[ВВЕДЕНИЕ 5](#_Toc453867677)

[1 АНАЛИЗ И ПОСТАНОВКА ЗАДАЧИ 7](#_Toc453867678)

[1.1 Анализ предметной области 7](#_Toc453867679)

[1.2 Описание существующих аналогов 7](#_Toc453867680)

[1.3 Формирование рекомендаций по созданию АСОИ 10](#_Toc453867681)

[1.4 Постановка задачи на создание АСОИ 11](#_Toc453867682)

[2 ПРОЕКТИРОВАНИЕ АСОИ 12](#_Toc453867683)

[2.1 Назначение АСОИ 12](#_Toc453867684)

[2.2 Проектирование программного обеспечения АСОИ 13](#_Toc453867685)

[2.2.1 Инструментальные средства для реализации АСОИ 13](#_Toc453867686)

[2.2.2 Проектирование интерфейса пользователя с системой 18](#_Toc453867687)

[2.2.3 Архитектура клиентской части 19](#_Toc453867688)

[2.2.4 Входные и выходные данные 25](#_Toc453867689)

[3 РЕАЛИЗАЦИЯ И ИСПЫТАНИЕ АСОИ 28](#_Toc453867690)

[3.1 Прототипирование интерфейса 28](#_Toc453867691)

[3.2 Настройка Ruby on Rails 30](#_Toc453867692)

[3.3 Создание корневого приложения 33](#_Toc453867693)

[3.4 Шаблонизатор 35](#_Toc453867694)

[3.5 Модели/Коллекции 37](#_Toc453867695)

[3.6 Модули 39](#_Toc453867696)

[3.7 Подмодуль Users.List 41](#_Toc453867697)

[3.7.1 Представления подмодуля Users.List 42](#_Toc453867698)

[3.7.2 Контроллер подмодуля Users.List 43](#_Toc453867699)

[3.8 Модуль User 45](#_Toc453867700)

[3.8.1 Представления подмодуля User.Show 46](#_Toc453867701)

[3.8.2 Контроллер подмодуля User.Show 47](#_Toc453867702)

[3.9 Тестирование АСОИ 49](#_Toc453867703)

[4 РАСЧЕТ ЭКОНОМИЧЕСКИХ ПОКАЗАТЕЛЕЙ 53](#_Toc453867704)

[4.1 Расчет объема функций программного обеспечения 53](#_Toc453867705)

[4.2 Расчет себестоимости программного средства 55](#_Toc453867706)

[4.3 Определение цены ПП и чистой прибыли 58](#_Toc453867707)

[5 РЕСУРСО- И ЭНЕРГОСБЕРЕЖЕНИЕ 61](#_Toc453867708)

[ЗАКЛЮЧЕНИЕ 62](#_Toc453867709)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 63](#_Toc453867710)

ВВЕДЕНИЕ

Изначально World Wide Web (WWW) представлялась ее создателям как «пространство для обмена информацией, в котором люди и компьютеры смогут общаться между собой». Поэтому первые web-приложения представляли собой примитивные файл-серверы, возвращавшие статические HTML-страницы.

Однако на сегодняшний день web-приложения ушли достаточно далеко от простых статических страниц и накладывают серьезные ограничения на инфраструктуру, необходимую для их работы. Большинство современных web-приложений, вне зависимости от их направленности, динамичны и позволяют пользователю взаимодействовать с сервером без перезагрузки страницы. Более того, некоторые web-приложения позволяют получать обновления данных в режиме реального времени. Однако данные возможности требуют не только широкий и надежный канал связи, но высокую скорость работы самого приложения, что можно обеспечить только высокой эффективностью хранения и обработки данных.

Размеры современного интернет-пространства также накладывают серьезные ограничения на web-приложения — они должны легко масштабироваться и оставаться надежными под изменяющейся нагрузкой.

Редко можно встретить приложение, имеющее заметную коммерческую ценность и менее ста тысяч пользователей. Такие масштабы требуют особых подходов к проектированию архитектуры приложения и использования специальных технологий.

Выбранная предметная область – одностраничные web-приложения (Single Page Applications), весьма актуальна и бурно развивается. Все крупнейшие корпорации, такие как Google и Facebook, Яндекс и VK, продвигают свои методологии и технологии для упрощения разработки и надежности web-приложений. Главным трендом на данный момент, является создание легко расширяемой и легко поддерживаемой большой командой разработчиков, архитектуры.

Целью данного дипломного проекта является разработка приложения для обработки и визуализации статистики продаж в виде масштабируемого web-приложения.

Для достижения этой цели необходимо выполнить следующие задачи:

* проанализировать основные подходы к проектированию масштабируемых web-приложений;
* определить инструментарий, с помощью которого возможно создание приложения;
* спроектировать архитектуру собственного приложения, основные его компоненты и механизм их взаимодействия;
* реализовать и развернуть web-приложение;
* провести тестирование приложения и доработку по мере необходимости.

1 АНАЛИЗ И ПОСТАНОВКА ЗАДАЧИ

1.1 Анализ предметной области

Целью автоматизации данной предметной области является регулирование бизнес-процессов компании за пределами корпоративной сети и обеспечение доступа сотрудникам к данным по продажам удаленно через интернет в любой точке мира.

Данная автоматизация облегчает процесс контроля над продажами, сокращает расходы на эксплуатацию (резервное копирование, администрирование, информационную безопасность) и позволяет обеспечить доступ к содержимому с различных настольных и мобильных устройств.

В качестве объекта автоматизации рассматривается web-приложение, отвечающее за бизнес-процессы компании.

Для успешной и эффективной организации этой деятельности, учета, контроля и управляемости, существует необходимость:

* иметь постоянно обновляемую нормативно-справочную информации о структуре и трудовых ресурсах предприятия, структуре и характеристиках объектов обслуживания, структуре, содержании, нормативной трудоемкости и стоимости работ для всех видов обслуживания;
* обеспечить оперативное, максимально удобное для руководства и персонала предприятия получение и отображение данных;
* иметь в любой момент времени данные о состоянии продаж.

Автоматизация этих задач позволит повысить уровень централизации управления трудовыми, материальными и финансовыми ресурсами предприятия, расширить зону обслуживания, повысить его качество, в конечном итоге повысить эффективность бизнеса, его конкурентоспособность.

1.2 Описание существующих аналогов

В качестве аналогичного приложения рассмотрим Cyfe <http://www.cyfe.com/> (см. рисунок 1.1).



Рисунок 1.1 – Website Cyfe

Cyfe является приложением по типу все-в-одном, которое помогает контролировать бизнес-данные и бизнес-процессы в одном месте. Панель управления Cyfe предоставляет открытый доступ для клиентов и заказчиков к данным в режиме реального времени, позволяет создавать и рассылать отчеты клиентам, кастомизировать виджеты и экспортировать/импортировать данные.

Также, к преимуществам Cyfe относятся:

* кастомные источники данных;
* кастомный домен и логотип;
* кастомная визуализация данных;
* архив;
* экспорт PDF/CSV;
* доступ к API;
* легкость настройки.

Стартовая страница выглядит следующим образом (см. рисунок 1.2).



Рисунок 1.2 – Стартовая страница приложения

Предоставлен базовый функционал. В верхней части экрана находятся панели навигации: переключатель между панелями управления, функциональная панель, переключатель дат архива. В левой части экрана расположена навигационная панель администрирования приложения. В контентной области приложения находятся виджеты, отображающие определенные данные.

Функциональная панель содержит следующие кнопки:

* добавление виджета;
* создание новой панели управления;
* редактирование названия текущей панели управления;
* изменение фона;
* переключение в режиме просмотра для TV;
* экспорт панели управления;
* дублирование панели управления;
* удаление панели управления.

Навигационная панель отвечает за перемещение между структурными элементами приложения:

* список всех доступных пользователю панелей управления;
* список всех клиентов и заказчиков;
* настройки приложения;
* справочная информация;
* выход из учетной записи пользователя;
* индикатор и триггер темы оформления.

Однако, для решения нашей задачи, данное приложение является избыточном по своему функционалу. Главным преимуществом нашего приложения должны быть конфиденциальность и хранение данных на нашем сервере. Приложение Cyfe не адаптируется под различные размеры экранов устройств.

Большая часть функционала доступна лишь по платной подписке, что составляет 50$/месяц без учета налогов и комиссий. Базовый функционал не рассчитан на полноценное использование, а пригоден лишь в качестве демонстрации возможностей.

1.3 Формирование рекомендаций по созданию АСОИ

Созданное приложение должно предоставлять пользователю информацию о продажах конкретного продавца и обеспечить визуализацию данных на странице, а также иметь постоянно обновляемую нормативно-справочную информацию о структуре и трудовых ресурсах предприятия, структуре и характеристиках объектов обслуживания, структуре, содержании, нормативной трудоемкости и стоимости работ для всех видов обслуживания;

Интерфейс приложения должен быть интуитивно понятным, неперегруженным ненужным функционалом, отзываться на определенные действия пользователя.

Стартовая страница должна содержать список продавцов в виде таблицы с различными фильтрами, сортировками, поиском, постраничной навигацией и возможностью перехода на страницу конкретного продавца из списка.

В качестве элементов интерфейса должны быть графики, диаграммы, гистограммы, которые визуально отображают различные этапы и периоды продаж отдельно взятого продавца. Важнейшим элементом интерфейса должно являться представление информации о продавце в виде таблиц, а также таблицы с данными о текущих продажах.

Организационная структура продаж тесно связана со стратегией компании и ее организационной структурой в целом. Организационная структура продаж является частью системы продаж. Систему продаж можно определить как взаимосвязанную совокупность оргструктур, сбытовых сетей, каналов и технологий продаж. Организационная структура продаж представляет собой совокупность подразделений продаж компании и органов управления.

1.4 Постановка задачи на создание АСОИ

На основании рассмотренной информации об аналоге в подразделе 1.2, обозначим основные задачи при проектировании АСОИ:

* создание простого и удобного пользовательского интерфейса, учитывая специфику приложения;
* обеспечить быстродействие приложения;
* обеспечить возможность получать данные со своего сервера;
* поддерживать адаптивный дизайн.

**Общесистемные требования:**

* приложение должно обеспечивать необходимый уровень интерактивности;
* приложение должно иметь иерархическую структуру;
* ресурсы приложения должны быть подготовлены для финальной версии продукта.

**Требования к структуре:**

* приложение должно быть легко масштабируемо;
* необходимо использовать общепринятые паттерны проектирования архитектуры.

**Требования к техническому обеспечению:**

* приложение должно работать на устройствах с минимальной тактовой частотой процессора 1 GHz и оперативной памятью не менее 256 Мб.

**Требования к программному обеспечению:**

* приложение проектируется для работы на операционных системах под управлением Windows 7 и старше, Mac OS 10 и старше, iOS 8, Android 5 и старше.

**Требования к лингвистическому обеспечению:**

* для клиентской стороны приложения рекомендуется использовать JavaScript(ECMA Script 5), HTML5, CSS3;
* для сборки проекта рекомендуется использовать Ruby On Rails.

2 ПРОЕКТИРОВАНИЕ АСОИ

2.1 Назначение АСОИ

В соответствии с требованиями, которые определены к приложению в постановке задачи на дипломное проектирование (пункт 1.3), определим назначение разрабатываемого приложения.

Разрабатываемое приложение предназначено для облегчения мониторинга информации о продажах и сделках.

