId» – внешние ключи к таблице «currency».

Сущность «stable\_waste» содержит в себе информацию о стабильных затратах. Так или иначе у каждого есть ежемесячные платежи имеющие постоянный характер и вносить их каждый месяц по новой не имеет смысла. К таким платежам можно отнести плату за паркинг, коммунальные расходы, плата по кредиту. В данной таблице большое количество атрибутов.

Поле «value» хранить в себе сумму стабильных затрат пользователя, поле «date» – дату совершения платежа, поле «name» содержит описание затраты, поле «currencyId» указывает в какой валюте свершалась данная затрата, поле «isActive» показывает является ли данная запись активной.

Описание данной сущности представлено в таблице 3.4.

В данной сущности индекс: «id» – первичный уникальный ключ, а поле «currencyId» – внешние ключи к таблице «currency».

Сущность «tags» содержит в себе информацию о персональных комментариях пользователя. Они имеют очень важную ценность, так как иногда бывает ситуации, когда платежи имеют разные типы, но при этом пользователь хочет иметь возможность объединить их в одну группу.

Таблица 3.4 – Атрибуты сущности «stable\_waste»

|  |  |
| --- | --- |
| Имя поля | Формат поля |
| id | int |
| date | int |
| value | real |
| currencyId | int |
| isActive | int |
| name | text |

Например, при путешествии в другую страну пользователь будет тратить деньги и на топливо, и на питание, и на проживание – это все разные типы затрат, но он сможет объединить их под одним тегом, например, «отпуск» и позже узнать сколько именно он потратил денег на данную поездку. Данная таблица представляет из себя два поля, одно из которых уникальный идентификационный номер, второй – имя тега. Описание ее атрибутов приведено в таблице 3.5.

Таблица 3.5 – Атрибуты сущности «tags»

|  |  |
| --- | --- |
| Имя поля | Формат поля |
| id | int |
| name | text |

В данной сущности индекс: «id» – первичный уникальный ключ.

Сущность «type» содержит в себе информацию о типах затрат. Они имеют очень важную ценность, так как помогают распределить затраты пользователя по разным группам и далее иметь возможность показать ему затраты по каждой из групп. Данная таблица представляет из себя три поля, одно из которых уникальный идентификационный номер, второе – имя типа, третье – иконка типа. Описание ее атрибутов приведено в таблице 3.6.

Таблица 3.6 – Атрибуты сущности «tags»

|  |  |
| --- | --- |
| Имя поля | Формат поля |
| id | int |
| icon | text |
| name | text |

В данной сущности индекс: «id» – первичный уникальный ключ. На рисунке 3.10 представлена схема базы данных пользователя.

База данных всех пользователей содержит две таблицы. Проведем описание сущностей. Сущность «users» содержит в себе информацию о всех пользователях, сохраненных в системе, а также суммарные доход каждого за все предыдущие периоды.



Рисунок 3.10 – Схема базы данных пользователя

В данной таблице большое количество атрибутов. Поле «email» хранить в себе электронный адрес пользователя, данное поле очень важно для авторизации пользователя в системе, поэтому он должно быть уникальным и неповторяющимся, поле «name» – имя пользователя в системе, поле «last Value» содержит суммарный доход пользователя за все предыдущие месяцы, поле «lastDate» указывает за какой период был подсчитан суммарный доход пользователя, поле «password» содержит в себе пароль пользователя в зашифрованном виде, то есть даже при краже данных, никто не сможет расшифровать его, кроме самого пользователя поскольку кличем владеет только он. Описание данной сущности представлено в таблице 3.7.

В данной сущности индекс: «id» – первичный уникальный ключ.

Таблица 3.7 – Атрибуты сущности «users»

|  |  |
| --- | --- |
| Имя поля | Формат поля |
| id | int |
| name | text |
| email | text |
| password | text |
| last Value | int |
| lastDate | int |

Сущность «sessions» содержит в себе информацию о всех сессиях пользователей. В данной таблице большое количество атрибутов. Поле «use rid» хранит идентификационный номер пользователя, поле «session» – имя сессии, поле «date» хранит дату открытия данной сессии, таким образом можно всегда будет узнать устарели он или нет. Описание данной сущности представлено в таблице 3.8.

Таблица 3.8 – Атрибуты сущности «sessions»

|  |  |
| --- | --- |
| Имя поля | Формат поля |
| id | int |
| userId | int |
| session | text |

В данной сущности индекс: «id» – первичный уникальный ключ, а поле «userId» является вторичным кличем к таблице «users». На рисунке 3.11 представлена схема общей базы данных.

Данных таблиц достаточно для полноценной работы приложения. Теперь имеет смысл написать модуль для работы с базами данных, а так как их будет несколько, то нужен еще один модуль, который будет контролировать подключения к каждой из баз данных, а также будет отключать ненужные, так как держать подключения к базам данных, которые уже давно не использовались пользователем не имеет смысла.

Модуль для работы с базой данных должен выполнять следующие функции:

* хранить подключенные базы данных;
* выдавать необходимый модуль для работы с базой данных по запросу;
* находить путь для подключения к базе даны;
* создавать базу данных, при необходимости;
* создавать подключения к новым базам данных;
* удалять устаревшие подключения к базе данных.

Начать стоит именно с создания подключения к базе данных, так как для этого понадобиться новый суб-модуль, который будет общим для всех подключаемых баз данных.

В функции данного модуля должны входить:

* подключение к базе данных, если оно не создано;
* отключение соединения с базой данных;
* выполнение передаваемых в модуль запросов к базе данных.



Рисунок 3.11 – Схема общей базы данных

Из описанного выше может показаться, что данный модуль не такой большой, чтобы выносить его в отдельный модуль, однако в последующем расширении функциональности проекта, данный набор функций может быть расширен, что упростит работу с модулем.

Возвращаясь к модулю для управления базами данных нужно отметить, что для хранения активных подключений данному модулю понадобится хеш-таблица – это значительно ускорит поиск нужной базы данных и, в целом, проведению любой операций с ними.

Как только в модуль приходит запрос на получении модуля для работы с базой данных, он ищет необходимы модуль в хеш-таблице и, если находит, то повращает найденный модуль в вышестоящий модуль. Если же модуль не был найден, то делает попытку подключиться к новой базе данных. Если подключение удалось, то модуль заносит его хеш-таблицу и возвращает модуль в вышестоящий модуль. Если же подключение не удалось или произошла ошибка, то она также возвращается в вышестоящий модуль. На рисунке 3.12 представлена работа модуля по управлению базами данных.

Не стоит забывать про функцию отключения устаревших баз данных. Так как все открытые подключения храниться в хеш-таблице, там же имеет смысл хранить время последнего запроса к этой базе. На сервер нужно запустить бесконечный цикл, который будет каждые десять минут проверять какие базы за это время стали ненужными и, пользуюсь методом базы данных, закрывать соединение и удалять базу из хеш-таблицы. Данный метод позволит высвобождать память, что ускорит работу сервера. Для пользователя только первое обращение к базе будет занимать больше времени в связи с подключением к базе, однако дальнейшая работа будет проходить довольно-таки быстро.



Рисунок 3.12 – Работа модуля по управлению базами данных

## **3.8** Клиентский блок для работы с базой данных

Активное подключение к сети интернет не всегда возможно, особенно сейчас, когда большинство населения пользуется мобильными телефонами, сеть в которых не всегда бывает доступна. Как только пользователь заезжает в тоннель или подальше от города сеть начинает пропадать. Именно поэтому в программе должна быть возможность сохранить данные на локальном устройстве а затем, при восстановлении соединения с сетью, синхронизировать данные.

Для работы данного функционала нужен модуль, которые будет работать с локальным хранилищем устройства пользователя. В данном проекте имеет смысл использовать Local Storage.

LocalStorage – это локальное хранилище, то есть специально отведенное место в браузере, где можно записывать, читать и удалять какие-либо данные. На самом деле, локальное хранилище очень похоже на cookie, но есть отличия. Cookie очень ограничены: в одной cookie может быть всего 4096 символов, а их количество на один домен примерно от тридцати до пятидесяти в зависимости от браузера. В локальном хранилище можно хранить от 5 до 10 мегабайт или даже больше на длительное время, но в большинстве браузеров размер составляет 5 мегабайт. Это места вопле достаточно, чтобы хранить пользовательские данные до стабильного подключения к сети.

Самое большое отличие cookie от LocalStorage - это то, что первый работает с сервером, а второй нет. Использовать локальное хранилище нужно там, где не нужна работа с сервером, а нужно хранить какие-то временные данные.

С данным модулем работать довольно-таки просто, большая часть его функций включена в работу браузера. Однако его все равно нужно обернуть в модуль для того, чтобы делать записи в отдельно отведенную ячейка памяти браузера. Это значительно упростит работу разработчика, так как ему не нужно будет думать в какую ячейку записывать данные, все его внимание будет сосредоточено на данных, которые нужно сохранить.

Также необходимо учесть, что бесконечно хранить данные пользователя нельзя, так как пользователь может выйти из аккаунта и его несинхронизированные данные могут достаться другому пользователю. Следует предусмотреть вариант оповещения пользователя о возможной утере несинхронизированных данных при выходе из аккаунта.

## **3.9** Блок работы с распознаванием речи

Ввод данных играет важную роль в жизни приложения. Данный процесс должен быть максимально удобен для пользователя, так как это основная функция приложения – сбор данных пользователя. Но бывают такие ситуации, когда времени на ручной ввод информации нет, а данные ввести нужно быстро, или пользователь в данный момент ведет машину и хочет внести затраты на топливо, которое он только что заправил. В данном случае на помощь приходит технология распознавания речи.

Распознавание речи – самая интересная и сложная задача искусственного интеллекта. В данной задаче задействовано множество достижений из большого количества различных областей, начиная от компьютерной лингвистики и заканчивая цифровой обработкой сигналов. Если же посмотреть на эти сигналы, то там нельзя увидеть ни слов, ни четких в выражении фонем, так как разные «речевые события», как их называют, перетекают друга в друга, при этом не образуя четких границ. Так, например, одна и тоже фраза, сказанная разными людьми и в различной обстановке, будет выглядеть каждый раз по-разному. Но люди же как-то распознают речь и понимают друг друга, а значит существуют инварианты, исходя из которых по сигналу можно восстановить, что именно было сказано. Именно поиском данных инвариантов является основной целью акустического моделирования.

В данный момент на рынке представлено много сервисов по распознаванию голоса, но так как основной акцент приложения будет делаться на русскоязычную аудиторию, то имеет смысл рассмотреть распознавание речи от компании Яндекс «Yandex.SpeechKit».

Процесс установки данной подпрограммы в проект не составляет трудности, остается лишь подключить библиотеку к проекту и пользоваться услугами сервиса. Однако, сервис предоставляет услуги лишь по распознаванию речи, и для того, чтобы приложение могло понять, что от него хотят, нужно разработать новый модуль, который будет расшифровывать какую именно команду произнес пользователь и в зависимости от нее, будет выполнять заложенные в него функции. Для этого необходимо будет добавить минимальные набор команд, которые будет понимать приложение.

**4** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## **4.1** Разработка сервера приложения

Разработка начинается с сервера приложения, потому как разрабатывать клиентское приложении без минимально функционирующей серверной части не имеет смысла.

Первая и самая основная цель серверной стороны – это запускать сервер. Так как сервер будет работать на платформе NodeJS, то самый удобный и безопасный способ работать с сервером – это подключить библиотеку ExpressJS. Это уже давно зарекомендовавший себя модуль, который имеет большой и гибкий набор конфигураций.

Необходимо добавить данную библиотеку в проект с помощью NPM-менеджера. Для этого нужно использовать команду: «npm install express --save». Процесс установки модуля отражен на рисунке 4.1. После установки нужно подключить модуль к основному файлу сервера и произвести пробный запуск сервера на 1337 порту. По умолчанию, сервер будет запущен на локальном адресе компьютера без каких-либо дополнительных настроек. Пробный запуск сервера представлен на рисунке 3.2.

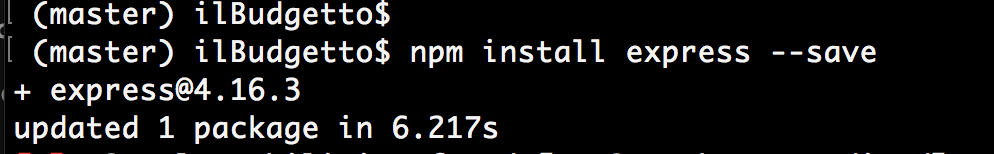


Рисунок 4.1 – Установка библиотек для проекта

Имеет смысл вынести конфигурацию и настройки сервера в отдельный файл. Это упростит работу с проектом и дальнейшее ориентирование в файлах проекта. Сервер должен быть достаточно безопасным, так как будет хранить данные бюджета пользователей. Здесь будет использоваться подход, при котором к серверу будет открыто как можно меньше путей. Именно поэтому в конфигурации сервера нужно закрыть доступ ко всем скрытым файлам, начинающимся с точки. Также в конфигурации следует указать, чтобы по умолчанию открывались только файлы с расширением html и не разрешалась никакая переадресация. На рисунке 4.2 представлен модуль конфигурации веб-сервера проекта.

Сервер будет содержать и статический данные, такие как скрипты, файлы стилей и скриптов. Имеет смысл перенести статические файлы в одну папку и запасть сервер для работы именно с ней в качестве корневого каталога. Таким образом доступ к серверу будет строго ограничен статическими файлами.

Для обмена данными между клиентом и сервером будет использоваться формат JSON – это текстовый формат обмена данными, основанный на JavaScript. По умолчанию данный модуль не включен в конфигурацию сервера, поэтому его необходимо добавить через NPM-менеджер, а затем добавить в конфигурацию. На этом конфигурация сервера для работы со статическими данными закончена.



Рисунок 4.2 – Конфигурация веб-сервера



Рисунок 4.3 – Создание объекта сервера с полной конфигурацией

Далее необходимо создать модуль, который будет создавать объект сервера с необходимыми параметрами.

Для этого создается отдельный модуль, который подключает к себе библиотеку «ExpressJS», модуль для работы с JSON-форматом, а также конфигурацию статического сервера. Полученный объект экспортируется в вышестоящий модуль для запуска сервера. Полная версия модуля представлена на рисунке 4.4.

Теперь следует приступить в запуску сервера. Для этого необходимо создать новый модуль. Так как основной объект сервера уже создан, его нужно только подключить, а также подключить стандартные библиотеки NodeJS для создания серверов с помощью протокола HTTP, а также защищенного соединения HTTPS, которое будет шифровать данные.

HTTPS является расширением HTTP протокола и создан для поддержки шифрования соединения с целью повышения безопасности. В данном протоколе данные передаются поверх криптографических протоколов TLS и SSL. Для HTTPS по умолчанию используется TCP-порт 443, в отличие от HTTP, в котором используется TCP-порт 8080.