Потенциальным пользователем данного приложения может быть любое физическое или юридическое лицо, которое владеет устройством с браузером и доступом в интернет. Контент приложения является динамическим, т.е. может изменяться ежеминутно.

Приложение должно иметь удобную навигацию, интуитивно понятный интерфейс, удобный поиск, постраничную навигацию, сортировку и фильтрацию данных.

Рассмотрим более детально каждую функцию системы.

Навигация должна обеспечивать переходы между состояниями и страницами приложения. Данная возможность позволяет сориентировать клиента в иерархической структуре приложения и обеспечивает быстрый доступ к необходимым данным.

Интуитивно понятный интерфейс предназначен для упрощения работы с приложением. Немаловажным фактором является быстрота освоения работы приложения пользователем.

Поиск обеспечивает максимально быстрый доступ к необходимой пользователю информации.

Постраничная навигация является способом облегчить пользователю процесс перемещения между данными.

Сортировка – это процесс упорядочивания элементов в списке. В случае, когда элемент списка имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки.

Фильтрация – это вид сортировки по определенным запросам данных.

2.2 Проектирование программного обеспечения АСОИ

2.2.1 Инструментальные средства для реализации АСОИ

Сценарный язык (язык сценариев от англ. scripting language) — высокоуровневый язык сценариев (англ. script) — кратких описаний действий, выполняемых системой. Разница между программами и сценариями довольно размыта.

**JavaScript.**

**JavaScript** изначально создавался для того, чтобы сделать web-странички «живыми». Программы на этом языке называются скриптами. В браузере они подключаются напрямую к HTML и, как только загружается страничка – тут же выполняются.

Программы на JavaScript – обычный текст. Они не требуют какой-то специальной подготовки.

В этом плане JavaScript сильно отличается от другого языка, который называется Java.

JavaScript может выполняться не только в браузере, а где угодно, нужна лишь специальная программа – интерпретатор. Процесс выполнения скрипта называют «интерпретацией».

Во все основные браузеры встроен интерпретатор JavaScript, именно поэтому они могут выполнять скрипты на странице. Но, разумеется, JavaScript можно использовать не только в браузере. Это полноценный язык, программы на котором можно запускать и на сервере, и даже в стиральной машинке, если в ней установлен соответствующий интерпретатор.

Современный JavaScript – это «безопасный» язык программирования общего назначения. Он не предоставляет низкоуровневых средств работы с памятью, процессором, так как изначально был ориентирован на браузеры, в которых это не требуется.

Что же касается остальных возможностей – они зависят от окружения, в котором запущен JavaScript. В браузере JavaScript умеет делать всё, что относится к манипуляции со страницей, взаимодействию с посетителем и, в какой-то мере, с сервером:

* Создавать новые HTML-теги, удалять существующие, менять стили элементов, прятать, показывать элементы и т.п.;
* Реагировать на действия посетителя, обрабатывать клики мыши, перемещения курсора, нажатия на клавиатуру и т.п.;
* Посылать запросы на сервер и загружать данные без перезагрузки страницы (эта технология называется "AJAX");
* Получать и устанавливать cookie, запрашивать данные, выводить сообщения.

JavaScript – быстрый и мощный язык, но браузер накладывает на его исполнение некоторые ограничения.

Это сделано для безопасности пользователей, чтобы злоумышленник не мог с помощью JavaScript получить личные данные или как-то навредить компьютеру пользователя.

Этих ограничений нет там, где JavaScript используется вне браузера, например на сервере. Кроме того, современные браузеры предоставляют свои механизмы по установке плагинов и расширений, которые обладают расширенными возможностями, но требуют специальных действий по установке от пользователя

Большинство возможностей JavaScript в браузере ограничено текущим окном и страницей.

* JavaScript не может читать/записывать произвольные файлы на жесткий диск, копировать их или вызывать программы. Он не имеет прямого доступа к операционной системе. Современные браузеры могут работать с файлами, но эта возможность ограничена специально выделенной директорией – «песочницей». Возможности по доступу к устройствам также прорабатываются в современных стандартах и частично доступны в некоторых браузерах;
* JavaScript, работающий в одной вкладке, не может общаться с другими вкладками и окнами, за исключением случая, когда он сам открыл это окно или несколько вкладок из одного источника (одинаковый домен, порт, протокол);
* из JavaScript можно легко посылать запросы на сервер, с которого пришла страница. Запрос на другой домен тоже возможен, но менее удобен, т. к. и здесь есть ограничения безопасности.

**jQuery.**

jQuery — библиотека JavaScript, фокусирующаяся на взаимодействии JavaScript и HTML. Библиотека jQuery помогает легко получать доступ к любому элементу DOM, обращаться к атрибутам и содержимому элементов DOM, манипулировать ими. Также библиотека jQuery предоставляет удобный API для работы с AJAX. Сейчас разработка jQuery ведётся командой jQuery во главе с Джоном Резигом.

Основной целью создания jQuery Резиг видел возможность закодировать многоразовые куски кода, которые позволят упростить JavaScript и использовать их так, чтобы не беспокоиться о кросс-браузерных вопросах. Библиотека была представлена общественности на компьютерной конференции «BarCamp» в Нью-Йорке в 2006 году.

Библиотека jQuery содержит функциональность, полезную для максимально широкого круга задач. Тем не менее, разработчиками библиотеки не ставилась задача совмещения в jQuery функций, которые подошли бы всюду, поскольку это привело бы к большому коду, бо́льшая часть которого не востребована. Поэтому была реализована архитектура компактного универсального ядра библиотеки и плагинов. Это позволяет собрать для ресурса именно ту JavaScript-функциональность, которая на нём была бы востребована.

Работу с jQuery можно разделить на 2 типа:

* получение jQuery-объекта с помощью функции $(). Например, передав в неё CSS-селектор, можно получить jQuery-объект всех элементов HTML, попадающих под критерий и далее работать с ними с помощью различных методов jQuery-объекта. В случае, если метод не должен возвращать какого-либо значения, он возвращает ссылку на jQuery объект, что позволяет вести цепочку вызовов методов согласно концепции текучего интерфейса;
* вызов глобальных методов у объекта $, например, удобных итераторов по массиву.

**CoffeeScript.**

CoffeeScript ([’kɔ:fɪ skrɪpt]; кофи скрипт) — язык программирования, транслируемый в JavaScript. CoffeeScript добавляет синтаксический сахар в духе Ruby, Python, Haskell и Erlang для того, чтобы улучшить читаемость кода и уменьшить его размер. CoffeeScript позволяет писать более компактный код по сравнению с JavaScript.

JavaScript-код, получаемый трансляцией из CoffeeScript, полностью проходит проверку JavaScript Lint.

Создателем языка является Джереми Ашкенас.

Изначально компилятор был написан на Ruby, но в версии 0.5, которая вышла 21 февраля 2010 года, компилятор был реализован на самом же CoffeeScript.

CoffeeScript был радушно воспринят в Ruby-сообществе. Встроенная поддержка CoffeeScript была добавлена в веб-фреймворк Ruby on Rails с версии 3.1.

Главным преимуществом данного языка является встроенная поддержка классов, которые нативно не поддерживаются в JavaScript.

**BackboneJS.**

Backbone.js придает структуру веб-приложениям с помощью моделей с биндингами по ключу и пользовательскими событиями, коллекций с богатым набором методов с перечислимыми сущностями, представлений с декларативной обработкой событий; и соединяет это все с вашим существующим REST-овым JSON API.

Backbone.js— это небольшая JavaScript-библиотека, которая структурирует код клиентской стороны приложения. Она упрощает управление задачами и распределение их в приложении, упрощая поддержку кода.

Библиотека Backbone многофункциональна и популярна: вокруг нее существует активное сообщество разработчиков, а для самой библиотеки имеется множество плагинов и расширений. Backbone используется для создания нестандартных приложений такими компаниями, как Disqus, Walmart, SoundCloud и LinkedIn.

Главная цель Backbone — обеспечить удобные методы считывания данных и манипуляции ими, чтобы избавить разработчиков от необходимости заново реализовывать объектную модель JavaScript. Backbone — это, скорее, не фреймворк, а библиотека, — хорошо масштабируемая и эффективно работающая с другими компонентами, от встраиваемых виджетов до полномасштабных приложений.

Библиотека Backbone предоставляет разработчику минимальный набор примитивов для структурирования данных (модели, коллекции) и пользовательских интерфейсов, полезных при создании динамических приложений на JavaScript. Она не накладывает строгих ограничений на разработку и обеспечивает свободу и гибкость в выборе методов создания оптимальных интерфейсов для веб-приложений. Можно воспользоваться штатной архитектурой Backbone или расширить ее под свои требования.

Главное в библиотеке Backbone — это не набор виджетов и не альтернативный способ структурирования объектов; библиотека лишь предоставляет приложению инструменты для считывания данных и манипулирования ими. Backbone также не требует, чтобы разработчик пользовался конкретным шаблонизатором; можно использовать микрошаблоны библиотеки Underscore.js (или одной из ее зависимостей) и, таким образом, связывать представления с HTML-кодом, созданным с помощью выбранного шаблонизатора.

Многочисленные примеры приложений, созданных с помощью библиотеки Backbone, наглядно демонстрируют ее способность к масштабированию. Backbone также успешно работает с другими библиотеками, что позволяет встраивать Backbone-виджеты в приложения на AngularJS, совместно использовать Backbone и TypeScript или же применять отдельные классы Backbone (например, модели) для работы с данными в простых приложениях. Структурирование приложений с помощью Backbone не ухудшает их производительность. В Backbone не используются циклы, двухстороннее связывание, непрерывный опрос обновлений структур данных, а применяемые механизмы преимущественно просты.

Backbone включает хорошо составленную документацию своего исходного кода, с помощью которой любой разработчик легко разберется в том, что происходит «за кулисами».

**MarionetteJS.**

Marionette.js — масштабируемая и составная архитектура для приложений на базе BackboneJS.

BackboneJS, отличный инструмент для создания одностраничных клиентских приложений, но как и видно из названия, он дает только самую базу. А что делать, когда наше приложение разрастается и требуется применять более сложные архитектурные паттерны? Именно для этого и нужен MarionetteJS. В нем есть, много того, что не достает BackboneJS. К примеру автоматизация таких процессов как: рендеринга представления, написание с нуля разных вариантов видов (itemView, collectionView, compositeView, region, layout), уже ставшими шаблонными в BackboneJS разработке. Так же MarionetteJS упрощает работу с событиями и их прослушкой.

Многие события, которые используются всегда, в MarionetteJS стоят по умолчанию. Множество дополнительных методов придающих гибкости вашим видам, моделям и коллекциям, а так же улучшенная работа с утечками памяти.

2.2.2 Проектирование интерфейса пользователя с системой

Пользовательский интерфейс приложения представляет собой иерархическую структуру. Запуск приложения осуществляется в среде браузера Google Chrome. Пользователь вводит в адресную строку адрес приложения.

При запуске первой страницы, мы видим список всех доступных пользователей в виде таблицы. Основная функция этой страницы – поиск необходимого пользователя и переход на странице об его детальной информации. Таблица представляет собой наиболее структурированный элемент страницы, содержит заголовки полей, возможности сортировки, фильтрации, постраничной навигации и поиска. Элементы таблицы разделены чересстрочными фонами для наиболее удобной читабельности больших объемов данных.

Для перехода на страницу с данными о конкретном пользователе необходимо нажать кнопку “Show” (показать). Данная страница содержит общие данные о пользователе, такие как:

* Representative (представитель);
* Team (команда);
* Coach (тренер);
* Start date (начало работы);
* End date (конец работы);
* Opportunities (возможные продажи);

Также есть блок с метриками, которые отображают различные виды статистики по продажам в виде диаграмм, круговых диаграмм и графика.

Заключительной таблице на странице является таблица с данными по возможным продажам с возможностями, которая содержит ряд полей, таких как:

* название;
* аккаунт;
* сумма;
* стадия;
* вероятность сделки.

2.2.3 Архитектура клиентской части

При создании архитектуры нашего приложение прибегнем к помощи паттернов проектирования.

Паттерны проектирования — это проверенные решения типичных задач разработки, помогающие улучшить организацию и структуру приложений. Используя паттерны, мы применяем коллективный опыт квалифицированных разработчиков в решении подобных задач.

Разработчики настольных и серверных приложений уже давно пользуются многочисленными паттернами проектирования, однако в разработке клиентских приложений подобные паттерны стали применяться лишь несколько лет назад.

Библиотека BackboneJS использует паттерн проектирования “Модель-Представление-Контроллер” (Model-View-Controller, MVC).

MVC представляет собой паттерн проектирования архитектуры, который улучшает структуру приложения путем разделения его задач. Он позволяет изолировать бизнес-данные (модели) от пользовательских интерфейсов (представлений) с помощью третьего компонента (контроллеров), который управляет логикой и вводом пользовательских данных, а также координирует модели и представления (см. рисунок 2.1).

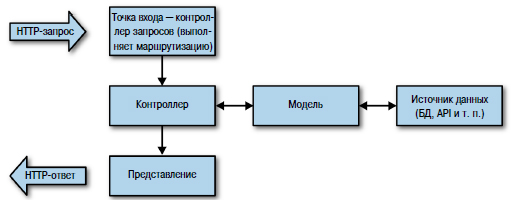


Рисунок 2.1 - Жизненный цикл запросов/ответов HTTP для MVC

**MVC на клиентской стороне и одностраничные приложения.**

Ряд исследований подтвердил, что сокращение задержек при взаимодействии пользователей с сайтами и приложениями положительно влияет на их популярность, а также на степень вовлеченности их пользователей. Это противоречит традиционному подходу к разработке веб-приложений, который сконцентрирован на серверной части и требует полной перезагрузки страницы при переходе с одной страницы на другую. Даже при наличии мощного кэша браузер должен выполнить разбор CSS, JavaScript и HTML, а также отобразить интерфейс на экране.

Такой подход не только приводит к повторной передаче пользователю большого объема данных, но и отрицательно влияет на задержки и общую быстроту взаимодействия интерфейса с пользователем. В последние годы для уменьшения ощущаемых пользователем задержек разработчики зачастую создают одностраничные приложения, которые сначала загружают единственную страницу, а затем обрабатывают навигационные действия и запросы пользователей, не перезагружая страницу целиком.

Когда пользователь переходит к новому представлению, приложение обращается за дополнительным содержимым для этого представления с помощью запроса XHR (XMLHttpRequest), как правило, к серверному REST API или конечной точке. AJAX (Asynchronous JavaScript and XML, асинхронный JavaScript и XML) обеспечивает асинхронное взаимодействие с сервером, при котором передача и обработка данных происходит в фоновом режиме без вмешательства в работу пользователя с другими частями страницы. Это делает интерфейс более быстрым и удобным.

Одностраничные приложения могут использовать такие возможности браузера, как API журнала для обновления поля адреса при перемещении из одного представления в другое. Эти URL также позволяют пользователям создавать закладки на определенных состояниях приложения и обмениваться ими без необходимости загружать совершенно новые страницы. Типичное одностраничное приложение содержит небольшие логические элементы с собственными пользовательскими интерфейсами, бизнес-логикой и данными (см. рисунок 2.2).

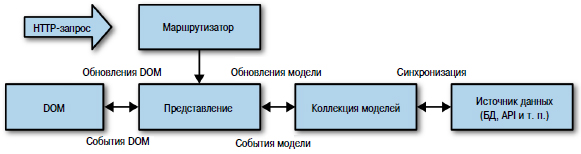


Рисунок 2.2 - Подход Backbone к обработке запросов

**Модели.**

В зону ответственности модели относятся:

* модели обычно поддерживают валидацию атрибутов, где атрибуты представляют свойства модели, например ее идентификатор;
* при использовании моделей в реальных приложениях нам, как правило, требуется сохранять их. Это дает возможность редактировать и обновлять модели, зная при этом, что их последние состояния будут сохранены, например, в локальном хранилище веб-браузера или синхронизованы с базой данных;
* за изменениями одной модели могут одновременно наблюдать несколько представлений;
* фреймворки нередко предоставляют способы группировки моделей. В Backbone такие группы называются коллекциями. Управление моделями в группах позволяет создавать логику приложения на основе уведомлений, поступающих от группы, когда модель внутри группы изменяется. Это устраняет необходимость отслеживать состояния отдельных моделей вручную. Ниже мы рассмотрим этот механизм в действии. Коллекции также полезны при выполнении сводных вычислений с участием нескольких моделей.

В нашем случае понадобятся следующие модели данных:

* список всех пользователей;
* конкретный пользователь и его данные;
* данные для построения визуальных элементов каждого конкретного пользователя.

**Представления.**

В зону ответственности представлений относятся:

* пользовательское взаимодействие с отображением, считыванием и редактированием данных модели;
* отображение содержимого модели с помощью механизма шаблонов;
* наблюдение за изменения в модели/коллекции.

Для нашего приложения мы создадим представления для отображения всех пользователей, а также конкретного пользователя с внутренними подпредставлениями, что обеспечит модульности представлений, легкость поддержки такого кода. Среди подпредставлений мы выделим основные составные части, такие как:

* детали профиля пользователя;
* графики и диаграммы, отображающие состояния продаж за определенные периоды времени;
* таблица предложений с их атрибутами.

**Использование шаблонов.**

Шаблонизаторы JavaScript часто используются для определения шаблонов представлений в виде HTML-разметки, содержащей переменные шаблона. Блоки шаблонов могут храниться вне или внутри тегов <script> с нестандартным типом (например, text/template). Переменные отделены друг от друга с помощью специального синтаксиса.

Как правило, шаблонизаторы JavaScript принимают данные в различных форматах, в том числе в последовательном формате JSON, который всегда является строкой. Рутинная работа по заполнению шаблонов данными выполняется самим фреймворком.

**Контроллеры.**

Большинство JavaScript-фреймворков, реализующих паттерн MVC, отклоняется от его традиционной интерпретации в том, что касается контроллеров.

В Backbone представления, как правило, содержат логику контроллера, а маршрутизаторы помогают управлять состоянием приложения, однако ни те ни другие не являются контроллерами по канонам MVC.

Однако, т.к. в своем дипломном проекте мы используем MarionetteJS, логику контроллера мы назначим на Marionette.Controller. По сути это объект, целью которого является управление модулями, роутерами, видами и т.д.

В нашем приложении контроллеры будут регулировать отношения между модулями и подмодулями.

Схематично это будет выглядеть следующим образом (см. рисунок 2.3).

скачанные файлы.png

Рисунок 2.3 – Схема работы контроллеров приложения

Рассмотрим составные части библиотеки MarionetteJS. Из важнейших частей следует выделить:

* Marionette.Application — объект приложения который запускает его, добавляет инициализаторы и многое другое;
* Marionette.AppRouter — делает роутер конфигуратором;
* Marionette.Callbacks — управление набором функций обратного вызова, и выполнение их по мере необходимости;
* Marionette.View — базовый класс вида, от которого наследуются все остальные виды Marionette.js. Не предназначен для прямого использования;
* Marionette.ItemView — представление для рендеринга индивидуальной модели;
* Marionette.CollectionView — представление, которое перебирает коллекцию и рендерит индивидуальный экземпляр itemView для каждой модели;
* Marionette.CompositeView — составное представление, предназначенное для рендеринга сложных иерархий моделей;
* Marionette.Layout — вид, который рендерит разметку и создает менеджер регионов (частей разметки) для управления ими в его пределах;
* Marionette.Commands — дополнение для Backbone.Wreqr.Commands, которое позволяет компонентам в приложении утверждать, что некоторая работа должна быть сделана, но без того, чтобы быть явно связаной с компонентом, который выполняет работу;
* Marionette.Controller — объект, целью которого является управление модулями, роутерами, представлениями и т.д.;
* Marionette.Module — модули и подмодули приложения;
* Marionette.Region — управление визуальными регионами приложения, включая отображение и удаление контента;
* Marionette.Renderer — рендер шаблонов с данными или без них, последовательным и обычным способом;
* Marionette.RequestResponse — расширение Backbone.Wreqr.RequestResponse — простой запрос/ответ фреймворк.
* MarionetteJS содержит внутри себя библиотеки агрегатторов событий:
* Backbone.Wreqr.EventAggregator — агрегатор событий, используется чтобы облегчить работу с событиями. Реализация pub/sub (издатель/подписчик) шаблона проектирования;
* Backbone.Wreqr.Commands — простая система выполнения команд;
* Backbone.Wreqr.RequestResponse — простая "запрос/ответ" система.

Паттерн MVC помогает разделять логику приложения и пользовательский интерфейс, упрощая модификацию и поддержку и того и другого. Благодаря такому разделению разработчику гораздо легче понять, где вносить изменения в данные, интерфейсы и бизнес-логику приложения и что должны проверять модульные тесты.

**2.2.4 Входные и выходные данные**

В качестве входных данных для web-приложения будем использовать данные в формате JSON. Он представляет собой простой формат обмена данными, удобный для чтения и написания как человеком, так и компьютером. Он основан на подмножестве языка программирования JavaScript. Хранение информации каждого клиента будет происходить в базе данных сервера. Так же на Backend части приложения будет реализован API, благодаря чему, мы сможем получать информацию с помощью простого HTTP запроса. Данная возможность позволяет нам получать информацию с любой реализации Backend, т.е. мы не привязаны к определенной платформе.

Для нашей реализации в рамках фреймворка BackboneJS, нам необходимо оборачивать входные данные в сущность, понятную фреймворку, а именно Model/Collection. Это позволит применять предопределенные методы для обработки и фильтрации данных. Также мы реализуем создание дефолтной модели, т.к. в случае незаполненности некоторых полей, мы будем выводить пустое значение, а не неопределенное.

В качестве дефолтной модели мы определяем следующие поля для пользователя:

defaults:

"avatar" : ""

"annualSalesObjeective": ""

"team" : ""

"coach": ""

"startDate" : ""

"endDate" : ""

"closed" : ""

"closedOf" : ""

"newOpportunities" : ""

"newOpportunitiesOf" : ""

"sales" : ""

"target" : ""

"opportunityList" : [

{

name: ""

account: ""

amount: ""

stage: ""

closeDate: ""

winProbability: ""

representative: ""

}

]

Для создания представлений графических элементов на странице пользователя мы будем использовать различные форматы ввода данных. Для создания круговой диаграммы параметрами будут массив значений и интерполяционная функция, а также селектор в DOM дереве:

new (Chartist.Pie)(

'#chart--proposal-ratio',

series: [5,3,4],

labelInterpolationFnc: (value) ->

Math.round(value / data.series.reduce(sum) \* 100) + '%'

)

Выходными же данными в нашем случае будет визуализация этих данных на стороне клиента.