HTTPS не относится к отдельному виду протокола. Это все тоже протокол HTTP, который работает через шифрованные транспортные механизмы TLS и SSL и обеспечивает защиту от разного рода атак: от сниф атак и атак типа «man-in-the-middle», при том условии, что будет использоваться шифрующее средство, и что сертификат, установленный на сервере, проверен и ему доверяют.



Рисунок 4.4 – Создание объекта сервера с полной конфигурацией

Перед непосредственным запуском сервера нужно создать сертификаты для работы для работы сервер под HTTPS. На рисунке 4.5 представлена команда для создания локальных сертификатов.



Рисунок 4.5 – Создание локального сертификата

После создания сертификатов их нужно внести в конфигурацию сервера и, воспользовавшись стандартными модулями, запустить сразу два сервера:

* HTTP-сервер с портом 8080;
* HTTPS-сервер с портом 8433, с подключенными к нему сертификатами шифрования.

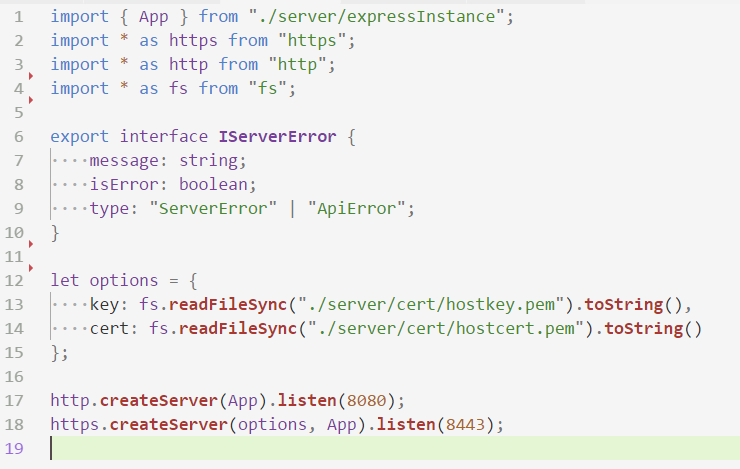


Рисунок 4.6 – Модуль для запуска сервера

Два сервера необходимы для того, чтобы можно было пользователя, который случайно зашел не незащищенное соединение, переадресовать на зашифрованное соединение. Полная версия модуля для запуска сервера представлена на рисунке 4.6.

## **4.2** Разработка клиентской части приложения

Основной модуль клиентской части приложения будет содержаться в JavaScript файле, его нужно загрузить в HTML файл.

HTML файл будет статически загружаться с сервера, автоматически загружать статические стили и основной JavaScript файл, после чего начнется построение всех оставшихся частей приложения.

В HTML файлe указываются пути для загрузки скриптов и стилей, также указываются мета-теги для оптимизации страницы браузера под мобильные устройства. В дальнейшем приложение будет обернуто в приложение Cordova, следовательно функции зумирования страницы должны быть отключены. Все эти и другие элементы указываются в шапке, там же подключается файл стилей.

Полная конфигурация шапки представлена на рисунке 4.7.



Рисунок 4.7 – Конфигурация шапки HTML-документа

В теле страницы описываются основные элементы страницы, на основании которых будут строиться остальные части программы. К данным элементам относиться:

* блок заголовка;
* блок меню;
* блок основных элементов;
* скрипт основного модуля клиентской части;
* скрипт библиотеки для распознавания речи.

Блок заголовка постоянно виден на экране и должен содержать только элементы которые всегда должны быть на виду. В данный момент это иконка для открывания и закрывания блока меню, а так же иконка голосового ввода команд.

Блок меню, как и блок заголовка, представляют из себя суб-модули. Все, что нужно блоку меню для работы – это список страниц и ссылок на них, а также элемент меню, который находится в статическом HTML файле.

На основании полученных данных, строится список всех элементов меню, после чего, он добавляется к элементу меню. Полная конфигурация тела HTML-документа представлена на рисунке 4.8.

После создания основного HTML-документа следует переходить к создания основного скрипта на клиентской стороне. Данный скрипт, в первую очередь, должен провести полную проверку браузера на соответствие требованиям данного приложения.

Для этого следует создать отдельный модуль, который будет содержать все необходимые проверки функций, которым должен соответствовать браузер.



Рисунок 4.8 – Конфигурация основных элементов HTML-документа

Если все проверки пройдены, то построение страниц проекта может быть продолжено, если же нет пользователю выводиться сообщение, что версия его браузера устарела и для корректной работы приложения требуется установить более новую версию браузера.

После проверки на возможность работы программы в конструкторе основного модуля сначала находятся все основные элементы страницы и на основании их строиться объекты основных модулей, таких как меню и шапка приложения.

В меню передаются основные названия страниц и ссылки на них, а в шапку – основной элемент и обработчик события для открытия и закрытия меню. На рисунке 4.9 представлена инициализация основных компонент.



Рисунок 4.9 – Инициализация основных элементов программы

После инициализации основных элементов уже можно запускать приложение. Дальнейшая разработка приложения обязательно должна сопровождаться просмотром результатов работы. На рисунке 4.10 представлена стартовая страница приложения.



Рисунок 4.10 – Инициализация основных элементов программы

Как видно из рисунка 4.9 макет страницы открывается при запущенном сервере и следует приступать к разработки следующих модулей.

Основными компонентами, через которые будет проходить основная логика, и от работы которых будет зависеть работа всего приложения – это модуль для работы с адресной строкой браузера, в которой будет храниться состояние приложения, и модуль по загрузке страниц приложения.

Модуль для работы имеет довольно большую функциональность. На рисунке 4.11 представлены диаграмма данного класса. Данный модуль включает в себя один суб-модуль который преобразует строку браузера в JSON-форма и обратно, этим занимается библиотека ReasonsJS.

Модуль использует только часть функциональности данной библиотеки. Большая часть функций имеет свое назначение и интегрирует работы со стандартным модулем браузера, который называется History.

Основная логика модуля – это ждать любого изменения адреса страницы. Для этого он подписан и слушает событие изменения адреса браузера. Как только состояние адреса меняется данный модуль преобразует строку адреса в нужный и известный вышестоящему модулю формат JSON и отправляет его на дальнейшую обработку.

Логика может происходить и в обратному порядке, когда приложению нужно срочно изменить состояние на новое, если, допустим пользователь ввел неверный адрес или перешел на элемент страницы, который был удален. В данном случае формируется новое состояние страницы в формате JSON и передается в модуль, модуль в свою очередь преобразует новое состояние в строку и изменяет состояние адресной строки.

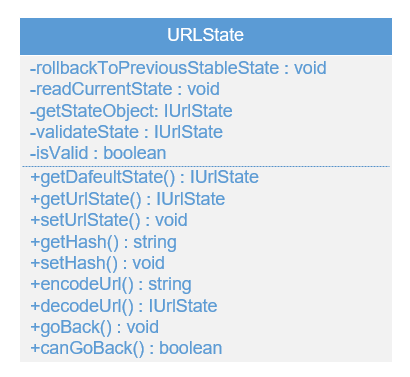


Рисунок 4.11 – Диаграмма класса Restate

Модуль также иметь методы для проверки состояния приложения на валидность для предотвращения некорректного состояния. На случай таких событий предположены методы для возврата в предыдущее состояние. Также модуль может эмулировать работу браузера, а именно перейти на обратную страницу.

Модуль для загрузки страниц имеет немаловажную функцию – он меняет страницы приложения в зависимости от состояния приложения. Модуль всегда имеет струнцу по умолчанию, на которую будет переходить при неправильном состоянии приложения. Также модуль содержит в себе объекты все страниц приложения – это хеш-таблица, которая заполняется по мере открытия пользователем новых страницы приложения. Не имеет смысла создавать объекты каждой страницы при открытии приложения, так как это будет занимать лишнюю память устройства.

Каждый раз когда происходи изменение состояния, модуль берет имя новой страницы и ищет его хеш-таблице. Если страница не была найдена, значит на нее заходят впервые, модуль инициализирует объект данной станицы и выполняет повторный поиск. Как только страница будет найдена, модуль проверяет не является ли данная страница активной, ведь возможен вариант, что пользователь уже находиться на данной станице, но он лишь поменял конфигурации данной страницы. Если страница та же самая, то все, что нужно сделать, это лишь вызвать метод «focus» у текущей страницы и передать в нее новое состояние. Если же новая страница не является текучей, то вызывается метод «blur» для текущей страницы, затем новая страницы становится текучей для данного модуля, и затем для уже новой страницы вызывается метод «focus» с новым состоянием.

## **4.3** Разработка модуля распознавания голосовых команд

В данный момент на рынке представлено много сервисов по распознаванию голоса, но так как основной акцент приложения будет делаться на русскоязычную аудиторию, то имеет смысл рассмотреть распознавание речи от компании Яндекс «Yandex.SpeechKit».

Процесс установки данной подпрограммы в проект не составляет трудности, остается лишь подключить библиотеку к проекту и пользоваться услугами сервиса.

Сервис предоставляет услуги лишь по распознаванию речи, и для того, чтобы приложение могло понять, что от него хотят, нужно разработать новый модуль, который будет расшифровывать какую именно команду произнес пользователь и в зависимости от нее, будет выполнять заложенные в него функции. Для этого необходимо будет добавить минимальные набор команд, которые будет понимать приложении.

После подключения библиотеки нужно зарегистрироваться в системе Яндекс, чтобы получить доступ к распознаванию – он представляет из себя ключ, который будет оправляться с запросами на распознавание.

Для вставки элемента в проект нужно создать элемент скаченной библиотеки, а в опции включить полученный ключ, а также подписаться на событие распознавания текста. Подробно подключение модуля изображено на рисунке 4.12.

Как было сказано выше, результатом распознавания будет текст, а значит нужен модуль, который будет разбивать распознанный текст на лексемы и узнавать какие команду пользователь хотел выполнить.

Так как чаще всего пользователь будет использовать данную функцию для внесения своих затрат, то следует на этом заострить внимание.

Для того, чтобы системы команд была гибкой, список главных лексем, означающих операцию ввода затрат был расширен до следующих слов:

* затрата;
* прибыль;
* плюс;
* минус;
* и все варианты данных слов.

Так как система работает с несколькими валютами, то для каждой были написаны свои обозначения, как символами, так и словами. Все это представляет из себя несколько регулярных выражений.

Как только на обработку приходит распознанный текст, первое, что проверяется – это наличие ключевых слова, что помогает понять, что должна быть использована команда для распознавания затраты.



Рисунок 4.12 – Работа с модулем распознавания речи

Первым, что необходимо распознать – это сумму затраты и валюту. Для этого вызывается метод, который ищет все числа с обозначениями валюты в строке. Затем метод проходит по каждой найденной строке, вычленяет из нее валюту и оставшеюся часть преобразует в число. Необходимо отметить, что при отсутствии числа или валюты метод вернет ошибку и распознавание прервется.

После нахождения суммы и валюты платежа подобным образом вычисляется была ли это затрата или же прибыль пользователя.

На каждом предыдущем этапе найденные элементы вырезаются из строки, поэтому на последнем этапе все, что осталось в строке можно считать комментарием к платежу.

На этом операция распознавания команды заканчивается, из распознанных данных формируется объект в формате JSON и отправляется в вышестоящие модуль для дальнейшей обработки.

**5** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

## **5.1** Назначение и описание компонент программы

Для реализации программного продукта использовались следующие инструменты:

* база данных SQLite;
* серверная часть на платформе NodeJS;
* клиентская часть на языках TypeScript, JavaScript, HTML и CSS;

Приложение реализовано на прототип но-ориентированном сценарном языке программирования TypeScript, который в последствии компилируется в язык JavaScript.

Разработанный программный продукт включает 17 основных TypeScript файлов:

* ilBudgetto.ts – это блок ядра клиентского приложения, который организует работу всех суб-модулей;
* menu.ts предназначен для создания меню;
* header.ts предназначен для создания шапки программы и расположению на ней основных элементов;
* urlState.ts предназначен для работы с адресной строкой браузера;
* pageLoader.ts предназначен для переключения между страницами приложения;
* chart.ts предназначен для построения графиков;
* clientApi.ts предназначен для создания API-запросов к серверной стороне;
* dateParser.ts предназначен для конвертации локальной даты пользователя в формат серверной даты, также содержит функции для работы с датами;
* entityLoader.ts предназначен для сосредоточения всех видом API-запросов в одном месте для простоты использования;
* errors.ts содержит все типы ошибок, которые используются в проекте;
* supported.ts содержит набор функций для проверки наличия необходимых функций в браузере;
* pages.ts содержит информацию о всех доступных приложения страницах;
* server.ts служит для запуска сервера;
* express Instance.ts служит для создания объекта сервера;
* dnInstance.ts служит для создания объекта для работы с базой данных;
* dbManager.ts служит для работы с объектами баз данных;
* api.ts содержит методы для работы с API.

Также в проекте присутствует большое количество файлом со стилями, они представлены в виде фалов с расширением LESS, языка динамических стилей. После полного описания стилей они компилируются в один файл CSS, который и подключается к программе.

При создании нового пользователя для него автоматически создается его собственная база данных, с которой будет работать только он, она храниться в отдельном файле и никак не может быть затронута другими пользователями.

## **5.2** Тестирование программного продукта

В данном разделе приведено описание последнего этапа тестирования разработанного программного обеспечения.

Всем известно, что тестирование очень важный процесс в разработке приложений. Без тестирования программные дефекты и не состыковки оставались бы в программах и с каждым разом ухудшали их работу, а компании несли бы крупные потери и тратили больше количество времени на поиски и устранение неполадок. После выпуска стабильной версии программного продукта исправлять ошибки сложнее, и, зачастую, это обходится довольно дорого. Процесс автоматизации тестирования в большей степени увеличивает покрытие кода и, как привило, в перспективе уменьшает будущие расходы разработчика. Чаще всего разработчики сами создают автоматизированные тесты, но в последнее время этим занимаются специально обученные татуировщики, автоматизаторы, как их еще называют. Здесь может быть как модульное тестирование – это короткие тесты каких-либо частей функционала программы, либо более масштабное тестирование, оно осуществляется на уровне интеграции.

Так или иначе, поиск инструментов для автоматизации тестирования на платформе NodeJS может быть непростой. Существует ряд библиотек для осуществления процесса тестирования, вот некоторые из них:

* Mocha;
* Chai;
* Mockery;
* Jenkins.

Принцип работы каждой из библиотек похожий как процесс установки библиотеки в проект, он состоит из нескольких этапов:

* добавить в проект библиотеку с помощью NPM-менеджера;
* перенастроить конфигурацию проекта для работы с библиотекой;
* добавить тестовый скрипт;
* написать тесты;
* выполнить тесты.

Процесс добавления библиотек к проекту уже описывался ранее, поэтому его можно пропустить. Для перенастройки конфигурации проекта нужно добавить команду для запуска теста в конфигурационный файл, как это показано на рисунке 5.1.