# 3 РЕАЛИЗАЦИЯ И ИСПЫТАНИЕ АСОИ

**3.1 Прототипирование интерфейса**

Для прототипирования пользовательского интерфейса воспользуемся библиотекой Bootstrap 3.

Существует несколько способов начать работу с Bootstrap, каждый из которых интересен в зависимости от уровня опыта и конкретной потребности использования.

Сайт MaxCDN предоставил Bootstrap возможность пользоваться услугами CDN для распространения файлов CSS и JavaScript. Чтобы воспользоваться этой возможностью, укажем ссылки на ресурсы в разделе head:

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css">

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"script>

Bootstrap доступен в двух версиях. Одна из них содержит скомпилированный вариант, а другая минимизированный (уменьшенный по объему). В обоих случаях каталоги и файлы логически сгруппировны.

Bootstrap автоматически адаптирует просмотр страниц под разные разрешения мониторов.

Bootstrap спроектирован для лучшей работы в новых браузерах, то есть старые браузеры не всегда могут правильно отображать стили, хотя полностью функциональны в визуализации определенных компонентов.

Используя разметку и классы Bootstrap в соответствии с документацией, создаем прототип нашего приложения.

Главная и внутренняя страницы содержат все указанные в главе 2.1 элементы (см. рисунки 3.1-3.5).

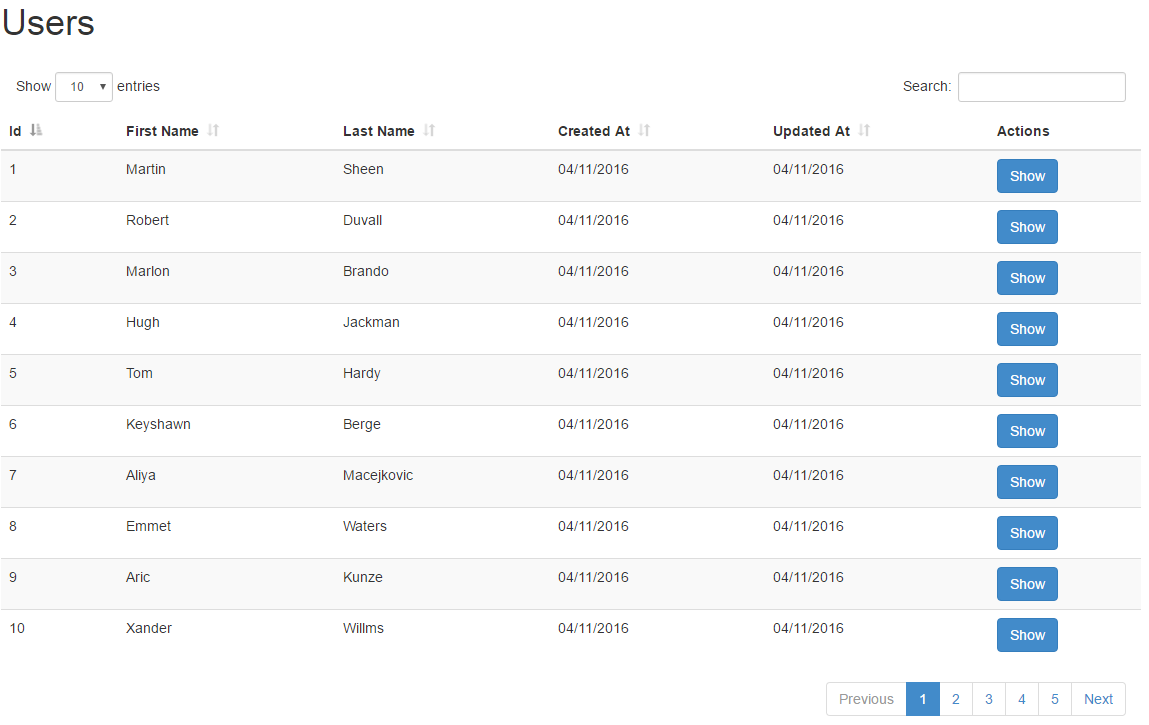


Рисунок 3.1 – Главная страница приложения.

При переходе на страницу конкретного пользователя мы видим следующие секции (см. рисунки 3.2-3.4).

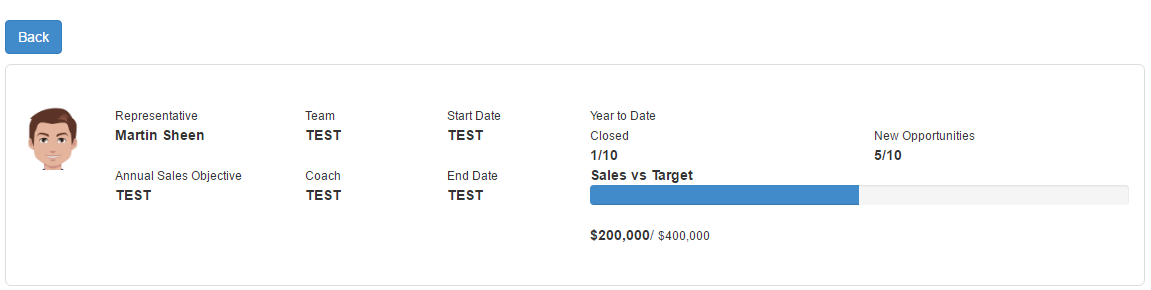


Рисунок 3.2 – Информация о пользователе

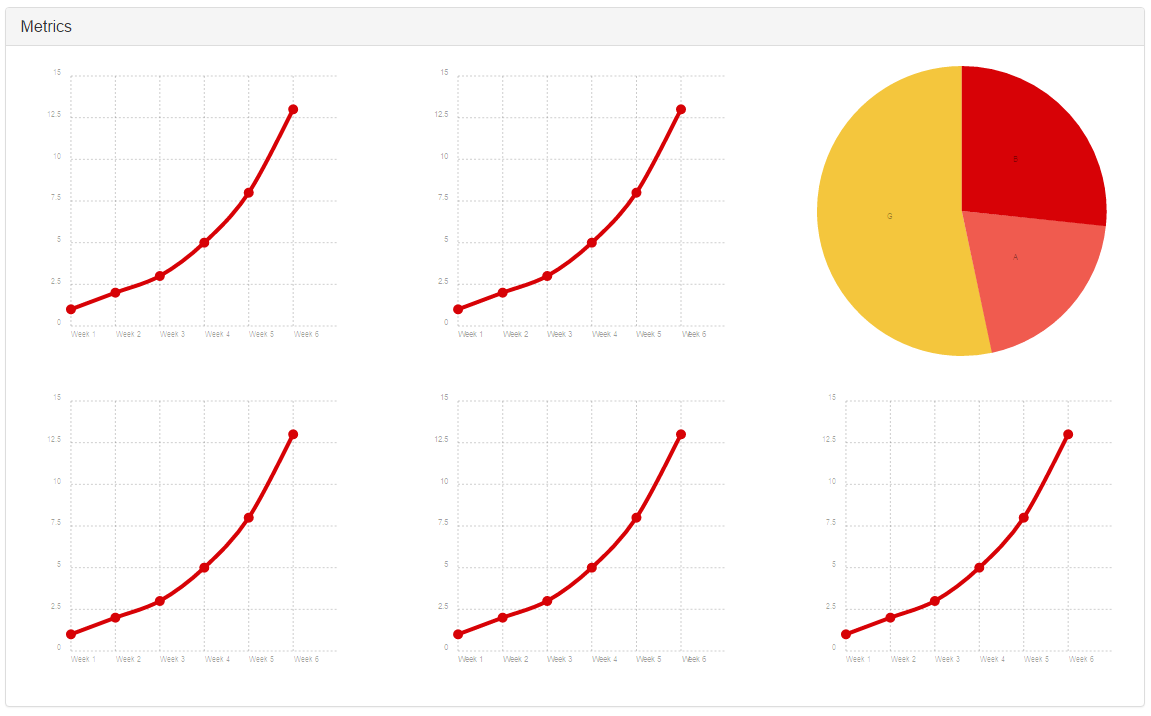


Рисунок 3.3 – Метрики пользователя

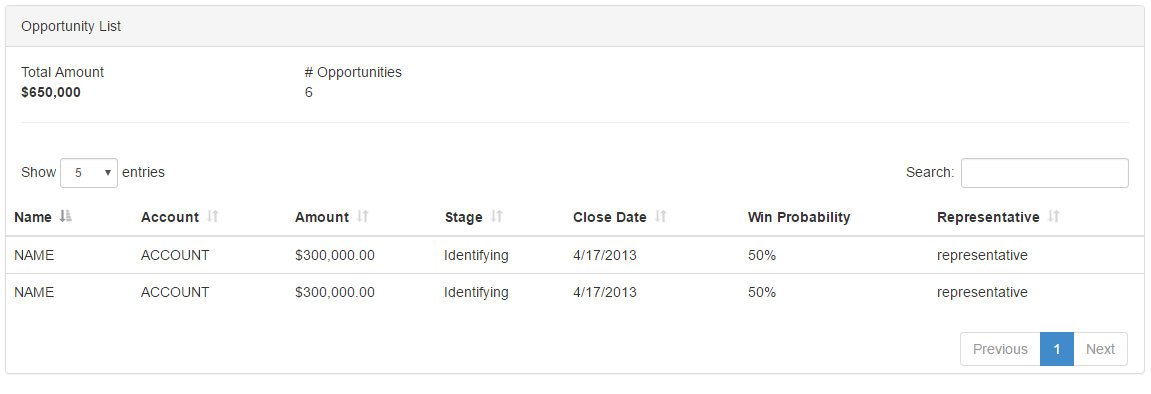


Рисунок 3.4 – Таблица возможных продаж

**3.2 Настройка Ruby on Rails**

В качестве обработчика файлов проекта и сборки статических ресурсов будем использовать сервер Ruby on Rails.

Ruby on Rails (RoR) — фреймворк, написанный на языке программирования Ruby, реализует архитектурный шаблон Model-View-Controller для веб-приложений, а также обеспечивает их интеграцию с веб-сервером и сервером баз данных.

Вокруг Rails сложилась большая экосистема плагинов, которые также называются «джемы» (gem с англ. — «самоцвет»). Для управлениями плагинами существует специальная система RubyGems. Некоторые из них со временем были включены в базовую поставку Rails, например Sass и CoffeeScript; другие же, хотя и не были включены в базовую поставку, являются стандартом де-факто для большинства разработчиков, например, средство модульного тестирования RSpec.

Так как разработка приложения происходит под управлением OS Windows 8, для установки RoR воспользуемся пакетом RubyInstaller for Windows (<http://rubyinstaller.org/downloads/>). Скачиваем пакет Ruby 2.3.0, устанавливаем его в операционную систему. Настраиваем переменные окружения в соответствии с технической документацией сайта.

Для создания приложения на Ruby on Rails переходим в необходимую директорию, открываем в ней окно команд, зажав клавишу Shift + правая клавиша мыши (см. рисунок 3.5).



Рисунок 3.5 – Открытие окна команд

В окне команд пишем команду **rails new dashboard**, где команда **new** отвечает за создание нового приложения, **dashboard** – просто название приложения (см. рисунок 3.6).

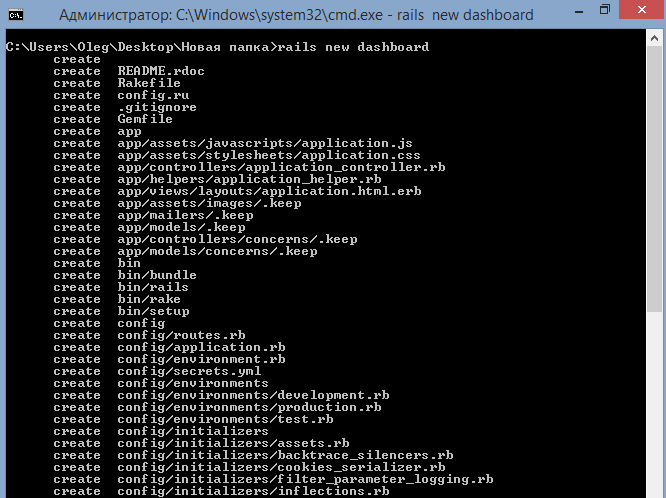


Рисунок 3.6 – Создание нового приложения RoR

В корневой директории проекта открываем файл Gemfile и указываем необходимые для разработки пакеты (джемы) (см. рисунок 3.7).



Рисунок 3.7 – Список необходимых пакетов

Для старта приложения на Ruby on Rails, в окне команда пишем: **rails s**, где **s** – означает **serve** (исполнять, обслуживать).

Для просмотра приложения, открываем браузер и в адресной строке указываем 0.0.0.0:3000.

При запущенном приложении Ruby on Rails на этом адресе и этом порту мы увидим экран приветствия (см. рисунок 3.8).

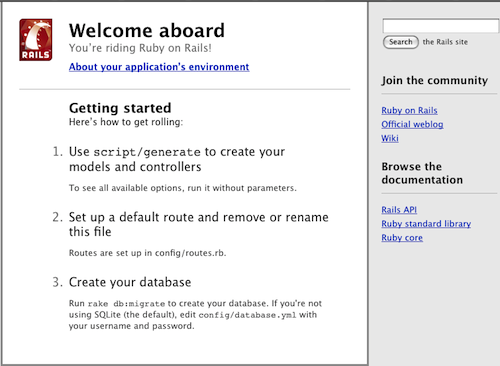


Рисунок 3.8 – Экран приветствия Ruby on Rails

Сервер запущен и готов к разработке.

**3.3 Создание корневого приложения**

Первым шагом в создании приложения на BackboneJS/MarionetteJS является создание корневого файла приложения.

Создаем файл app.js.coffee (двойное расширение необходимо для сборщика проекта Ruby on Rails) в директорию /app/assets/javascripts/backbone/app.js.coffee.

Следуя лучшим практикам JavaScript приложений, оборачиваем наше приложение в функцию, в которую передаем зависимости:

@MR = do (Backbone, Marionette) ->

Данный код фрагмент кода создает в глобальной области window (глобальный объект в каждом браузере) анонимную самовызывающуюся функцию с двумя аргументами – нашими библиотеками.

Затем оборачиваем наше приложение в обертку MarionetteJS, что позволит использовать методы и помощники этой библиотеки:

App = new Marionette.Application

Для использования стандартного роута используем метод rootRoute. Так как первой страницей при загрузке должна быть страница пользователей, указываем имя:

App.rootRoute = "users"

Расширим базовые объекты приложения методами-помощниками. В директории /app/assets/javascripts/backbone/config/marionette/ создаем файл application.js.coffee со следующим кодом:

do (Backbone) ->

\_.extend Backbone.Marionette.Application::,

navigate: (route, options = {}) ->

route = "#/" + route

Backbone.history.navigate route, options

getCurrentRoute: ->

Backbone.history.fragment

Оборачиваем в анонимную самовызывающуюся функцию и передаем в качестве аргумента библиотеку Backbone. С помощью библиотеки Underscore (\_ - это ссылка на объект библиотеки) расширяем объект Backbone.Marionette.Application вешая на его прототип через супер-метод (::) два новых метода:

* navigate (навигация) – принимает роут, на который нужно перейти и опции;
* getCurrentRoute (получить текущий роут) – используем метод модуля Backbone.history, который возвращает нам строку с текущим роутом.

Следующим действием будет настройка и инициализация модуля history.

App.on "start", () ->

if Backbone.history

Backbone.history.start() unless Backbone.History.started

@navigate(@rootRoute, trigger: true) if @getCurrentRoute() is ""

При полной загрузке приложения, объект App генерирует событие “start”, мы подписываемся на это событие и в функции обратного вызова (callback) проверяем доступен ли модуль Backbone.history. Если модуль доступен и загружен, инициализируем его и переводим пользователя методом @navigate на корневой роут @rootRoute, который хранит значение “users”. Переход на страницу users осуществляется только в том случае, если адресная строка не содержит никаких других клиентских роутов.

В последней строке application.js.coffee мы возвращаем объект приложения App, для дальнейшего chaning (сцепливания) – возможность добавлять функционал к текущему объекту.

Теперь, при переходе по адресу приложения 0.0.0.0:3000 (localhost), путь автоматически изменится на <http://localhost:3000/#/users> (см. рисунок 3.9).

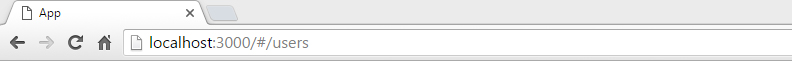


Рисунок 3.9 – Адресная панель при загрузке приложения

Создаем регион “application”, куда будем помещать представления всего приложения:

App.addRegions

application : "#application"

В качестве ключа указывается имя региона (application), в качестве значения мы указываем валидный jQuery селектор, который начинается с “#”, если мы ищем по идентификатору элемента или с “.”, если ищем по классу HTML.

**3.4 Шаблонизатор**

Далее переходим в каталог /app/views/application/index.html.erb. Здесь будет храниться наша разметка-обертка всего приложения и JavaScript код, инициализирующий “start” событие:

<div class="container">

<div class="row">

<div class="col-lg-12">

<br>

<div id="application"></div>

</div>

</div>

</div>

<%= javascript\_tag do %>

$(function() {MR.start();});

<% end %>

HTML-разметка представляет собой классы библиотеки Bootstrap, которые создают контейнеры для представлений, выравнивают по центру область приложения и обеспечивают адаптивность для различных устройств. Здесь же расположен наш главный регион “application”. Сейчас он пустой, и заполнять мы его будем с помощью шаблонизатора.

В качестве шаблонизатора мы будем использовать ECO (Embedded CoffeeScript templates). Данный шаблонизатор позволит нам рендерить представления на серверной стороне, что ускорит быстродействие приложения на клиентской стороне. Для его использования в MarionetteJS нам необходимо переопределить стандартный функционал рендеринга шаблонов. Для этого создадим в директории /app/assets/javascripts/backbone/config/marionette/ файл renderer.js.coffee и согласно документации библиотеки переопределим метод объекта Rendered.render:

Backbone.Marionette.Renderer.render = (template, data) ->

path = JST["backbone/apps/" + template]

unless path

throw "Template #{template} not found!"

path(data)

Метод принимает два аргумента:template(шаблон) и data(данные). Т.к. все наши шаблоны будут храниться в директории /app/assets/javascripts/backbone/apps, мы используем этот адрес в качестве префикса, чтобы избавиться от повторяющихся строк кода. В случае, если путь к шаблону указан неверно, возвращаем ошибку в консоль.

**3.5 Модели/Коллекции**

Модели в Backbone.js содержат данные приложения, а также логику, связанную с этими данными: валидацию, вычисляемые поля и т.д.

Коллекции — это упорядоченные наборы моделей.

Backbone.sync — функция, которую Backbone вызывает каждый раз, когда пытается прочитать/сохранить модель с/на сервер. По умолчанию она использует (jQuery).ajax, чтобы делать RESTful JSON-запросы, и возвращает jqXHR. Её можно переопределить, чтобы использовать другую стратегию персистентности — WebSocket'ы, XML-транспорт, или localStorage.

В реализации по умолчанию, когда Backbone.sync посылает запрос на сохранение модели, её атрибуты будут переданы, сериализованные как JSON, и посланы в теле HTTP с контент-типом application/json. Возвращая JSON-ответ, посылайте атрибуты модели, которые были изменены сервером, и должны быть обновлены на клиенте.

Функция jQuery.ajax() выполняет асинхронный HTTP запрос (Ajax), что создает для нас определенные трудности при последовательности отображения представлений. Проблема в том, что сначала будет отображаться представление, на момент пустое, т.к. данные в фоновом режиме еще не загрузились клиенту, затем произойдет ререндер представления. С точки зрения пользователя это будет выглядеть как дефект.

Хранение сущностей приложения, а именно моделей и коллекций, является одним из важнейших вопросов. С одной стороны необходимо реализовать простой механизм получения данных с сервера, с другой стороны, данный механизм должен быть достаточно гибким.

Создадим отдельный каталог, расположенный вне директорий наших приложений /dashboard/app/assets/javascripts/backbone/entities. Это обеспечит доступ к необходимым данным всем модулям глобально.

Перепишем метод загрузки данных с сервера:

App.commands.setHandler "when:fetched", (entities, callback) ->

xhrs = \_.chain([entities]).flatten().pluck("\_fetch").value()

$.when(xhrs...).done ->

callback()

Использую встроенный паттерн сообщений, создаем исполняемый код “when:fetched”, позволяющий в качестве функции обратного вызова выполнять по сути синхронный код. Т.е. до того момента, пока не загружены необходимые данные, код из callback выполняться не будет. Это решает проблему с дефектом интерфейса при отображении представлений.

Для каждой сущности данных в директории entities (сущности) создаем отдельный файл, к примеру, для модели и коллекции пользователей, users.js.coffee:

class Entities.User extends Entities.Model

defaults:

…

Класс коллекции выглядит следующим образом:

class Entities.Users extends Entities.Collection

model: Entities.User

url: "users"

Свойство (или функция) url предназначена для того, чтобы указывать положение коллекции на сервере. Модели в коллекциях с определённым url будут использовать его, чтобы конструировать свои собственные URL'ы.

Определим объект API, через который будем манипулировать данными и сущностями:

API =

getUsers: ->

users = new Entities.Users

users.fetch

reset: true

users

Метод getUsers реализует следующий функционал:

* создаем экземпляр класса Entities.Users;
* используем метод fetch, который получает актуальные данные с сервера в виде jqXHR-объекта;
* возвращаем объект users в соответствии с документацией.

Используя паттерн сообщений MarionetteJS, а именно reqres, который по сути является реализацией "подписчик/издатель”, создадим издателя “users:entities”:

App.reqres.setHandler "users:entities", ->

API.getUsers()

**3.6 Модули**

Все приложение мы разделим на условные части по функциональности и смыслу. В каталоге /app/assets/javascripts/backbone/apps мы будем хранить наши модули / modules. Будем придерживаться общепринятого в среде разработчиков MarionetteJS паттерна хранения модулей:

* отдельная папка для каждого модуля;
* одноименный файл модуля с расширением js.coffee;
* подпапки, в случае необходимости, в которых лежат подмодули;
* каждая папка / подпапка модуля/подмодуля хранит в себе папку с шаблонами (templates), module.view.js.coffee – файл, который хранит представления и их функционал, module.controller.js.coffee – файл, который хранит логику данного модуля;
* исключить прямое общение подмодулей внутри модуля без использования медиатора.

Рассмотрим модуль “users” с точки зрения этого паттерна.