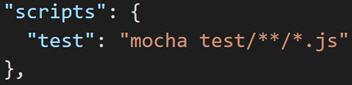


Рисунок 5.1 – Подключение модуля тестирования к проекту

После этого необходимо создать пару тестовых скриптов, в которых будут находиться тесты.

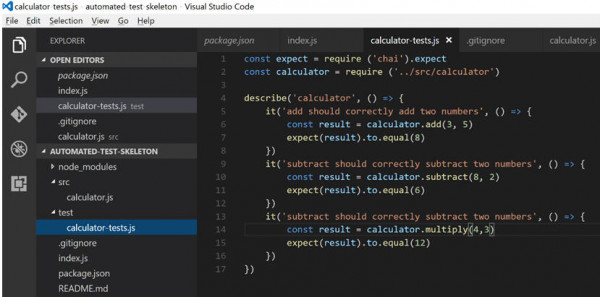


Рисунок 5.2 – Пример теста в библиотеке Chai

После того как все тесты были написаны, с помочью консоли можно их запустить, делается это командой «npm run test». Пример результата выполнения данной команды представлен на рисунке 5.3.

Теперь можно приступить к тестированию проекта, но для начала стоит выяснить, что нужно тестировать. Пользователь всегда пользоваться рабочим приложением, значит ему необходимо, чтобы основной набор задач приложения всегда работа. Основными задачами приложения являются:

* ввод данных пользователя;
* анализ данных пользователя;
* предоставление отчетов в виде графиков;
* распознавание речевых команд для ввода данных;
* изменение данных;
* просмотр данных.

Если пройти по данному списку задач, то можно смело сказать, что многие из них невозможно будет протестировать с помочью написания тестов. Для них нужно проводить мануальное или, как его еще называют, ручное тестирование.

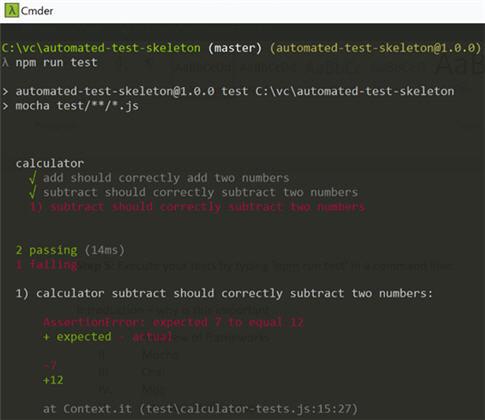


Рисунок 5.3 – Результат выполнения тестов

В ходе работы над проектом были написаны написаны автоматизированные тесты для проверки модулей, которые не связаны с пользовательским интерфейсов.

Каждый метод вызова API был проверен на выполнение поставленной задачи, а также на возвращение ошибок при некорректном вызове метода. Также на серверной стороне проведена проверка модулей для работы с базами данных: подключение новой базы данных, закрытие неактивного соединения с базой данных, попытка подключения к несуществующей базе данных.

Для корректной работы серверной и клиентских частей была выполнена проверка модели по конвертации клиентского времени в серверное и обратно. В результате тестирования ошибок не было обнаружено.

На клиентской стороне также есть модули, которые можно протестировать с помочью автоматизированного тестирования. Так была выполнена проверка модуля для отправления запросов к серверу, а также протестирован модуль, объединяющий все возможные вызовы к API. В результате тестирования была выявлена опечатка, которая запрашивала данные из другой таблицы. Неисправность была исправлена, что еще раз доказало плюсы написанию тестов.

Был протестирован модуль для работы с адресной строкой браузер в результате чего была устранена ошибка с неправильным возвратом к предыдущему состоянию приложения при неправильном новом состоянии.

Для пользовательского интерфейса было проведено ручное тестирование. Была проверена работа страниц по вводу информации и ее редактированию. Также были протестированы все возможные отчеты, из изменения при внесении новых затрат и прибылей. Также были проверены страницы для внесения стабильных затрат и прибыли пользователя – проблемы не были обнаружены.

Немаловажную часть заняло тестирование проекта на разных размерах устройств, так как интерфейсы пользователя могут отличаться в данных видах экранов. В процессе тестирования было принято решение о скрытии панели меню от пользователя на маленьких экранах. На больших экранах меню всегда будет видно.

В результате проведенного тестирования было выявлено несколько неисправностей, которые были исправлены. После исправления приложение работает стабильно и отвечает все заявленным нормам.

**6** РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

## **6.1** Требования к аппаратному и программному обеспечению

Для конфигурации сервера и его запуска необходимо будет установить на компьютер платформу NodeJS. Данная платформа для все видом операционных систем, что делает ее удобной для разработки продуктов. Скачать данную платформу поискав в интернете, либо по прямой можно скачать по прямой ссылке: «https://nodejs.org/en/download/». Выбирать нужно стабильную версию, которая указана слева, как на картинке 6.1. На момент написания проекта стабильная версия имела номер 8.11.2.

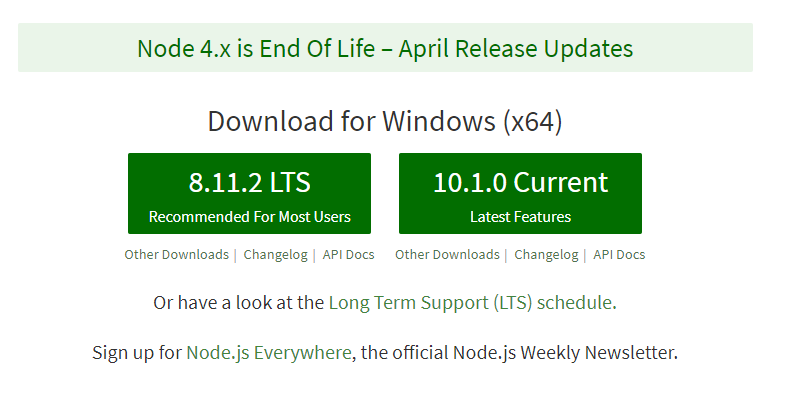


Рисунок 6.1 – Скачивание NodeJS

После скачивания дистрибутива, нужно открыть скаченный файл и проследовать указаниям мастера по установке программы.

Следующим этапом будет скачивание необходимых модулей для проекта. Необходимо открыто консоль и перейти в папку проекта. В папке проекту уже находить файл конфигурации, поэтому нужно будет только скачать нужные библиотеки. Для экономии места все библиотеки не включаются в проект, так как эта необходимость нужна лишь при его разработке.

Для загрузки нужных библиотек в корне открытого каталога нужна вызвать команду: «npm install». Данная команда откроет файл конфигурации и скачает нужные библиотеки. После выполнения команды на окно консоли будет выведен список установленных пакетов и надпись об окончании установки, как показано на рисунке 6.2.

Следующим этапом будет компиляция скриптов для работы программы. По умолчанию файлы скриптов хранятся на языке TypeScript и для компиляции из в JavaScript нужно перейти в папку «server» и выполнить команду: «ts -w». После компиляции будет выведено сообщение о удачном выполнении операции, как это показано на рисунке 6.3.

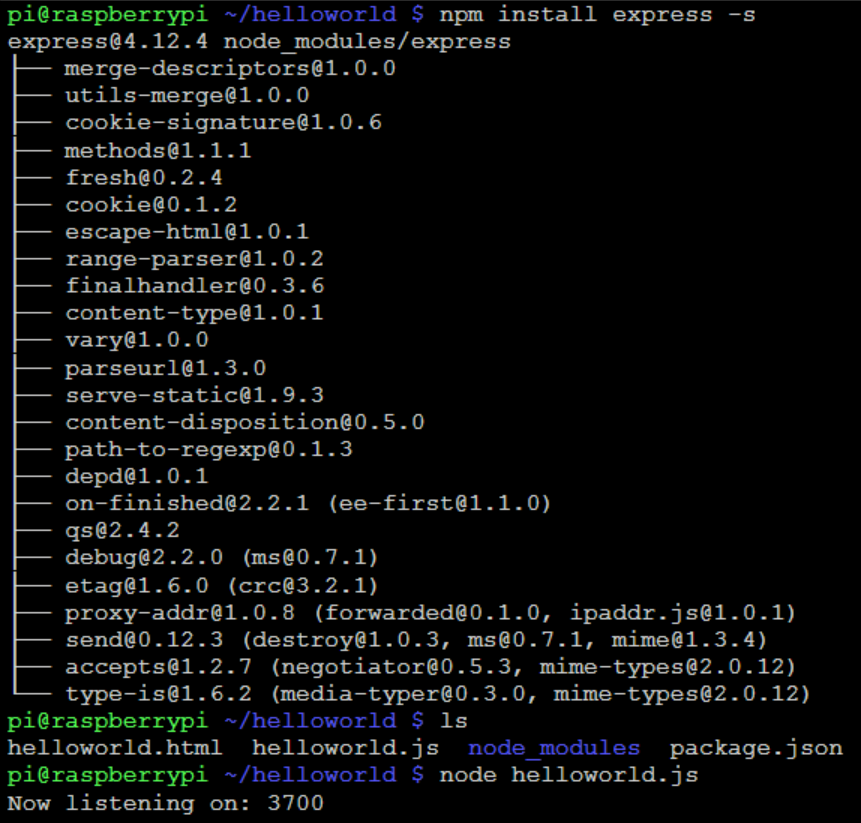


Рисунок 6.2 – Установка пакетов проекта

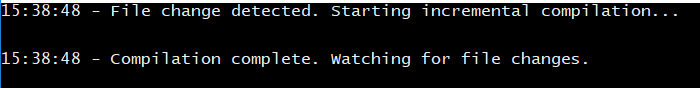


Рисунок 6.3 – Компиляция из языка TypeScript в JavaScript

Теперь можно вернуться в корневой каталог и собрать нужные файлы проекта. Для этого в коренном каталоге нужно вызвать команду: «gulp less» –данная команда выполнит компиляцию стилей из языка LESS в CSS. После чего нужно выполнить команду: «gulp bundle», которая соберет все файлы необходимые для работы в один JavaScript-файл и положит в нужную папку.

Для запуска сервера нужно перейти в каталог «server» и выполнить команду: «node server.jess».

После выполнения данной команды должно появиться сообщение об успешном запуске сервера и подключении к основной базе данных. Сообщение об удачном старте показано на картинке 6.4.

Необходимо отметить, что для работы программы также есть некоторые ограничения. Программа может работать самых последних версиях интернет-браузеров, таких как Microsoft Edge 16, Google Chrome 64, Firefox 60, Safari 11.

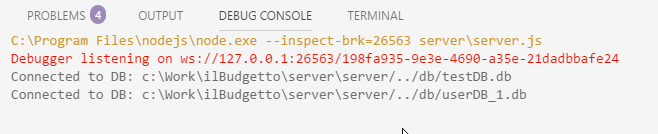


Рисунок 6.4 – Успешный запуск сервера

Для мобильных устройств поддерживаются такие платформы как Android 7 и iOS 11. На данном этапе алгоритм установки и запуска сервера окончен.

## **6.2** Руководство по эксплуатации

После запуска сервера нужно зайти на локальный адрес компьютера использую порт 8443 и протокол HTTPS, как это показано на рисунке 6.5.

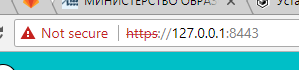


Рисунок 6.5 – Адрес приложения

При заходе пользователя встречает главная страница приложения. Необходимо отметить, что на большом и малом экранах она будет отличаться. На большом экране (рисунок 6.6) меню не будет преграждать страница, а на малом (рисунок 6.7) оно будет видно.

На главное странице отображена текучая дата, текущий полный счет пользователя, то есть все деньги, которые у него есть на данные момент.

Также на экране отображено количество денег доступных ему на сегодня, если он хочет не выйти в минус в конце месяца, при условии, что каждый день он будет соблюдать данное правило.

Ниже расположены элементы управления:

* список затрат за месяц;
* внесение затраты;
* внесение прибыли.

Кнопки для внесения затраты и прибыли приведут пользователя на одну и туже страницу, однако будут иметь разные значения в сумме платежа.

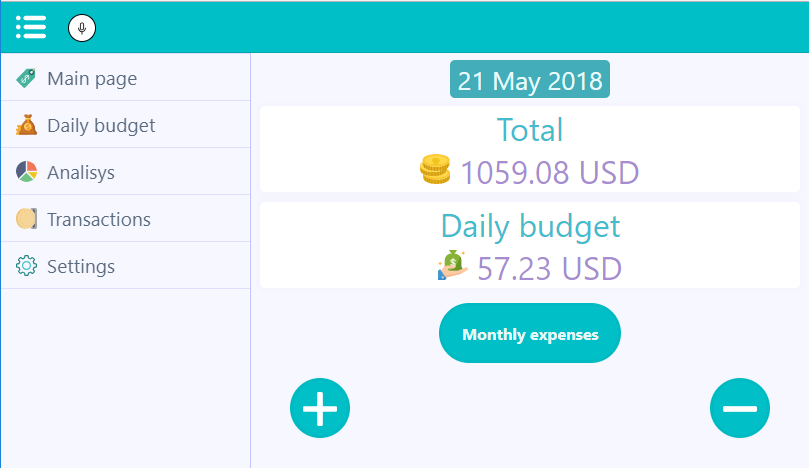


Рисунок 6.6 – Главная страница на большом экране

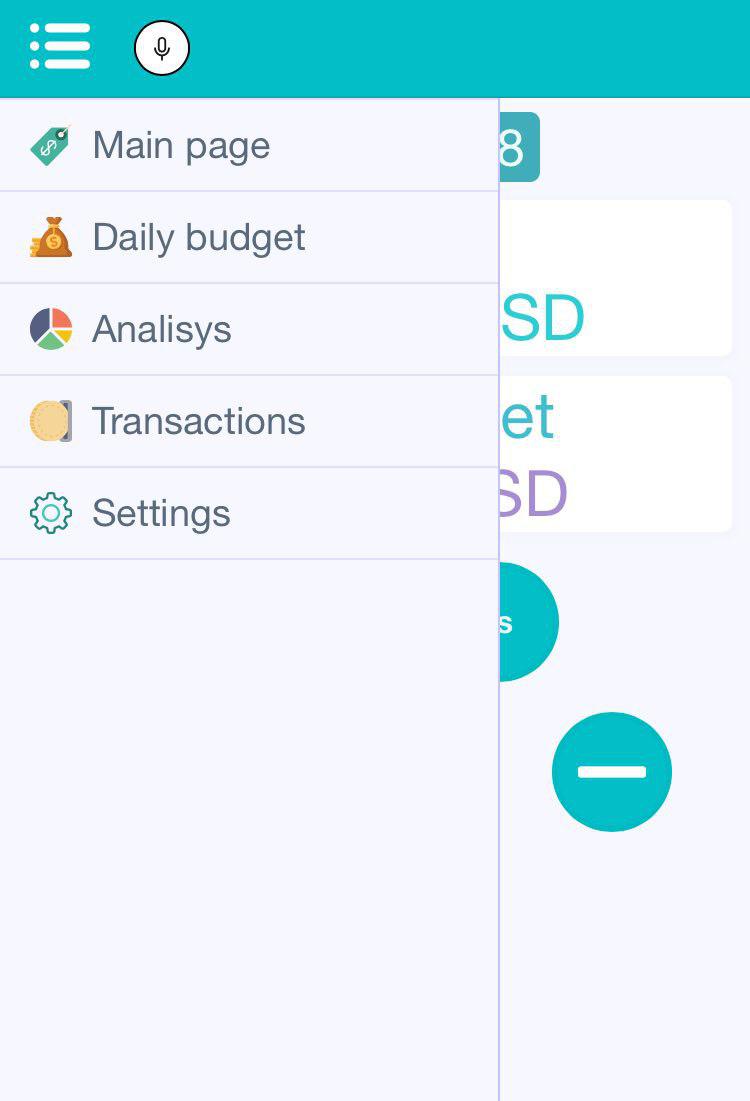


Рисунок 6.7 – Главная страница на маленьком экране

На страницу внесения затрат пользователь вносит всю необходимую информацию, пример представлен на рисунке 6.8.