Структура модуля представлена на рисунке 3.10.

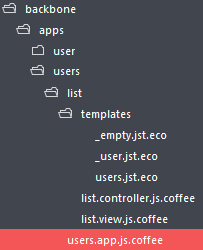


Рисунок 3.10 – Структура модуля Users

Для создания модуля в MarionetteJS используем метод module нашей обертки @MR:

@MR.module "Users", (Users, App, Backbone, Marionette, $, \_) ->

Модуль принимает три аргумента:

* имя модуля;
* зависимости и библиотеки;
* функция обратного вызова.

Для обработки URL-адресов на клиентской стороне модуля “Users” определим класс роутера. Наследуем методы от роутера Marionette – AppRouter.

Для роутера указываем контроллер и свойство appRoutes, в которое передаем непосредственно URL-адрес и имя функции, которая перехватит управление приложением при переходе по этому адресу.

class Users.Router extends Mn.AppRouter

controller: API

appRoutes:

"users" : "list"

Для инициализации нашего объекта роутера создаем новый экземпляр объекта и добавляем его в инициализацию приложения следующим образом:

App.addInitializer ->

new Users.Router

controller: API

В каждом модуле определяем приватный объект API. Он является примером инкапсуляции модулей, т.е. никакой другой модуль не сможет вызвать его методы извне:

API =

list: ->

Users.List.Controller.list()

Функция list вызывает подмодуль List, а конкретно его объект Controller. Модуль Users является связующим для всех своих подмодулей и в его обязанности входит общение с другими модулями через каналы сообщения , встроенные в Marionette.

Главная задача разработки на данном этапе, является необходимость недопустить общение частей приложения, которые расположены на разных уровнях. Это обеспечит предсказуемость разработки и поддержки кода.

**3.7 Подмодуль Users.List**

Рассмотрим подмодуль List модуля Users подробнее.

Определяем его следующим образом:

@MR.module "Users.List", (List, App, Backbone, Marionette, $, \_) ->

Вложенность подмодуля указывается разделителем - точкой.

Для отображения данных, определяем классы представлений. Воспользуемся предопределенными классами MarionetteJS.

**3.7.1 Представления подмодуля Users.List**

Класс представления для отображения единичного элемента коллекции (либо модели) называется ItemView. Определяем его следующим образом:

class List.User extends Mn.ItemView

template: "users/list/templates/\_user"

tagName: "tr"

Теперь, имея одиночный элемент, создадим класс, в темплейте которого будем его итерировать.

class List.Users extends Mn.CompositeView

template: "users/list/templates/users"

childView: List.User

childViewContainer: "tbody"

emptyView: List.Empty

Также предусмотрим вероятность того, что записей в данной коллекции может и не быть. Создадим класс представления, который отобразится в этом случае:

class List.Empty extends Mn.ItemView

template: "users/list/templates/\_empty"

tagName: "tr"

Темплейты представляют собой HTML5 разметку со специальными элементами шаблонизатора, заключенными в символы <%= %>.

Пример темплейта \_user.jst.eco:

<td><%= @id %></td>

<td><%= @first\_name %></td>

<td><%= @last\_name %></td>

<td><%= @created\_at\_formatted %></td>

<td><%= @updated\_at\_formatted %></td>

<td>

<button type="button" class="btn btn-primary js-show">Show</button>

</td>

Ответственным за логику подмодуля Users.List является List.Controller, определенный в файле list.controller.js.coffee.

**3.7.2 Контроллер подмодуля Users.List**

Для работы в одном пространстве имен подмодуля Users.List, объявим файл контроллера таким же образом, как и представления подмодуля:

@MR.module "Users.List", (List, App, Backbone, Marionette, $, \_) ->

Объявляем объект контроллера, привязанный к подмодулю List:

List.Controller =

Реализуем функцию инициализации представления, с возможностью передать в качестве аргумента коллекцию пользователей:

getView: (users) ->

new List.Users

collection: users

new List.Users – экземпляр класса представляения из list.view.js.coffee.

Для получения данных о пользователях воспользуемся запросом к издателю “users:entities”:

users = App.request "users:entities"

Затем используя полученные данные, получив экземпляр представления, отобразим список пользователей на странице:

App.execute "when:fetched", users, =>

# Get View

view = @getView users

# Render

App.application.show view

В определенный в app.js.coffee файле регион “application” помещаем HTML разметку с данными в div с идентификатором “#application”.

Благодаря библиотеке Bootstrap, таблица пользователей выглядит следующим образом (см. рисунок 3.11).

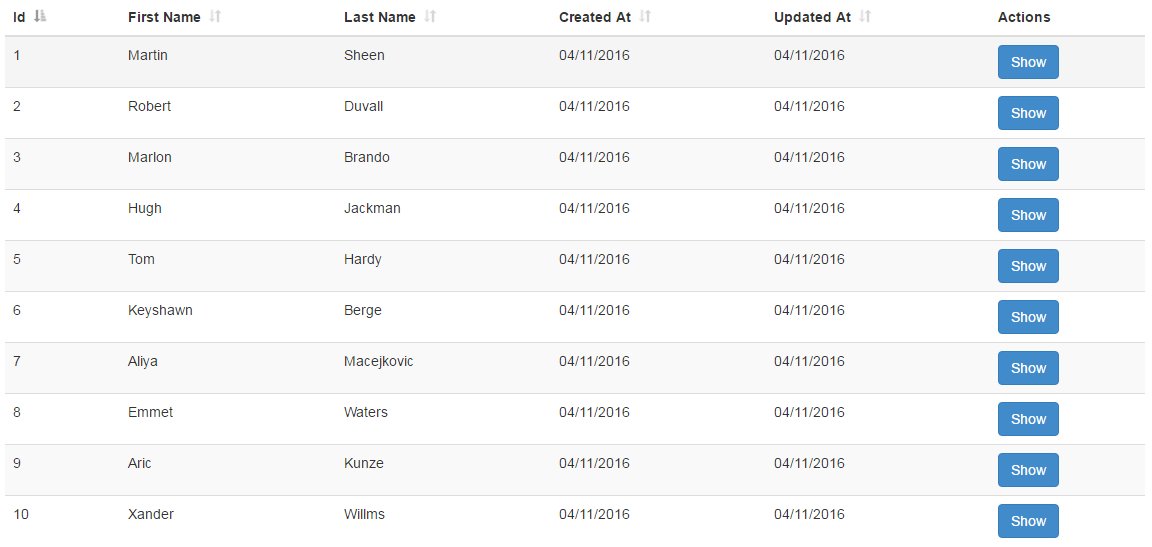


Рисунок 3.11 – Таблица Users

При нажатии кнопки “Show” в каждой из строк таблицы, мы переходим на соответствующего пользователя. Работу данной части системы обеспечивает модуль User, которое находится в директории /apps/user/user.app.js.coffee.

Рассмотрим его детально.

**3.8 Модуль User**

Как и в модуле Users определяем модуль User:

@MR.module "User", (User, App, Backbone, Marionette, $, \_) ->

Класс Router выглядит следующим образом:

class User.Router extends Mn.AppRouter

controller: API

appRoutes:

"users/user/:id" : "show"

Инициализируем Router:

App.addInitializer ->

new User.Router

controller: API

Объект API, отвечающий за логику модуля:

API =

show: (id) ->

User.Show.Controller.show id

Стандартным поведением модуля при его запуске будет метод show, определенный в объекте API:

User.on "start", ->

API.show()

Ключевым моментом в данном модуле, является перехват события “user:show”, по которому мы получаем идентификатор пользователя и переводим приложение по URL-адресу:

App.vent.on "user:show", (user) ->

App.navigate "users/user/#{user.attributes.id}", {trigger: true}

API.show user

Данное событие триггерится в модуле Users и через паттерн передачи сообщений MarionetteJS доходит до модуля User. В атрибутах переданной по каналу модели мы используем идентификатор id. Затем используя метод “show” рендерим все необходимые подмодули через User.Show.Controller:

API =

show: (id) ->

User.Show.Controller.show id

**3.8.1 Представления подмодуля User.Show**

Все представления подмодуля будут рендериться в определенный Layout, для которого мы создадим отдельный файл в директории модуля User, apps/user/show/show.view.js.coffee:

@MR.module "User.Show", (Show, App, Backbone, Marionette, $, \_) ->

class Show.Layout extends Mn.LayoutView

template: "user/show/templates/layout"

className: "user"

regions:

details: "#user-details-region"

charts: "#user-charts-region"

opportunity: "#user-opportunity-region"

events:

"click [data-role='back']": "back"

back: ->

App.vent.trigger "user:back"

class Show.Details extends Mn.ItemView

template: "user/show/templates/\_details"

className: "user-details"

class Show.Charts extends Mn.ItemView

template: "user/show/templates/\_charts"

className: "user-charts"

class Show.Opportunity extends Mn.ItemView

template: "user/show/templates/\_opportunity"

className: "user-opportunity"

В классах ItemView определены представления составных частей страницы, каждая из которых будет содержать свою логику и шаблоны.

**3.8.2 Контроллер подмодуля User.Show**

Определим метод “show” в объекте контроллера подмодуля:

Show.Controller =

show: (model) ->

users = App.request "users:entities"

App.execute "when:fetched", users, =>

user = users.get(model)

@layout = @getLayout()

@layout.on "show", =>

@showDetails user

@showCharts user

@showOpportunity user

App.application.show @layout

Представления Details, Charts, Opportunity инициализируется внутри Layout, что позволяет создать регионы для каждого представления.

Таким образом, мы получаем следующую страницу пользователя (см. рисунок 3.12).