Страница покупок за месяц (рисунок 6.9) содержит информацию о всех платежах совершенных пользователем за текущий месяц. Пользователь также может зайти и изменить любую запись и все данные будут мгновенно пересчитаны. Также список разделен на дни.

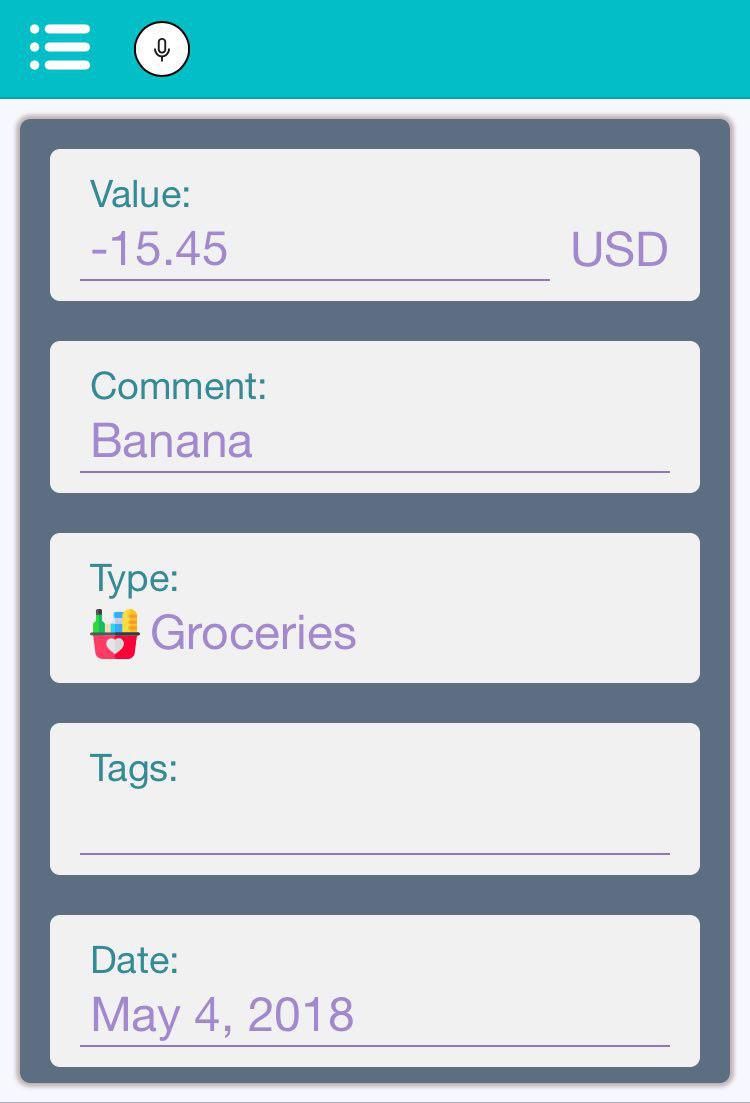


Рисунок 6.8 – Внесение затрат

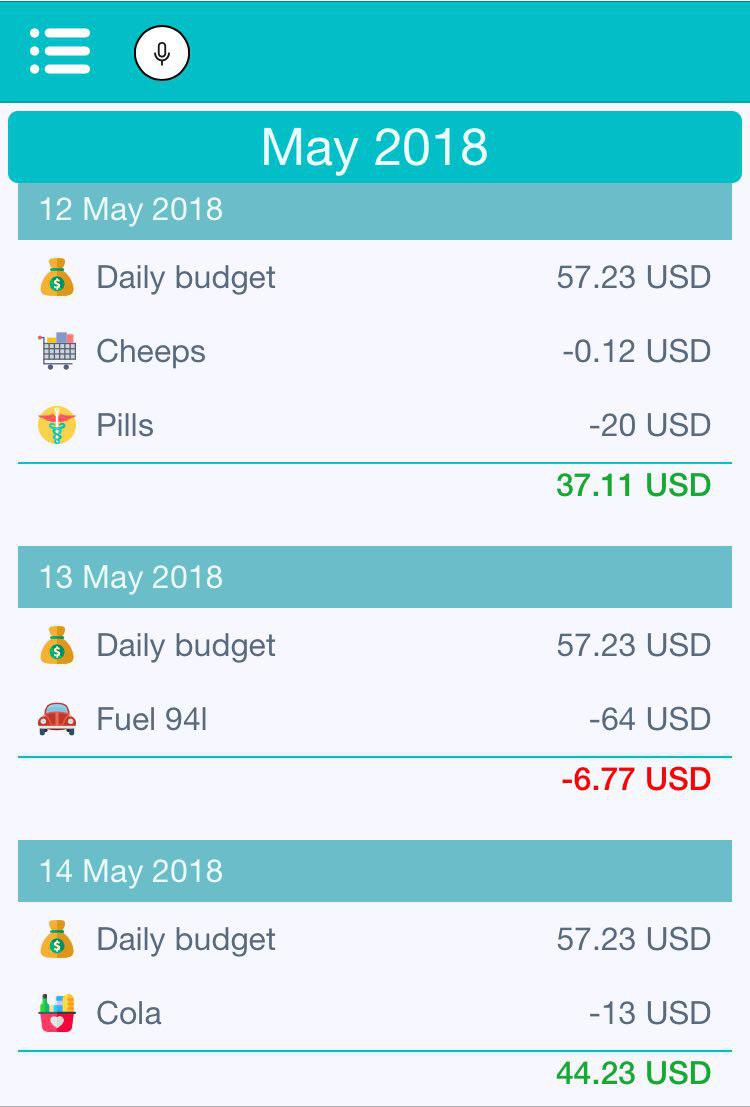


Рисунок 6.9 – Страница покупок за месяц

Для каждого дня расписан дневной бюджет и сколько осталось дневного бюджета по истечении дня. Положительный баланс подсвечен зеленым, а отрицательный – красным. Так пользователь сможет обратить свое внимание на большие затраты в течении дня, что будет стимулировать его делать меньше затрат.

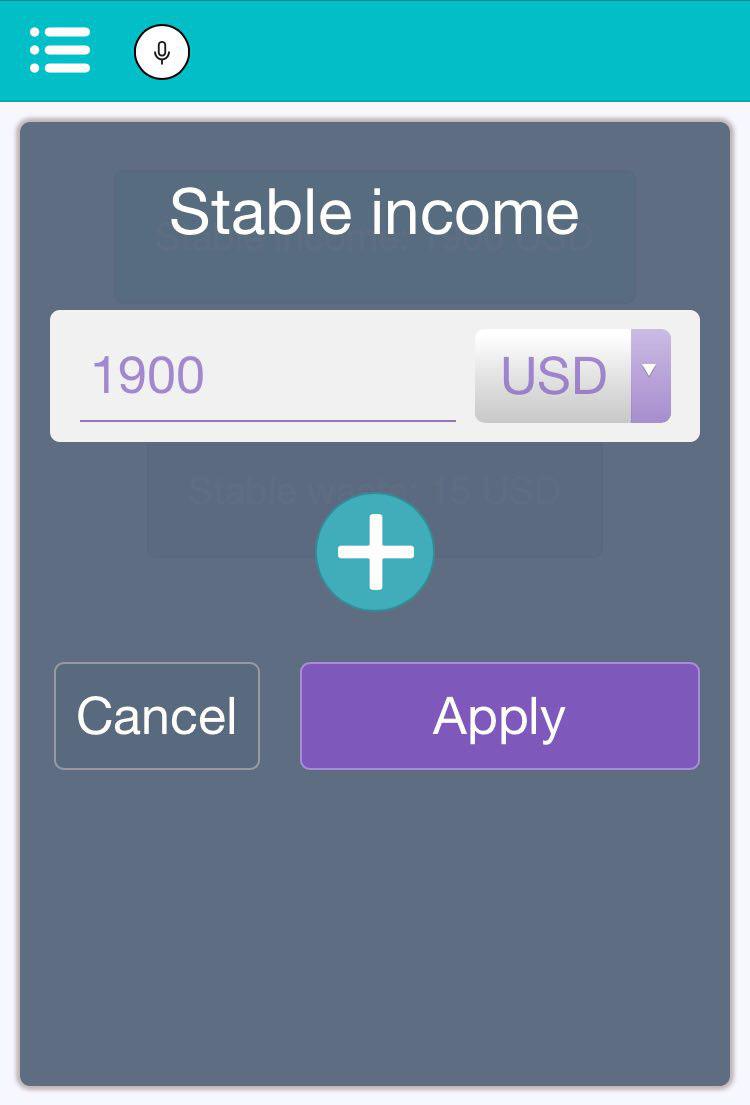


Рисунок 6.10 – Страница стабильных платежей

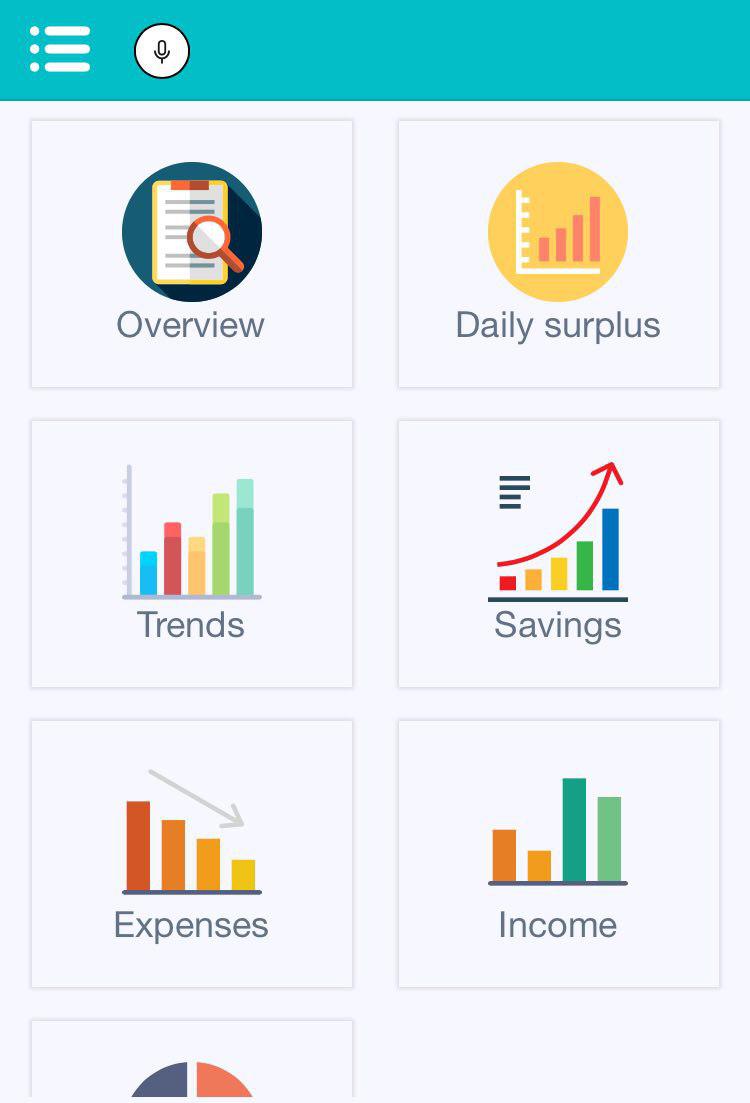


Рисунок 6.11 – Страница графиков и отчетов

Страница стабильных доходов и расходов (рисунок 6.10) предназначена для внесения стабильных платежей, например, заработная плата или плата за мобильный телефон.

Кроме внесения данных есть возможно просмотреть отчеты по затратам. Для этого существует страница с отчетами и графиками, она изображена на рисунке 6.11.

Существует график стабильных затрат, он показывает распределение всех стабильных затрат за текущий месяц, как это изображено на рисунке 6.12.

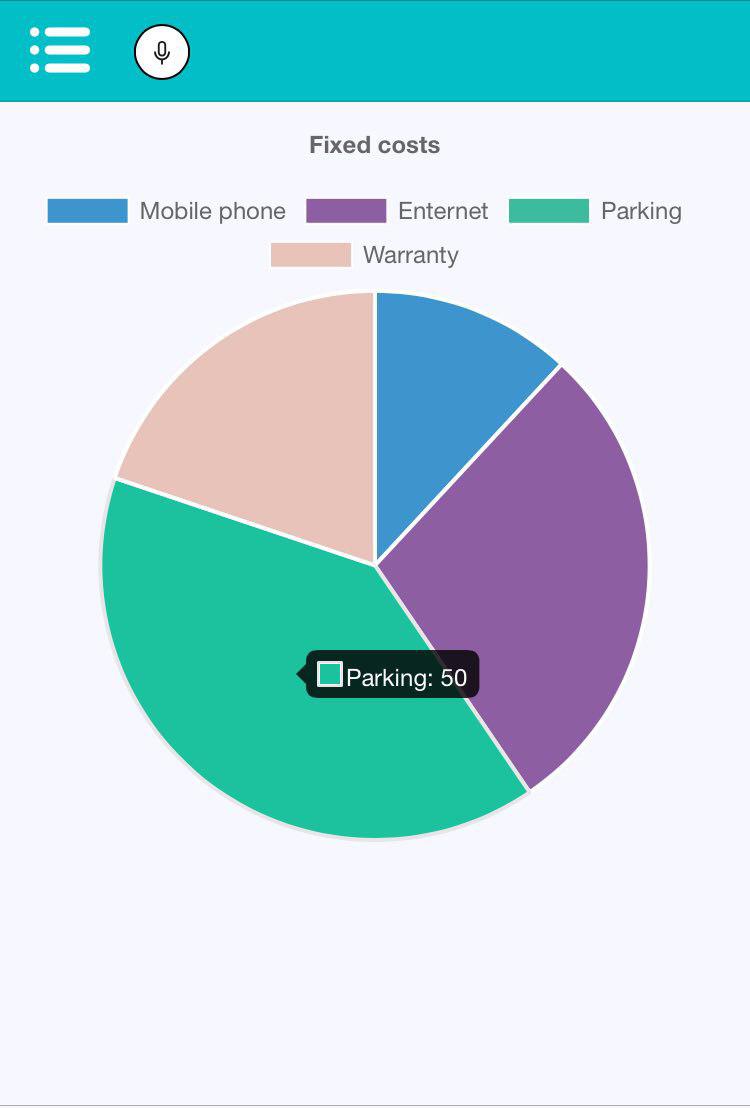


Рисунок 6.12 – График стабильных затрат

Данный график очень полезен и помогает определить самую большую стабильную затрату пользователя и должна призвать его к уменьшению данной затраты.