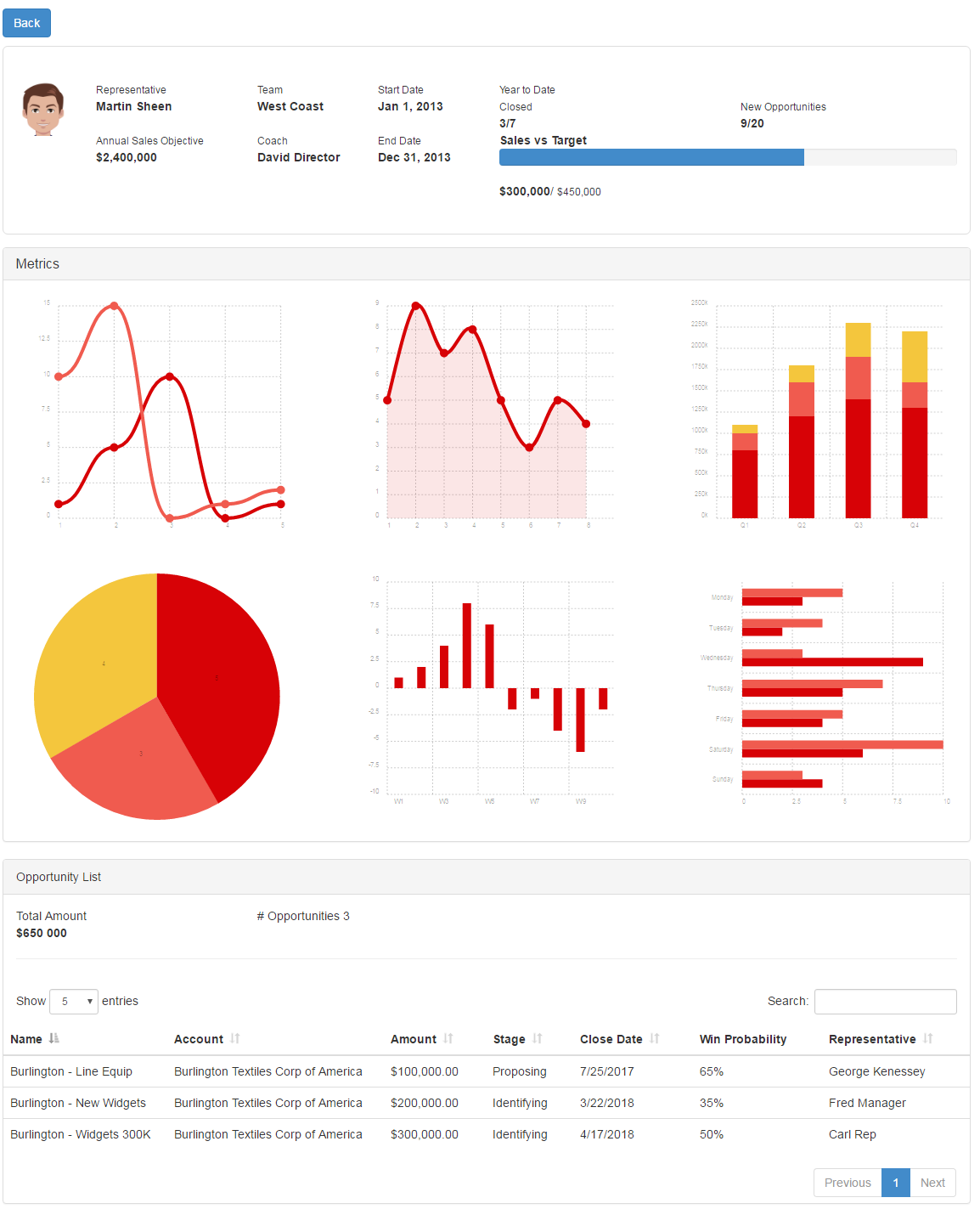


Рисунок 3.12 – Страница User

Используя аналогичные паттерны и подходы, реализуются остальные составные части приложения.

**3.9 Тестирование АСОИ**

Для тестирования web-приложения, откроем браузер и в адресной строке перейдем на локальный сервер <http://localhost:3000>. Стартовой страницей приложения является таблица со всеми пользователями (см.рисунок 3.13).

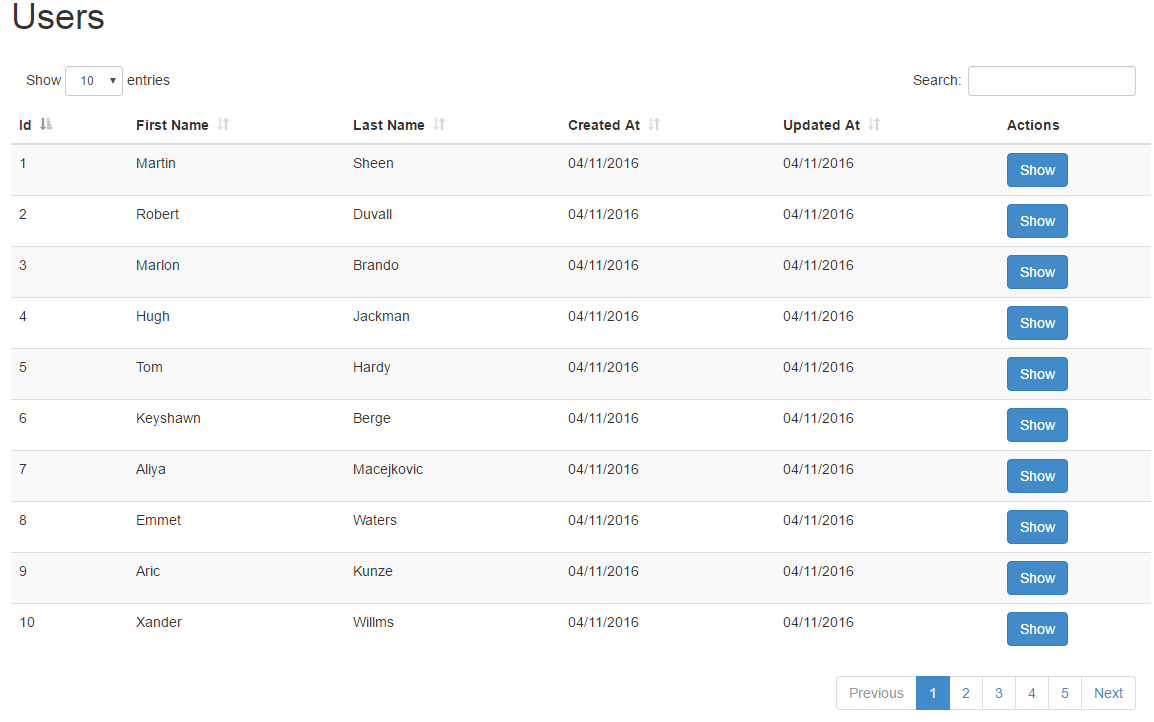


Рисунок 3.13 – Стартовая страница

Заметим, что в адресной строке, адрес изменился на <http://localhost/#/users>, что соответствует данной странице (см. рисунок 3.14).

https://i.gyazo.com/59e6ff1c434f9e656b82dee6fd79069e.png

Рисунок 3.14 – Адрес страницы пользователей

Далее протестируем переключатель одновременного вывода пользователей. Изменим значение 10, на 25 (см. рисунок 3.15).

https://i.gyazo.com/194faa5abaa9e4c31c2845ede1607980.png

Рисунок 3.15 – Вывод пользователей по 25 за раз

Как видим, последняя запись на странице имеет номер 25.

Далее проверим работу строки поиска (см. рисунок 3.16).

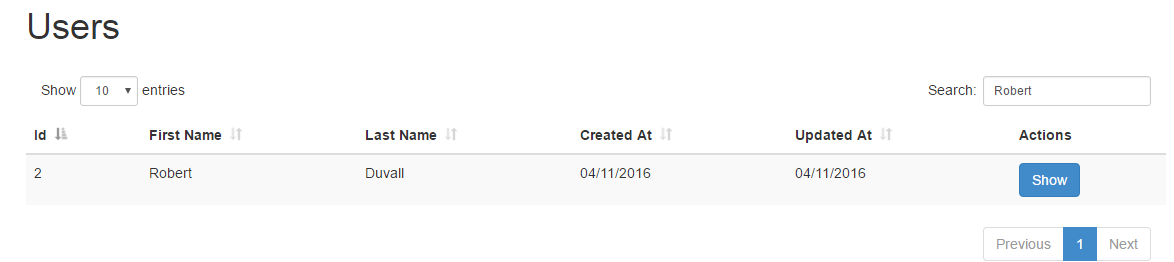


Рисунок 3.16 – Тестирование поиска

Запрос в строке поиска соответствует строке результата.

Работоспособность сортировки проверим нажатием на ячейку ID (идентификатора) (см. рисунок 3.17):

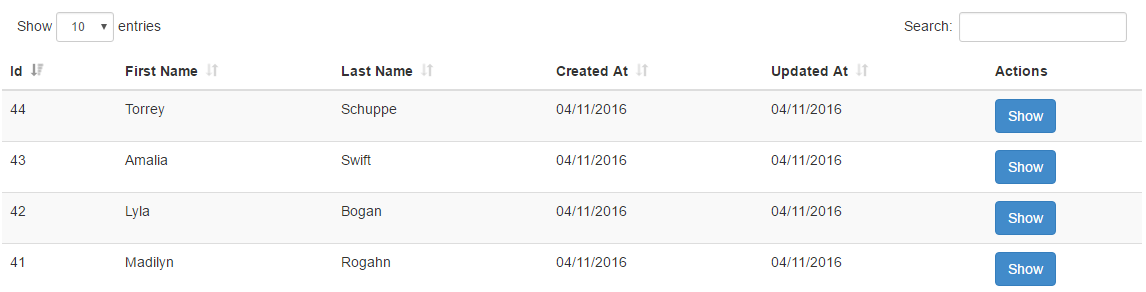


Рисунок 3.17 – Проверка сортировки

В результате сортировки, наша последняя запись с номером 44 оказалась вверху таблицы и все последующие номера идут по убыванию. Индикатор сортировки отображает верное направление.

Проверим постраничную навигацию, нажав страницу #4 (см. рисунок 3.18).

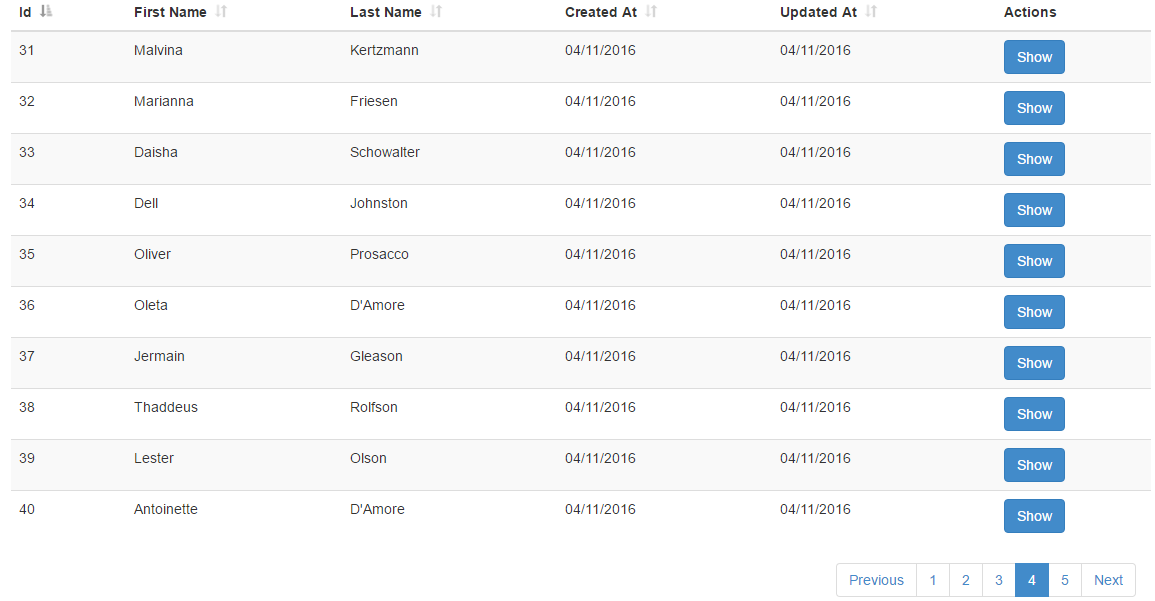


Рисунок 3.18 – Постраничная навигация

Как видим, записи начинаются с ID 31, индикатор страницы указывает на номер 4.

Проверим переход на страницу конкретного пользователя, для этого нажимаем на клавишу show (показать) пользователя #2 с именем Robert Duvall (см. рисунок 3.19).

https://i.gyazo.com/f5c5afd20e88c5d8d0a93eb8653c0229.png

Рисунок 3.19 – Переход на страницу пользователя

Как и ожидалось, в адресной строке вы видим адрес localhost:3000/#/users/user/2, что соответствует действительности.

После загрузки данных, на экране мы видим страницу пользователя #2 Robert Duvall (см. рисунок 3.20).