Немаловажным графиком является график затрат, который показывает какое количество денег пользователь потратил на каждый тип затрат. Это очень важно – знать как много денег уходит на пишу, обслуживание машины, топливо и прочее. Данный график изображен на рисунке 6.13 и отражает затраты по двум типам затрат.

Дневной бюджет также был не зря введен в приложение, так как позволяет точно вычислить дни с убыточным бюджетом (русинок 6.14).

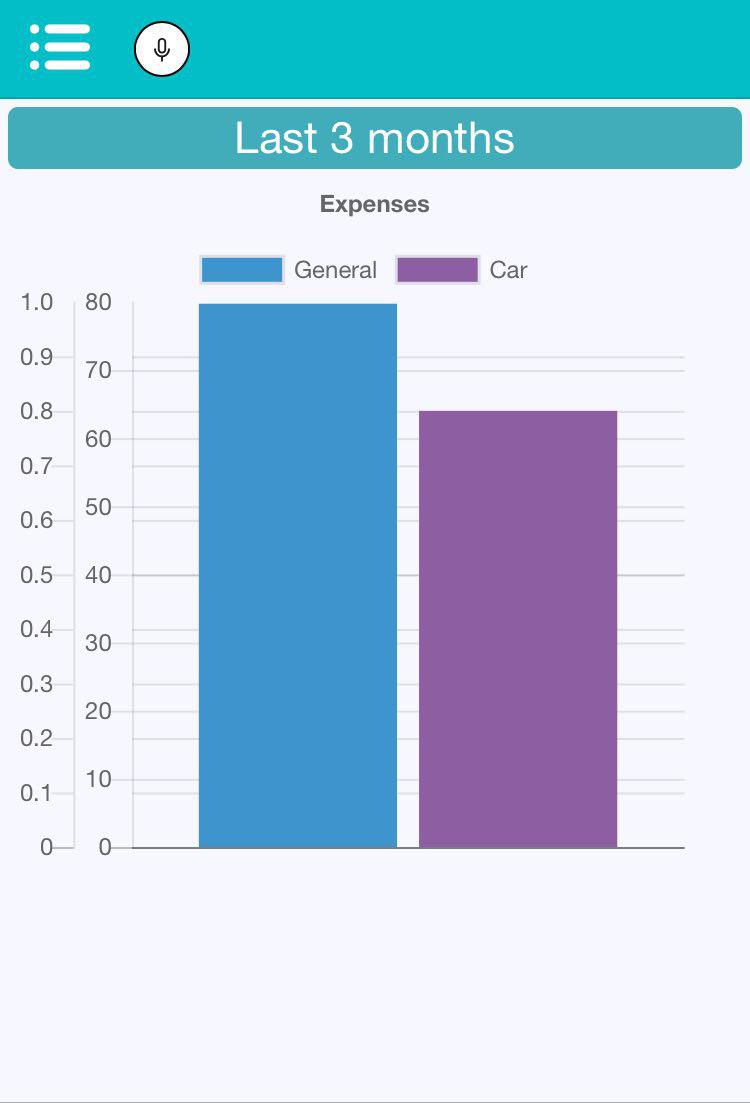


Рисунок 6.13 – График затрат по типам

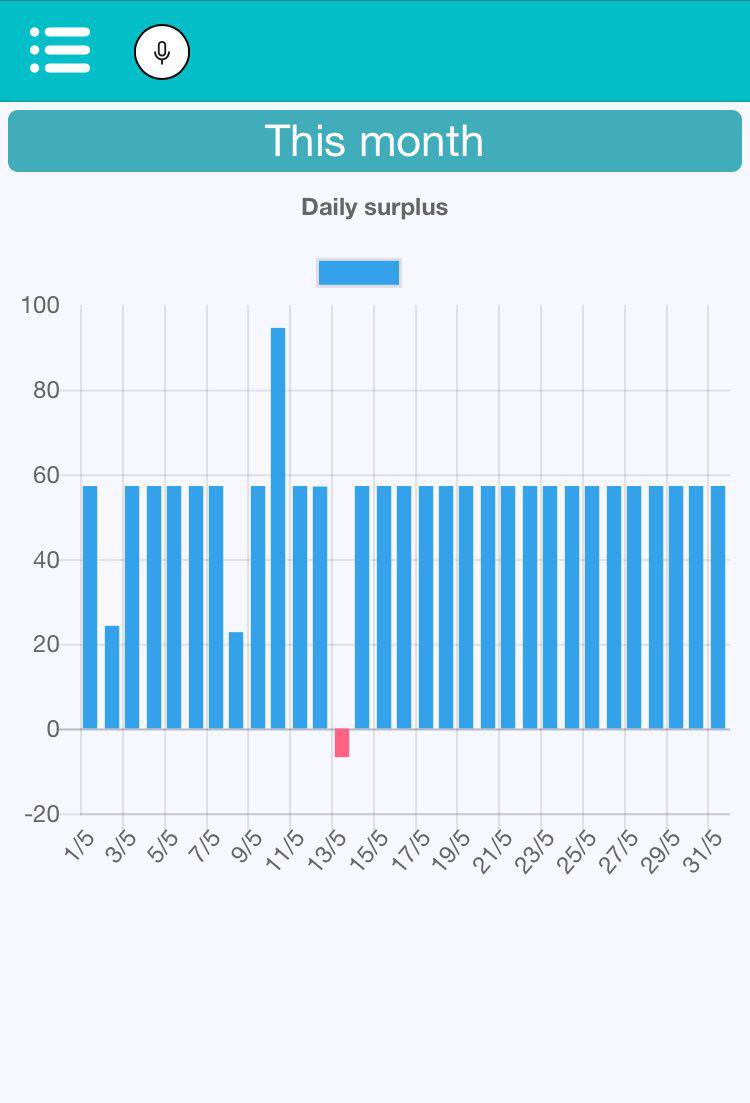


Рисунок 6.14 – График попаданий в бюджет

Человек всегда рад играть в игры и как это ни странно будет стремиться к уменьшению красных дней на графике.

Следующий график показывает прибыль пользователя за последние 3 месяца (рисунок 6.15), также на нем отображаются данные о дополнительной прибыли пользователя. Очень полезный график для просмотра доходов.



Рисунок 6.15 – График прибыли

За ним следует график, который объединяет все предыдущие графики и показывает соотношение прибыли пользователя к его стабильным затратам и фиксированным расходам. Таким образом можно наглядно увидеть какую часть дохода съедают затраты. Данные график является наиболее удобным для просмотра и пониманию картины в целом.

Благодаря разным цветам на графике, можно легко, не всматриваясь в цифры, посмотреть примерный процент затрат и остатка, а для более детальной информации при клике на график появляется вспомогательное окно с точными цифрами. Подробнее данный график можно изучить на рисунке 6.16.

Другим, не менее важным, графиком является график сбережения да текущий месяц, его можно увидеть на рисунке 6.17. Данный график стимулирует пользователя к минимизации своих затрат, так как показывает какой остаток денежных средств будет у пользователя при минимальных затратах до конца текущего месяца.



Рисунок 6.16 – График соотношения дохода и расхода

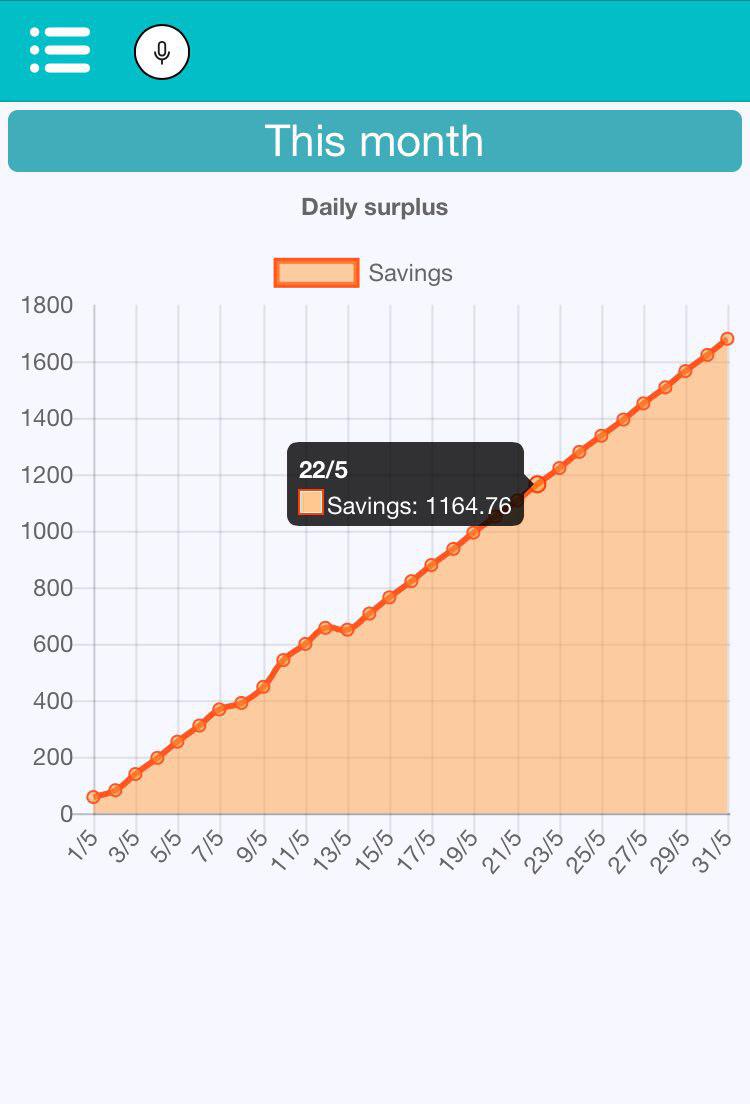


Рисунок 6.17 – График прогнозируемого остатка бюджета

Данный график отлично стимулирует. Он представлен в виде линейного возрастающего графика, если затраты пользователя не превосходят доходам. Смотря на дынный график пользователь будет стремиться сделать его как можно больше возрастающим, ведь все любят рост в любой области, особенно финансовой.

Есть пользователи, которые не любят смотреть на графики, тем более, когда они разбиты на разные страницы и нужно как-то соединять все это в голове. Для такого случая имеется страницы быстрого просмотра. На ней собраны все вышеперечисленные страницы, точнее там собраны все подсчеты, организованные на страницах с графиками. На рисунке 6.18 представлена данная страница со всеми показателями на текучий месяц.

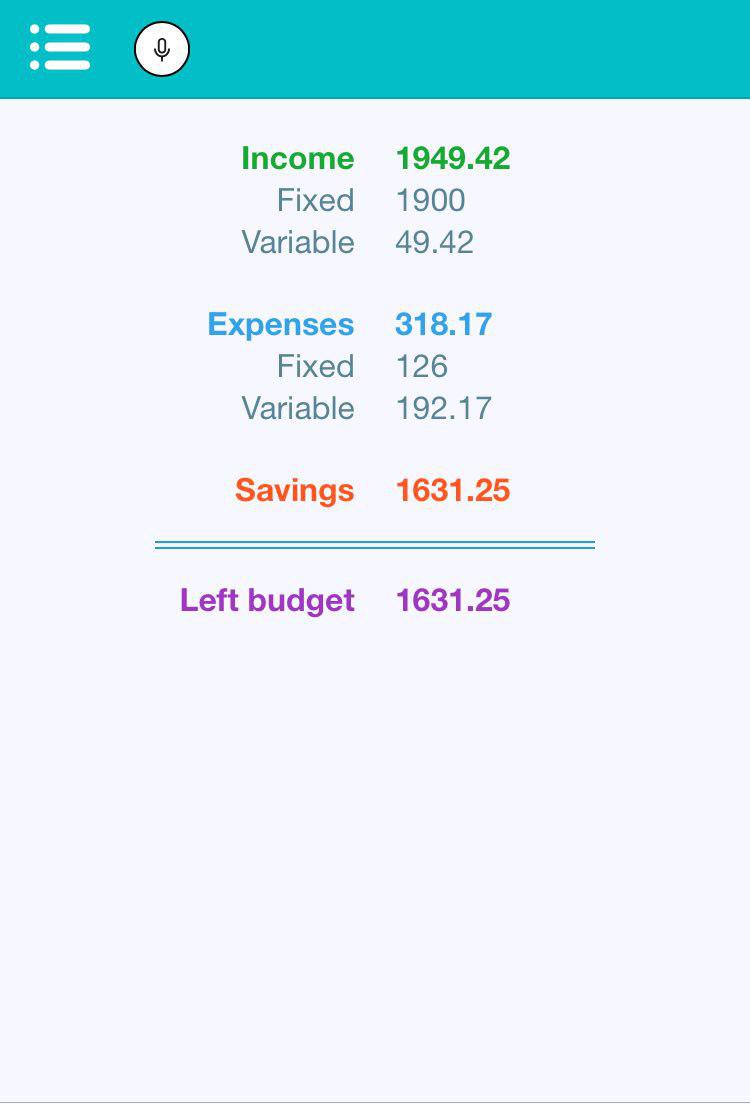


Рисунок 6.18 – Страница общей информации о затратах

Как можно видеть из рисунка все показатели разбиты по группам, а именно:

* прибыль;
* затраты;
* сбережения;
* оставшийся бюджет пользователя.

Таким образом можно легко угодить любому пользователю, каждый выберет свой вариант использования приложения.

Можно заметить, что в верхней панели приложения расположена иконка микрофона. Данная иконка отвечает за прием голосовых команд приложения. При нажатии на иконку она становиться желтой – это значит приложение слушает команду пользователя. Процесс заканчивается, когда приложение понимает, что пользователь произносить речь. Сказанное распознается и передается в модуль распознавания команды, после чего выполняется необходимая операция.

# 7 ЭКОНОМИЧЕСКАЯ ЧАСТЬ

**7.1** Характеристика приложения для ведения бюджета

В данном дипломном проекте разработан программный продукт, предназначенный для ведения бюджета.

Цель разработки программы – обеспечение более высокой производительности труда, большей надежности и достоверности информации, лучшей ее сохранности.

Актуальность темы дипломного проекта обуславливается необходимостью снижения временных и денежных затрат на выполнение стандартных рутинных операций, выполняемых человеком ежедневно.

Практическая значимость работы определяется разработкой реального программного средства, служащего для ведения бюджета как одним человеком в своих нуждах, так и начинающим свой бизнес, в качестве отправной точки для управления и контроля бюджета.

Ведение бюджета помогает решить ряд проблем:

* навести порядок в голове, относительно того, сколько человек зарабатывает и тратит, в каком направлении расходуется его бюджет и в каком направлении он должен идти;
* закрыть множество дыр в бюджете, через которые финансы уходят в неизвестном направлении;
* узнать насколько для человека важно то, на что он сегодня потратил свои деньги;
* стать умнее, эффективнее и успешнее;
* реализовать мечты, и сделать это как можно раньше;
* упорядочить свою жизнь.

Разработка данного приложения связана со значительными финансовыми вложениями и трудовыми затратами, поэтому требует соответствующего технико-экономического обоснования.

Проведение технико-экономического обоснования включает в себя расчет и оценку таких показателей эффективности инвестиций, как срок окупаемости программного средства, чистый дисконтированный доход и рентабельность инвестиций для обеспечения более высокой производительности труда, большей надежности и достоверности информации, лучшей ее сохранности.

## **7.2** Оценка трудоемкости и сроков разработки

Объем программного продукта (ПП) вычисляется на основе нормативных данных.

Полный объем программного продукта рассчитан по средним показателям и приведен в таблице 7.1.

Таблица 7.1 – Перечень и объем функций автоматизированной системы

|  |  |  |
| --- | --- | --- |
| Номер функции | Содержание функции | Объем (строк кода) |
| 101 | Организация ввода информации | 180 |
| 102 | Контроль, предварительная обработка и ввод информации | 520 |
| 109 | Организация ввода\вывода информации в интерактивном режиме | 320 |
| 203 | Формирование баз данных | 2000 |
| 206 | Обслуживание базы данных в интерактивном режиме | 3100 |
| 207 | Манипулирование данными | 8400 |
| 208 | Организация поиска и поиск в базе данных | 5230 |
| 304 | Обслуживание файлов | 520 |
| 506 | Обработка ошибочных и сбойных ситуаций | 430 |
| 507 | Обеспечение интерфейса между компонентами | 1200 |
| 703 | Расчет показателей | 500 |
| Всего | | 22400 |

Общий объем ПП рассчитывается по формуле:



где VO – общий объем ПП;

Vi – объем функций ПП;

n – общее число функцией.

Таким образом, получаем объем данной информационной системы:



На основании общего объема программного продукта определяется нормативная трудоемкость Тн. Данная величина устанавливается с учетом сложности ПО. Для ПО объемом 22400 строк кода – это 3-я группа сложности ПО, соответствует нормативная трудоемкость 414 человеко-дня.

С учетом дополнительного коэффициента сложности КСЛ рассчитывается общая трудоемкость ПП по формуле:

ТО = ТН • КСЛ,

где ТО – общая трудоемкость программного продукта;

ТН – нормативная трудоемкость программного продукта;

КСЛ – дополнительный коэффициент сложности программного продукта.

То = 414 • 1,06 = 439 (чел./дней)

При решении сложных задач, в которых длительный период разработки ПП трудоемкость всегда разбивается по стадиям разработки (техническое задание – ТЗ, технический проект – ТП, рабочий проект – РП, внедрение – ВН) с учетом степени использования типовых программ, новизны и удельного веса трудоемкости всех стадий разработки ПП, а также общей трудоемкости разработки продукта. А на основании общей трудоемкости рассчитывается уточненная трудоемкость, которая учитывает распределения по стадиям (ТУ).



где Тi – трудоемкость разработки ПП на i-й стадии (человеко-дней);

m – количество стадий разработки.

Трудоемкость ПП по стадиям определяется с учетом новизны и степени использования в разработке типовых программ и ПП:

Тi = dCTi • КН • КТ • ТО,

где dСТi – удельный вес трудоемкости i-ой стадии разработки ПП;

КН – поправочный коэффициент, который учитывает степень новизны ПП;

КТ – поправочный коэффициент, который учитывает степень использования.

По степени новизны разрабатываемый программный продукт относится к группе «В», которая имеет поправочный коэффициент 0,7, по степени использования типовых программ данный коэффициент будет равным 0,9.Исходя из степени новизны, определим коэффициенты удельных весов трудоемкости на каждой стадии:

1. Техническое задание:



1. Технический проект:



1. Рабочий проект:



1. Внедрение:



Рассчитаем трудоемкость по стадиям разработки:

ТТЗ = 0,09 • 0,7 • 0,9 • 439 = 25 (человеко-дней)

ТТП = 0,14 • 0,7 • 0,9 • 439 = 39 (человеко-дней)

ТРП = 0,61 • 0,7 • 0,9 • 439 = 169 (человеко-дней)

ТВН = 0,16 • 0,7 • 0,9 • 439 = 44 (человеко-дней)

Уточненная трудоемкость будет равна:

Ту = 25 + 39 + 169 + 44 = 276 (человеко-дней)

**7.3** Расчет затрат на разработку и отпускной цены продукта

Расчет оплаты труда всего персонала (Зо) представлен в таблице 7.2.

Таблица 7.2 – Расчет основной зарплаты научно-технического персонала

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Исполнители | Количество | Трудоемкость, дн. | Среднедневная заработная плата, руб. | Заработная плата, руб. |
| Руководитель | 1 | 76 | 28 | 2128 |
| Инженер-программист | 2 | 100 | 24 | 4800 |
| Всего | | | | 6928 |
| С учетом премий (50%) | | | | 10392 |

Дополнительная зарботная плата (ЗД) вычисляется следующим образом:

ЗД = ЗО • НД : 100,

где HД – норматив дополнительной заработной платы, %.

10392 • 10 : 100 = 1039.20 руб.

Отчисления на социальные нужды (РСОЦ) вычисляются по следующей формуле:

РСОЦ = (ЗО + ЗД ) • НСОЦ : 100,

где HСОЦ – страховые взносы на обязательное социальное страхование наемных работников (34%) и обязательное страхование от несчастных случаев на производстве (0,6%).

РСОЦ = (10392 + 1039.20) • (34 + 0,6) : 100 = 3955,20 руб.

Расходы по статье «Машинное время» (РМ) вычисляются по следующей формуле:

РМ = ЦМ • VО • НМВ : 100,

где ЦМ – цена одного машино-часа;

VО – общий объем программного продукта (строк исходного кода);

НМВ – норматив расхода машинного времени откладки кода (12%).

РМ = 0,8 • 22400 • 12 : 100 = 2150,4 руб.

Накладные расходы (РНАКЛ) вычисляются следующим образом:

РНАКЛ = ЗО • ННАКЛ : 100

где ННАКЛ – норматив накладных расходов (110 %).

РНАКЛ = 10392 • 110 : 100 = 11431,2 руб.

Общая сумма расходов по всем статьям в смете (Спи) вычисляется по следующей формуле:

Сп = Зо + Зд + Рсоц + Рм + Рнакл

Сп = 10392 + 1039.20 + 3955,20 + 2150,4 + 11431,2 = 28968 руб.

Затраты на адаптацию и сопровождение (Рса) вычисляются следующим образом:

РСА = СП • НРСА : 100

где НРСА – норматив расходов на сопровождение и адаптацию (5%).

РСА = 28968 • 5 : 100 = 1448,40 руб.

Полная себестоимость программного продукта (СПОБ) вычисляется следующим образом:

|  |  |  |
| --- | --- | --- |
|  | Споб = Сп + Рса  Споб = 28968 + 1448,40 = 30416,40 руб. |  |

Плановая прибыль (ПП) вычисляется следующим образом:

ПП = СПОБ • РП : 100

где РП – уровень рентабельности (15 %).

ПП = 30416,40 • 15 : 100 = 4562,46 руб.

Налог на добавленную стоимость (НДС) вычисляется по следующей формуле:

НДС = (СПОБ + ПП) • НДС : 100

где НДС – ставка налога на добавленную стоимость (20 %).

НДС = (30416,40 + 4562,46) • 20 : 100 = 6995,77 руб.

В итоге прогнозируемая отпускная цена ПП (ЦП) вычисляется по следующей формуле:

ЦП = СПОБ + ПП + НДС

ЦП = 30416,40 + 4562,46 + 6995,77 = 41974,63 руб.

## **7.4** Расчет стоимостной оценки затрат

Общие капитальные вложения (Ко) потребителя, то есть заказчика, связанные с приобретением, а также внедрением и использованием программного продукта, рассчитываются по формуле:

КО = КПР + КОС

где КПР – затраты пользователя на приобретение ПП по отпускной цене разработчика с учетом стоимости услуг по эксплуатации и сопровождению, руб.;

КОС – затраты пользователя на освоение ПП, р.

Затраты пользователя на освоение программного продукта определяются по формуле:

КОС = СПОБ • НОС : 100,

где Нос – норматив расходов на освоение программного продукта (10%).

КОС = 30416,40 • 10 : 100 = 3041,64 руб.

Общие капитальные затраты заказчика составят:

КО = 41974,63 + 3041,64 = 45016,27 руб.

## **7.5** Расчет стоимостной оценки результата

Для определения экономического эффекта от использования нового ПО у пользователя необходимо сравнить расходы по всем его основным статьям затрат на эксплуатацию нового ПП с расходами по соответствующим статьям базовой версии. При этом за базовую версию следует принимать подобный программный продукт, который используется в действующей автоматизированной системе, или же ручной вариант, если автоматизация системы отсутствует. При сравнении базового и новой версий программного обеспечения в качестве экономического эффекта должна выступать общая экономия всех ресурсов относительно базовой версии.

Результатом разработки и использования программного продукта является прирост чистой прибыли. Необходимо выполнить расчет прироста прибыли за счет экономии материальных и трудовых ресурсов.

Экономия затрат на заработную плату (СЗ) с учетом использования ПП в расчете на объем выполняемых работ определяется по формуле:

СЗ = (СЗО + СЗД) • АН,

где СЗО – экономия затрат на основную заработную плату сотрудника, р.;

СЗД – экономия затрат на дополнительную заработную плату сотрудника, р.;

Ан – объем всех выполненных работ за последний год.

До внедрения ПП трудоемкость заполнения данных бюджета, конвертация его в нужную валюту, формирование таблиц и их дальнейшая обработка составляла 4 человеко-часа, после внедрения программы – 2,5 человеко-часа. В среднем, поступает около 1900 таких операций в год.

Экономия затрат на основную заработную плату (СЗО) определяется:

СЗО = ССМ • (ТС – ТН) : ТЧ : ДР • КПР,

СЗО = 528 • (4 – 2,5) : 8 : 22 • 1,5 = 6,75 руб.

где ССМ – среднемесячная ЗП сотрудника, р.;

ТС, ТН – трудоемкость выполнения работ до и после внедрения нового ПП, нормо-час;

ТЧ – средняя длина рабочего дня, ч.;

ДР – среднемесячное число рабочих дней;

КПР – коэффициент премий (50%).

Экономия затрат на дополнительную заработную плату (СЗД) определяется по следующей формуле:

ЗЗД = ЗЗО • НД : 100,

ЗЗД = 6,75 • 10 : 100 = 0,68 руб.

где HД – норматив дополнительной заработной платы, %.

Экономия затрат по заработной плате при использовании нового программного продукта составляет:

СЗ = (6,75 + 0,68) • 1900 = 14117 руб.

Экономия затрат за счет сокращения начислений по заработной плате (СОЗ) определяется по формуле:

СОЗ = СЗ • Нсоц : 100,

где Нсоц – страховые взносы на обязательное социальное страхование наемных работников (34%) а также обязательное страхование от несчастных случаев на производстве (0,6%).

Соз = 14117 • (34 + 0,6) : 100 = 4884,48 руб.

Экономия затрат на оплату машинного времени (СМ) с учетом выполненного объема работ в результате применения нового программного продукта определяется по формуле:

СМ = СМЕ • АН,

где СМЕ – экономия затрат на оплату машинного времени при решении задач с использованием нового программного продукта, р.

Экономия затрат на оплату машинного времени СМЕ будет определяться по следующей формуле:

СМЕ = ЦМ • (МВС – МВН),

СМЕ = 0,8 • (4 – 2,5) = 1,2 руб.

где Цм – цена за один машино-час работы компьютера, руб.;

Мвс, Мвн – до и после внедрения нового программного продукта, нормо-час;

Экономия затрат на оплату машинного времени с расчетом на выполненный объем работ в результате применения нового продукта составит:

См = 1,2 • 1900 = 2280 руб.

Экономия затрат на материалы (СМТ) с учетом использования нового ПП в расчете на объем работ, которые были выполнены, определяется по формуле:

СМТ= СМТЕ • АН,

где Смте – экономия затрат на материалы с учетом использования нового ПП.

Экономия затрат на материалы с учетом использовании нового программного продукта вычисляется по формуле:

СМТЕ = МТС – МТН,

где Мтс, Мтн– средний расход материалов до и после внедрения нового программного продукта, р.

Экономия затрат на материалы составит:

Смте = 0,14 – 0,12 = 0,02 руб.

Экономия затрат на материалы (Смт) с учетом использования нового программного продукта составит:

СМТ = 0,02 • 1900 = 38 руб.

Общая годовая экономия текущих затрат, которая связана с использованием нового продукта (СО) будет определяться по формуле:

СО = СЗ + ССОЗ + СМ + СМТ

Со = 14117 + 4884,48 + 2280 + 38 = 21319,48 руб.

## **7.6** Расчет показателей эффективности использования продукта

Выпуск в эксплуатацию нового программного продукта позволит пользователю сэкономить на его текущих затратах. В качестве экономического эффекта для пользователя здесь выступает чистая прибыль, то есть дополнительная прибыль, которая остается в его распоряжении (ΔПч). Она определяется по формуле:

ΔПЧ = СО – (СО • НП : 100),

ΔПЧ = 21319,48 – (21319,48 • 18 : 100) = 17481,97 руб.

где Нп – ставка налога на прибыль (18%).

В процессе эксплуатации программного продукта чистая прибыль в общем итоге возмещает все капитальные затраты на производство. Однако, полученные при этом суммы результатов, то есть прибыли, и затрат, капитальных вложений, по годам приводят к единому времени, то есть расчетному году, тут за расчетный год принят 2018 год, путем умножения затрат и результатов за каждый год на коэффициент привидения, который рассчитывается по формуле:

,

где *Е*н – норматив привидения разновременных затрат и результатов (10,5) %;

*t*p – расчетный год;

*t* – номер года, который приводится к расчетному.

Коэффициентам приведения по годам с 2018 до 2021 года будут:

*α*1 = (1 + 0,105)1-1 = 1

*α*2 = (1 + 0,105)1-2 = 0,9

*α*3 = (1 + 0,105)1-3 = 0,82

*α*4 = (1 + 0,105)1-4 = 0,74

Результаты вычислений показателей эффективности внедрения программного продукта представлены в таблице 7.3.

Таблица 7.3 – Показатели эффективности внедрения ПП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Наименование показателей | По годам производства (руб.) | | | |
| 2018 | 2019 | 2020 | 2021 |
| 1. Прирост чистой прибыли (ПЧ) | 17481,97 | 17481,97 | 17481,97 | 17481,97 |
| 2. Тоже с учетом фактора времени  (ПЧ •аt) | 17481,97 | 15733,77 | 14335,22 | 12936,66 |
| 3. Общие капитальные затраты (КО) | 45016,27 | – | – | – |
| 4. Тоже с учетом фактора времени (КО•аt) | 45016,27 | – | – | – |
| 5. Чистый дисконтированный доход (п.2-п.4) | -27534,30 | 15733,77 | 14335,22 | 12936,66 |
| 6. Чистый дисконтированный доход с нарастающим итогом | -27534,30 | -11800,53 | 2534,69 | 15471,35 |
| 7. Коэффициент дисконтирования  (аt) | 1 | 0,90 | 0,82 | 0,74 |

В результате вычисления технико-экономического обоснования применения программного продукта были получены следующие значения показатели эффективности:

* дисконтированный чистый доход продукта за четыре года производства составит 15471,35 рублей;
* все инвестиции окупаются на третий год.

Полученные результаты говорят о рациональности изготовления данного продукта, так как программа окупит себя уже на третьем году работы. А если учесть, что среднестатистический период жизни программы четыре года, то данный программный продукт не только будет нести пользу, но принесет разработчику прибыль.

ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломного проекта были изучены особенности методологии разработки программных продуктов, произведены анализ, проектирование и реализация кроссплатформенного веб-приложения для ведения бюджета. Поведение и структура разработанного программного продукта были представлены в виде различных диаграмм, схем, таблиц.

Разработанное приложение предназначено для ведения личного бюджета пользователя и дает ему возможность стать умнее, эффективнее и успешнее, а также упорядочить свою жизнь и, что немаловажно, закрыть множество просчетов в бюджете. Разработанное приложение позволяет:

* вводить данные с помощью голосовых команд;
* создавать пользователей в системе;
* собирать данных о доходах и расходах;
* строить отчеты и графики расходов;
* анализировать и обрабатывать данные.

Аудиторией применения программы является широкая группа пользователей, вне зависимости от пола и возраста, что делает его универсальным в использовании. В первую очередь данная программа предназначена для людей делающих первые шаги в формировании и управлении своим бюджетом.

В ходе выполнения дипломного проекта также было проведено тестирование продукта, как ручным, так и автоматизированных тестированием. Выявленные неисправности были найдены и устранены. На данный момент работа приложения стабильна и отвечает всем заявленным требованиям.

Разработанный программный продукт является рентабельным и конкурентоспособным. К основным его достоинствам можно отнести:

* голосовой ввод данных;
* управление программой с помощью голосовых команд;
* простой и удобный интерфейс;
* быстрая скорость работы;
* безопасность хранения данных;
* мониторинг текущего дохода пользователя;
* большое количество графических отчетов.

Перечисленные выше достоинства говорят о хорошо проделанной работе. Самым основным и большим плюсом приложения является работа с голосовыми командами пользователя, для чего была подключена система распознавания голоса и разработан модуль для распознавания в произнесенной речи команд понятных пользователю.

Приложение также обладает и недостатками:

* возможность ввода данных без открытия приложения;
* возможность ведения семейного бюджета;
* возможность расчета бюджета для больших покупок;
* возможность уведомления пользователя о недельных затратах;
* возможность перехода в offline режим;
* возможность указания минимальной суммы остатка.

Как можно увидеть у разработанного приложения большой потенциал для развития. А так как архитектура приложения строилась с возможностью расширения функциональность, то добавить новые функции не будет сложной задачей, а кроссплатформенность приложения сделает разработку менее затратной, если сравнивать с разработкой для каждой операционной системы отдельно.

Таким образом, дипломный проект, реализующий кроссплатформенное веб-приложения для ведения бюджета широкой аудиторией людей выполнен в соответствии с заданием и полностью готов к практическому применению.

СПИСОК ИСПОЛЬЗованных ИСТОЧНИКОВ

1. Гибкая разработка веб-приложений / S.Ruby[и др.] – СПб. : Издательство Питер, 2014. – 448 с.
2. Мобильное приложение // Википедия [Электронный ресурс]. – 2018. – Режим доступа: https://en.wikipedia.org/wiki/Mobile\_app. - Дата доступа: 08.04.2018.
3. Камер, Д. Сети TCP/IP, Том III. Разработка приложений типа клиент/сервер для Linux/POSIX / Д. Э. Камер, Д. Л. Стивенс. – М. : Издательский дом «Вильямс», 2002. – 592 с.
4. RESTful Web APIs / L. Richardson [и др.] – Melbourne, 2015.
5. World Wide Web // Википедия [Электронный ресурс]. – 2018. – Режим доступа: https://en.wikipedia.org/wiki/World\_Wide\_Web. - Дата доступа: 08.04.2018.
6. Cordova: Introduction // The Apache Software Foundation [Электронный ресурс]. – 2018. – Режим доступа: https://cordova.apache.org/docs/ru/latest/guide/overview/. –Дата доступа: 09.04.2018.
7. Cordova: APIs and instruments // SlideShare [Электронный ресурс]. – 2014. – Режим доступа: https://www.slideshare.net/iivanoo/cordova-apis-and-instruments. – Дата доступа: 17.04.2018.
8. Mastering TypeScript / Nathan Rozentals – Birmingam, 2015.
9. SQLite // SQLite.org [Электронный ресурс]. – 2018. Режим доступа: https://www.sqlite.org/about.html. - Дата доступа: 11.04.2018.
10. Node.js 8 the Right Way / Jim R. Wilson – USA, 2017.

ПРИЛОЖЕНИЕ А

(*справочное*)

Листинг программного кода

//Модуль работы с базой данных

import \* as sqlite3 from "sqlite3";

const usersDBName = "testDB";

const getDBName = (userId: string) => userId === usersDBName ? `${usersDBName}.db` : `${ userId }.db`;

export { getDBName as GetDBName, usersDBName as UsersDBName };

export default class DataBase {

private dbName: string;

private db: sqlite3.Database;

constructor(dbPath: string) {

this.dbName = dbPath;

this.db = new sqlite3.Database(this.dbName,

sqlite3.OPEN\_READWRITE, e => {

if (e instanceof Error) {

throw e || new Error(

`Cannot connect to DataBase "${ this.dbName }"`);

} else {

console.log(`Connected to DB: ${ this.dbName }`);

}

});

}

public all<T extends any>(sqlScript: string) {

return new Promise<T[]>((resolve, reject) => {

this.db.serialize(() => {

this.db.all(sqlScript, (err, row) => {

if (err instanceof Error) {

reject(err);

} else {

resolve(row);

}

});

});

});

}

public insert(sqlScript: string) {

return new Promise<number>((resolve, reject) => {

this.db.serialize(() => {

this.db.run(sqlScript, function(err) {

if (err instanceof Error) {

reject(err);

} else {

resolve((this as any).lastID as number);

}

});

});

});

}

public remove(sqlScript: string) {

return new Promise((resolve, reject) => {

this.db.serialize(() => {

this.db.run(sqlScript, function (err) {

if (err instanceof Error) {

reject(err);

} else {

resolve();

}

});

});

});

}

public close() {

this.db.close();

}

}

//Менеджер для работы с базами данных

import DataBase, { GetDBName } from "./dbInstance";

interface IDbInstances {

[key: string]: DataBase | undefined;

}

const getPathToDb = (dbName: string) =>

`${ \_\_dirname }/../db/${ dbName }`;

export default class DbManager {

private instances: IDbInstances = {};

constructor(mainDBName: string) {

try {

let mainDb =

new DataBase(getPathToDb(GetDBName(mainDBName)));

this.instances[mainDBName] = mainDb;

} catch (e) {

console.warn(

`Cannot cannect to the DB "${ mainDBName }": ${ e.toString() }`);

}

}

private getInstance(dbName: string) {

let fullName = GetDBName(dbName);

try {

let mainDb = new DataBase(getPathToDb(fullName));

this.instances[dbName] = mainDb;

} catch (e) {

console.warn(

`Cannot cannect to the DB "${dbName}": ${e.toString()}`);

}

return this.instances[dbName];

}

public getDatabase(userId: string) {

let instance = this.instances[userId];

if (instance instanceof DataBase) {

return instance;

}

instance = this.getInstance(userId);

return instance;

}

}

//Серверный API-модуль

import { IIdEntity } from "../wwwroot/ilBudgetto";

import DbManager from "./dbManager";

import UsersApi from "./apiInstances/usersApi";

import DataBase, { UsersDBName } from "./dbInstance";

import IncomingApi from "./apiInstances/incomingApi";

import CurrencyApi from "./apiInstances/currencyApi";

import TypesApi from "./apiInstances/typesApi";

import StableWaste from "./apiInstances/stableWaste";

import StableIcome from "./apiInstances/stableIncome";

import TagApi from "./apiInstances/tagsApi";

export interface IApiEntity<T, Search> {

Add: (db: DataBase, entity: T) =>

PromiseLike<(T & IIdEntity)[]>;

Remove: (db: DataBase, entity: IIdEntity) =>

PromiseLike<IIdEntity[]>;

Set: (db: DataBase, entity: T & IIdEntity) =>

PromiseLike<(T & IIdEntity)[]>;

Get: (db: DataBase, search?: Search) =>

PromiseLike<(T & IIdEntity)[]>;

}

const apies = {

User: new UsersApi(),

Incoming: new IncomingApi(),

Currency: new CurrencyApi(),

Types: new TypesApi(),

StableWaste: new StableWaste(),

StableIncome: new StableIcome(),

Tags: new TagApi()

};

export type IApiType = "User" | "Incoming"

| "Currency" | "Types" | "StableWaste" | "StableIncome" | "Tags";

export type IApiMethod = "Get" | "Set" | "Remove" | "Add";

export interface IApiCall<T extends any> {

type: IApiType;

method: IApiMethod;

entity: T;

}

export default class Api {

private dbManager: DbManager;

constructor(dbManage: DbManager) {

this.dbManager = dbManage;

}

public parseRequest<T extends any>(req: IApiCall<T>) {

let api = this.getApi(req.type);

if (api instanceof Error) {

return Promise.reject(api);

}

let method = req.method;

let checkMethod = this.checkMethod(method, api);

if (checkMethod instanceof Error) {

return Promise.reject(checkMethod);

}

let db = this.getDbInstance(req.type);

if (db instanceof Error) {

return Promise.reject(db);

}

return (api[method] as any)(db, req.entity);

}

public getApi(type: IApiType) {

if (type && type in apies) {

return apies[type];

}

return new Error("Wrong type of call");

}

private getDbInstance(type: IApiType) {

let instance: DataBase | undefined;

if (type === "User") {

instance = this.dbManager.getDatabase(UsersDBName);

} else {

instance = this.dbManager.getDatabase("userDB\_1");

}

if (!instance) {

return new Error("Cannot run request in DB");

}

return instance;

}

private checkMethod(method: IApiMethod, api: IApiEntity<any, any>) {

if (method in api) {

return undefined;

}

return new Error("Wrong method of call");

}

}

//Модуль создания объекта сервера

import \* as express from "express";

import \* as bodyParser from "body-parser";

import { StaticConfig } from "./serverConfig";

const app = express();

app.use(express.static(\_\_dirname + "/../public", StaticConfig));

app.use(bodyParser.json());

export { app as App };

//Модуль запуска сервера

import { App } from "./server/expressInstance";

import \* as https from "https";

import \* as http from "http";

import DbManager from "./server/dbManager";

import Api, { IApiType, IApiMethod } from "./server/api";

import { UsersDBName } from "./server/dbInstance";

import \* as fs from "fs";

import { IUser, IUserSearch } from

"./server/apiInstances/usersApi";

import { getMonthPeriod, addMonths, toServerDate }

from "./wwwroot/js/dateParser";

import { IIncomingSearch, IIncoming }

from "./server/apiInstances/incomingApi";

import { IStableIncomeSearch, IStableIncome }

from "./server/apiInstances/stableIncome";

import { IStableWasteSearch, IStableWaste }

from "./server/apiInstances/stableWaste";

export interface IServerError {

message: string;

isError: boolean;

type: "ServerError" | "ApiError";

}

const DB\_MANAGER = new DbManager(UsersDBName);

const api = new Api(DB\_MANAGER);

const getIncoming = (toDate: number): Promise<IIncoming[]> =>

api.parseRequest({

entity: <IIncomingSearch>{

toDate: toDate

},

method: "Get",

type: "Incoming"

});

const getStableIncome = (toDate: number): Promise<IStableIncome[]> =>

api.parseRequest({

entity: <IStableIncomeSearch>{

toDate: toDate

},

method: "Get",

type: "StableIncome"

});

const getStableWaste = (toDate: number): Promise<IStableWaste[]> =>

api.parseRequest({

entity: <IStableWasteSearch>{

toDate: toDate

},

method: "Get",

type: "StableWaste"

});

const updateUserBudget = (entity: any, reqType: IApiType,

method: IApiMethod) => {

let isUserGet = (reqType === "User" && method === "Get");

let isRelatedUpdate = (requestType: IApiType, requestMethod:

IApiMethod) => requestMethod !== "Get" && (requestType === "Incoming"

|| requestType === "StableIncome" || requestType === "StableWaste");

if (isUserGet || isRelatedUpdate(reqType, method)) {

return api.parseRequest({

entity: {

id: "1"

},

method: "Get",

type: "User"

}).then((users: IUser[]) => {

let user = users[0];

let toDate = getMonthPeriod(

addMonths(new Date(), -1)).toDate;

let serverDate = toServerDate(toDate);

if ((isUserGet && user.lastDate === serverDate) ||

(entity.date && entity.date <= serverDate)) {

return Promise.resolve(entity);

}

return Promise.all([getIncoming(serverDate),

getStableIncome(serverDate), getStableWaste(serverDate)])

.then(data => {

let sum = 0;

data[0].forEach(value => {

sum += value.value;

});

data[1].forEach(value => {

sum += value.value;

});

data[2].forEach(value => {

sum += value.value;

});

return api.parseRequest({

method: "Set",

type: "User",

entity: {...user, ...<IUserSearch>{

lastDate: serverDate,

lastValue: sum

}}

}) as Promise<IUser>;

});

}).then(() => Promise.resolve(entity));

}

return Promise.resolve(entity);

};

App.post("/api", (req, res, next) => {

api.parseRequest(req.body).then(entity =>

updateUserBudget(entity, req.body.type, req.body.method))

.then(entity => {

res.contentType("application/json");

res.statusCode = 200;

res.json(entity);

next();

}).catch(e => {

let message = `Unknown API error: ${ e.toString() }`;

if (e instanceof Error) {

console.warn(e);

message = e.message;

}

let err: IServerError = {

isError: true,

message,

type: "ApiError"

};

res.contentType("application/json");

res.statusCode = 500;

res.json(err);

next();

});

});

let options = {

key: fs.readFileSync("./server/cert/hostkey.pem").toString(),

cert: fs.readFileSync("./server/cert/hostcert.pem").toString()

};

http.createServer(App).listen(8080);

https.createServer(options, App).listen(8443);

//Модуль клиентского API

import { IApiCall } from "../../server/api";

import { IServerError } from "../../server";

import { ServerError } from "./errors";

export default class ClientApi {

private host: string;

constructor(serverHost: string) {

this.host = serverHost;

}

public call<ApiCallType, ResultEntity>(api: IApiCall<ApiCallType>,

method: "POST" | "GET") {

let body = JSON.stringify(api);

let headers = new Headers();

headers.append("Content-Type", "application/json");

headers.append("Content-Length", body.length.toString());

headers.append("Origin", location.origin);

return fetch(`${this.host}/api`, {

method: method,

headers,

cache: "no-store",

credentials: "same-origin",

mode: "same-origin",

body: body

}).then(response => {

let contentType = response.headers.get("Content-Type");

if (response.ok && response.status === 200 && contentType &&

contentType.indexOf("application/json") === 0) {

return response.json();

}

throw new ServerError({

isError: true,

type: "ApiError",

message: "Unknown error on api call"

});

}).then((res: ResultEntity[] | IServerError) => {

if ("isError" in res && "type" in res && "message" in res) {

throw new ServerError(res);

}

if (!res) {

return [];

}

let result = Array.isArray(res) ? res : [res];

return result;

});

}

post<ApiCallType, ResultEntity>(api: IApiCall<ApiCallType>) {

return this.call<ApiCallType, ResultEntity>(api, "POST");

}

}

//Модуль работы с адресной строкой браузера

import { decode, encode, IUrlStateObj } from "./urlCoder";

import \* as address from "./addresses";

export default class UrlState {

private currentState: IUrlStateObj | null = null;

private previousStableState = "";

constructor(urlChange: () => void) {

this.currentState = this.readCurrentState();

address.change(() => {

this.currentState = this.readCurrentState();

urlChange();

});

}

private readCurrentState() {

let newState: IUrlStateObj = {};

try {

newState = this.validateState(this.getStateObject());

this.previousStableState = window.location.hash.slice(1);

} catch (e) {

console.log("Invalid url '" + window.location.hash +

"'. Stepping back to '#" + this.previousStableState + "'");

this.rollbackToPreviousStableState();

if (!newState) {

newState = this.readCurrentState();

}

}

return newState;

}

private rollbackToPreviousStableState() {

window.location.hash = this.previousStableState;

}

private getStateObject() {

return (decode(address.get()) || {}) as IUrlStateObj;

}

private validateState(curState) {

if (!this.isValid(curState)) {

curState = this.getDefaultState();

this.setHash(curState, false);

}

return curState;

}

private isValid(stateCandidate?: IUrlStateObj) {

let etalon = this.getDefaultState();

if (!stateCandidate) {

return false;

}

for (let name in etalon) {

if (!(name in stateCandidate)) {

return false;

}

}

return true;

}

private getDefaultState() {

let result: IUrlStateObj = {},

localState = this.currentState || {};

for (let name in localState) {

result[name] = {};

}

return result;

}

public getHash(obj: IUrlStateObj) {

let stateString = encode(obj);

return stateString ? `#${stateString}` : "";

}

public canGoBack() {

return history.length > 1;

}

public goBack() {

history.back();

}

public setHash(child: IUrlStateObj | string, addToHistory = true) {

let encodedState = typeof child === "string" ? child :

this.getHash(child);

address.set(encodedState, addToHistory);

}

public encodeUrl(state: IUrlStateObj) {

return encode(state);

}

public decodeUrl(state: string) {

return decode(state);

}

public getUrlState() {

return this.currentState;

}

public setUrlState(url: IUrlStateObj, addToHistory = true) {

this.setHash(url, addToHistory);

}

}

//Модуль для загрузки страниц

import UrlState from "./urlState";

import Header from "./header";

import { Pages } from "./pages/pages";

import ClientApi from "./clientApi";

import EntityLoader from "./entityLoader";

import { IUser } from "../../server/apiInstances/usersApi";

import { ICurrency } from "../../server/apiInstances/currencyApi";

import { IType } from "../../server/apiInstances/typesApi";

export interface IPage {

focus: (args: IPageArgs) => void;

initialize: () => void;

blur: () => void;

node: HTMLDivElement;

}

export interface IKeyTypedValue<T> {

[key: string]: T;

}

export interface IKeyValue {

[key: string]: any;

}

export interface IPageArgs {

urlState: UrlState;

setTitle: (text: string) => void;

goBack: () => void;

isCanGoBack: () => void;

getUrlState: () => IKeyValue;

setUrlState: (pagename: string, state?: IKeyValue) => void;

api: ClientApi;

store: EntityLoader;

header: Header;

getUser: () => IUser;

reloadUser: () => Promise<IUser>;

getCurrency: () => ICurrency[];

getTypes: () => IType[];

}

interface IPages {

[key: string]: IPage;

}

export default class PageLoader {

private defPageName = "main";

private currentPage: string;

private activePage: IPage;

private mainBlock: HTMLDivElement;

private pages: IPages = {};

constructor(mainConatiner: HTMLDivElement, defaultPage?: string) {

this.mainBlock = mainConatiner;

if (defaultPage) {

this.defPageName = defaultPage;

}

this.currentPage = this.defPageName;

if (Pages[this.currentPage]) {

let instance = new Pages[this.currentPage]();

this.mainBlock.appendChild(instance.node);

instance.initialize();

this.activePage = instance;

this.pages[this.currentPage] = instance;

} else {

throw new Error("There is not main page");

}

}

public loadPage(pageName: string | undefined, args: IPageArgs) {

let page = pageName ? pageName : this.defPageName;

if (this.currentPage === page) {

this.activePage.focus(args);

return;

}

let pageInstance = this.getPageInstance(page);

if (pageInstance) {

this.activePage.blur();

let child = this.mainBlock.firstChild;

if (child) {

this.mainBlock.removeChild(child);

}

this.currentPage = page;

args.setTitle(page);

this.activePage = pageInstance;

// append new page node and focus page

this.mainBlock.appendChild(this.activePage.node);

this.activePage.focus(args);

}

}

public getPageInstance(pageName: string) {

let page = this.pages[pageName];

if (page) {

return page;

}

if (Pages[pageName]) {

let instance = new Pages[pageName]();

this.mainBlock.appendChild(instance.node);

instance.initialize();

this.pages[pageName] = instance;

return instance;

}

return undefined;

}

}

//Модуль ядра клиенского приложения

import IsSupport from "./js/supported";

import Header, { speachIconId } from "./js/header";

import Menu from "./js/menu";

import UrlState from "./js/urlState";

import PageLoader, { IPageArgs } from "./js/pageLoader";

import ClientApi from "./js/clientApi";

import EntityLoader from "./js/entityLoader";

import SpeachParser from "./js/speachParser";

import { IUser } from "../server/apiInstances/usersApi";

import { ICurrency } from "../server/apiInstances/currencyApi";

import { cogIcon, insertCoinIcon, pieChartColorfullIcon,

moneyBag2Icon, priceTagIcon } from "./js/icons";

import { IType } from "../server/apiInstances/typesApi";

export interface IIdEntity {

id: string;

}

class Budgetto {

private productVersion = "0.5.1";

public support = new IsSupport();

public urlState: UrlState;

public api: ClientApi;

public store: EntityLoader;

private user: IUser;

public speachParser: SpeachParser;

get vesrion() {

return this.productVersion;

}

constructor() {

let headerElement: HTMLDivElement | null =

document.querySelector("#headerId");

let menuElement: HTMLDivElement | null =

document.querySelector("#menuId");

let menuBlock: HTMLDivElement | null =

document.querySelector("#menuBlockId");

let titleElement: HTMLTitleElement | null =

document.querySelector("#titleId");

let mainBlock: HTMLDivElement | null =

document.querySelector("#data\_block");

if (titleElement === null || headerElement === null ||

menuElement === null || menuBlock === null

|| mainBlock === null) {

throw new Error(`Cannot find header element`);

}

let pageLoader = new PageLoader(mainBlock, "main");

this.urlState = new UrlState(() => {

let state = this.urlState.getUrlState();

let pageNameString = state ? state.page : undefined;

pageLoader.loadPage(pageNameString, pageArgs);

menu.classList.remove("shown");

});

this.api = new ClientApi(location.origin);

this.store = new EntityLoader(this.api);

this.speachParser = new SpeachParser();

let menu = menuElement;

let onClickMenu = (e: Event) => {

e.preventDefault();

if (menu.classList.contains("shown")) {

menu.classList.remove("shown");

} else {

menu.classList.add("shown");

}

};

// create a header

let header = new Header(headerElement, onClickMenu);

// create a main menu

new Menu(menuBlock, [

{ name: "Main page", link: "#main", icon: priceTagIcon() },

{ name: "Daily budget", link: "#stable",

icon: moneyBag2Icon() },

{ name: "Analisys", link: "#analisys",

icon: pieChartColorfullIcon() },

{ name: "Transactions", link: "#list",

icon: insertCoinIcon() },

{ name: "Settings", link: "#settings",

icon: cogIcon("#238585", "#43c3c9") }

]);

let title = titleElement;

let pageArgs: IPageArgs = {

urlState: this.urlState,

header,

goBack: () => this.urlState.goBack(),

isCanGoBack: () => this.urlState.canGoBack(),

setTitle: (text: string) => {

title.textContent = `il budgetto | ${ text }`;

},

getUrlState: () => {

let state = this.urlState.getUrlState();

if (state && state.state) {

return state.state;

}

return {};

},

setUrlState: (pageNameStr: string, args: any) => {

this.urlState.setUrlState({

page: pageNameStr,

state: args

});

},

api: this.api,

store: this.store,

getCurrency: () => currency,

getUser: () => this.user,

getTypes: () => types,

reloadUser: () => this.loadUser()

};

let page = this.urlState.getUrlState();

let pageName = page && page.page || undefined;

let currency: ICurrency[];

let types: IType[];

Promise.all([

this.loadUser(),

this.store.currency.get(),

this.store.types.get()

]).then(data => {

let userData = data[0];

let currencyData = data[1];

let typesData = data[2];

if (!userData) {

throw new Error("Cannot get user info");

}

currency = currencyData;

types = typesData;

pageLoader.loadPage(pageName, pageArgs);

});

}

private loadUser() {

return this.store.user.get({ id: "1" }).then(data => {

let userData = data[0];

if (!userData) {

throw new Error("Cannot get user info");

}

this.user = userData;

return userData;

});

}

}

let budgetto = new Budgetto();

declare var ya: any;

window["budgetto"] = budgetto;

window["createSpeachElement"] = function () {

window["textlineElement"] = new

ya.speechkit.Textline("headerSpeachId", {

apikey: "5ad6ae51-9cc2-4301-85bb-ade0d8a077f1",

onInputFinished: function (text) {

let recognize = budgetto.speachParser.recognize(text || "");

let alertText = "";

if (recognize instanceof Error) {

alertText = `${recognize.message}: ${text}`;

} else {

alertText =

`Value: ${recognize.value}

${recognize.currency.name}\nType: General\nComment:

${recognize.comment}`;

}

alert(alertText);

}

});

let input: HTMLInputElement | null =

document.querySelector(`#${speachIconId} input`);

if (input !== null) {

input.readOnly = true;

input.addEventListener("click", (e) => {

e.preventDefault();

return false;

});

}

};

export default budgetto;

//Модуль для распознования команды введения платежа

export interface ISpaechRecognize extends IMoneyParse {

comment: string;

}

interface IMoneyParse {

value: number;

currency: {

id: number;

name: string;

};

}

export default class SpeachParser {

private moneyReg =

/(\d+\s?[р|к|\$]?(доллар(а|ов)?)?(цент(а|ов)?)?)/gm;

private typeReg =

/(\s?(затрат(ы|а)?)|(прибыл(ь|и)?)|(плюс(ани|ы|ик)?)|

(минус(ани|ы|ик)?)|(\-)|(\+)\s?)?/gm;

public recognize(text: string): ISpaechRecognize | Error {

let trimmed = text.trim();

let money = trimmed.match(this.moneyReg);

let result = this.parseMoney(money);

if (result instanceof Error) {

return result;

}

if (money) {

money.forEach(txt => {

trimmed = trimmed.replace(txt, "");

});

trimmed.trim();

}

let types = trimmed.match(this.typeReg);

let accum = this.parseType(types);

result.value \*= accum;

if (types) {

types.forEach(txt => {

trimmed = trimmed.replace(txt, "");

});

trimmed.trim();

}

return {...result, ...{

comment: trimmed.trim() || ""

}};

}

private parseType(text: RegExpMatchArray | null) {

if (!text) {

return -1;

}

for (let i = 0, len = text.length; i < len; i++) {

let txt = text[i].trim();

if (!txt) {

continue;

}

if (txt.indexOf("прибыл") !== -1

|| txt.indexOf("плюс") !== -1

|| txt.indexOf("+") !== -1) {

return 1;

}

if (txt.indexOf("затрат") !== -1

|| txt.indexOf("минус") !== -1

|| txt.indexOf("-") !== -1) {

return -1;

}

}

return -1;

}

private parseMoney(moneyText: RegExpMatchArray | null):

IMoneyParse | Error {

if (!moneyText) {

return new Error("Cannot recognize value");

}

let bigPart = "";

let smallPart = "";

let currency: {

id: number;

name: string;

} | undefined;

let bigReg = /([р|\$]?(доллар(а|ов)?)?)/gm;

for (let i = 0, len = moneyText.length; i < len; i++) {

let text = moneyText[i];

if (!bigPart && bigReg.test(text)) {

let num = this.getDigit(text);

if (num === undefined) {

continue;

}

currency = this.parseCurrency(text);

if (currency === undefined) {

continue;

}

bigPart = `${num}`;

} else if (!smallPart) {

let num = this.getDigit(text);

if (num === undefined) {

continue;

}

if (!bigPart && !currency) {

currency = this.parseCurrency(text);

}

smallPart = `${ num }`;

}

if (bigPart && smallPart) {

break;

}

}

let value = parseFloat(`${ bigPart }.${ smallPart || 0 }`);

if (!isNaN(value)) {

return {

value,

currency: currency || { id: 2, name: "BYN" }

};

}

return new Error("Cannot recognize value");

}

private getDigit(text: string) {

let digit = /(\d+)/gm;

let res = text.match(digit);

if (res !== null && res.length === 1) {

let num = parseInt(res[0]);

if (!isNaN(num)) {

return num;

}

}

return undefined;

}

private parseCurrency(text: string) {

if (text.indexOf("оллар") !== -1

|| text.indexOf("цент") !== -1

|| text.indexOf("$") !== -1) {

return { id: 1, name: "USD" };

}

if (text.indexOf("р") !== -1) {

return { id: 2, name: "BYN" };

}

return undefined;

}

}