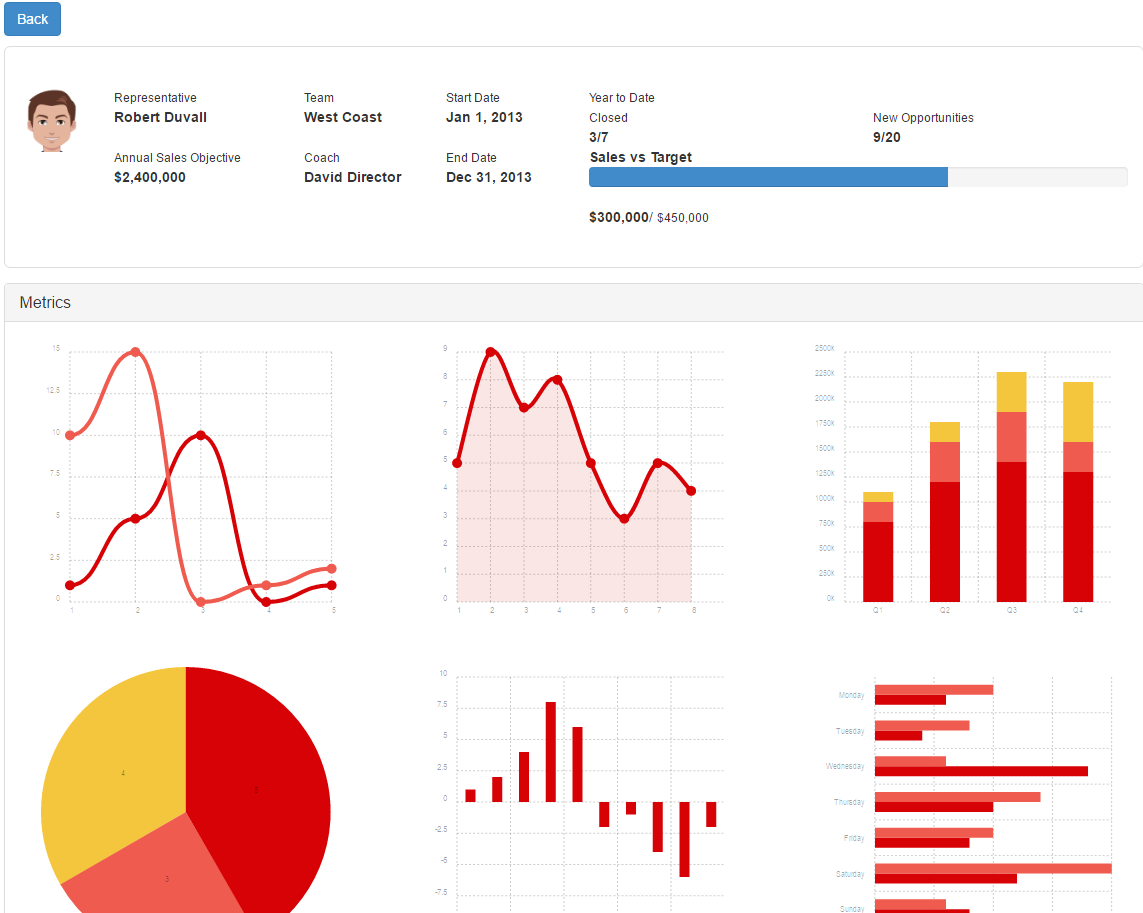


Рисунок 3.20 – Страница пользователя №2

Обращаем внимание, что в поле Representative указано верное имя.

Кнопка Back (назад), расположенная вверху страницы пользователя должна обеспечить переход на страницу всех пользователей (см. рисунок 3.21).

https://i.gyazo.com/b7990ba2148a28e08bd4ad0519d2077c.png

Рисунок 3.21 – Кнопка Back (назад)

Тестирование завершено успешно. Функционал соответствует поставленным задачам.

**4 РАСЧЕТ ЭКОНОМИЧЕСКИХ ПОКАЗАТЕЛЕЙ**

## 4.1 Расчет объема функций программного обеспечения

Технико-экономическое обоснование - это изучение экономической выгодности, анализ и расчет экономических показателей создаваемого инвестиционного проекта.

Стоимостная оценка программного средства у разработчика предполагает составление сметы затрат, которая включает следующие статьи расходов:

* заработную плату исполнителей (основную и дополнительную);
* отчисления на социальные нужды;
* материалы и комплектующие изделия;
* спецоборудование;
* машинное время;
* расходы на научные командировки;
* прочие прямые расходы;
* накладные расходы;
* затраты на освоение и сопровождение программного средства.

Прибыль — положительная разница между доходами (выручкой от реализации товаров и услуг) и затратами на производство или приобретение и сбыт этих товаров и услуг. Прибыль = Выручка − Затраты (в денежном выражении). Является важнейшим показателем финансовых результатов хозяйственной деятельности субъектов предпринимательства (организаций и предпринимателей).

Чистая прибыль – это часть валового дохода, которая остается в распоряжении предприятия после формирования фонда оплаты труда и уплаты налогов, отчислений, обязательных платежей в бюджет, в вышестоящие организации и банки. Чистая прибыль используется для стимулирования коллектива и расширения производства, а также является основным источником формирования доходов бюджета и денежных накоплений предприятия.

Общий объем ПО (Vо) определяется исходя из количества и объема функций, реализуемых программой ‑ формула (4.1):

 (4.1)

где Vi - объем отдельной функции ПО;

n - общее число функций.

Расчет общего объема ПО (количества строк исходного кода) предполагает определение объема по каждой функции.

В зависимости от организационных и технологических условий, в которых разрабатывается ПО, исполнители, по согласованию с руководством организации, могут уточнять (корректировать) объем на основе экспертных оценок.

Уточненный объем ПО (Vу) определяется по формуле (4.2):

, (4.2)

где Vуi – уточненный объем отдельной функции в строках исходного кода.

Объем программных средств (условных машинных команд) определяется на основе нормативных данных и включает функции, перечисленные в таблице 4.1.

Таблица 4.1 – Перечень и объем функций программного обеспечения

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Код функции** | **Наименование (содержание) функции** | **Объем функции строк исходного кода (LOC)** | | |
| **По каталогу Vi** | **Уточненный Vуi** | |
| 101 | Организация ввода информации | 130 | | 115 |
| 102 | Контроль, предварительная обработка и ввод информации | 480 | | 420 |
| 107 | Организация ввода/вывода информации в интерактивном режиме | 280 | | 260 |
| 303 | Обработка файлов | 1020 | | 990 |
| 305 | Формирование файла | 2130 | | 2120 |
| 501 | Монитор ПО (управление работой компонентов) | 1230 | | 1170 |
| 506 | Обработка ошибочных и сбойных ситуаций | 1530 | | 1480 |
| 601 | Проведение тестовых испытаний прикладных программ в интерактивном режиме | 3620 | | 3590 |
| Итого: | | 10420 | | 10145 |

В связи с использованием более совершенных средств автоматизации объемы функций 303, 305, 501, 601 были уменьшены и уточненный объем ПО (Vуi) составил 10145строк исходного кода вместо 10420.

## 4.2 Расчет себестоимости программного средства

Полная себестоимость (Сп) разработки программного продукта рассчитывается как сумма расходов по всем статьям. Определяется по формуле (4.3):

Сп=ЗПо+ЗПд+Рсоц+Рм+Рс+Рмв+Рнк+Рпр+Рнр+Ро+Рсо. (4.3)

где ЗПо – основная заработная плата;

ЗПд – дополнительная заработная плата;

Рсоц – отчисления на социальные нужды;

Рм – материалы и комплектующие изделия;

Рс – спецоборудование;

Рмв – машинное время;

Рнк – расходы на научные командировки;

Рпр – прочие прямые расходы;

Рнр – накладные расходы;

Ро и Рсо – затраты на освоение и сопровождение программного средства.

Основная заработная плата определяется на основании разряда, тарифной ставки и отработанного времени. Основная заработная плата исполнителей определяется за фактически отработанное время по формуле (4.4).

ЗПосн = Тст1 р \*Тк/ 22 \* Фрв \* Кпр, (4.4)

где Тст1 р – месячная тарифная ставка 1 разряда рабочего (с 1 апреля 2016 г. – 298 тыс бел. .руб.);

Тк – тарифный коэффициент согласно разряду исполнителя;

22 – среднее количество рабочих дней в месяце;

Фрв – фонд рабочего времени исполнителя (продолжительность разработки ПП, дни);

Кпр – коэффициент премий.

Дополнительная зарплата определяется по формуле (4.5).

ЗПдоп = ЗПосн \* Нд /100, (4.5)

где Нд - норматив дополнительной заработной платы.

Произведем расчет заработной платы разработчиков и результаты занесем в таблицу 5.2.

Основная заработная плата руководителя:

ЗПосн = 298 000\*3,25/22\*15\*1,3 = 858443,1818 (бел. руб.)

Дополнительная заработная плата руководителя:

ЗПдоп = 858443,1818\*15/100 = 128766,477 (бел. руб.)

Основная заработная плата программиста:

ЗПосн = 298 000\*3.04/22\*55\*1,3 = 2944240,00 (бел. руб.)

Дополнительная заработная плата программиста:

ЗПдоп = 2944240,00\*15/100 = 441636,00 (бел. руб.)

В таблице 5.2 покажем расчет заработной платы.

Таблица 4.2 ‑ Расчет заработной платы

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Категории работников** | **Разряд** | **Тарифный коэффициент (Кт)** | **Фонд рабочего времени, дни** | **Коэффициент премирования (Кпр)** | **Норматив дополнительной зарплаты, %** | **Заработная плата, бел. руб.** | | |
| **Основная** | **Дополни-**  **тельная** | **Всего** |
| Руководитель проекта | 114 | 33,25 | 115 | 11,3 | 115 | 858433,181 | 128766,477 | 987199,658 |
| Программист 1-й категории | 113 | 23,04 | 755 | 11,3 | 115 | 2944240,00 | 441636,00 | 3385876,00 |
| ИТОГО | - | - | - | - | - | 3802673,181 | 570402,477 | 4373075,658 |

Выполним расчет себестоимости ПП в виде таблицы (см. табл. 4.3).

Таблица 4.3 – Определение себестоимости ПП

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование статей затрат** | **Норматив** | **Расчетная формула** | **Сумма затрат, руб** |
| 1. Заработная плата всего  в т.ч. основная  дополнительная | **-**  -  - | **-**  -  - | 4373075,658  3802673,181  570402,477 |
| 2. Отчисления на социальные нужды | 34,6 % | Рсоц = ЗПобщ\* (34,6)/100 | 1574979,099 |
| 3. Спецоборудование | - | - | Не применялось |
| 4. Материалы | 5% | Рм = ЗПосн × 0,05 | 190133,659 |
| 5. Машинное время | 0,7 | Цмi\*  \* Нмв | 426090 |
| 6. Научные командировки | - | - | Не планировались |
| 7. Прочие затраты | - | - | Не планировались |
| 8. Накладные расходы | 50 % | Рнр = ЗПосн × 0,5 | 1901336,590 |
| 9. Сумма расходов | - | СуммЗатрат=ЗПосн+ЗПдоп+Рсоц+Рм+Рс+Рмв+ Рнк+Рпр+Рнр | 8465615,006 |
| 10. Затраты на освоение | 10% | Ро = Сумма затрат \* 0,1 | 846561,500 |
| 10. Затраты на сопровождение | 10% | Рсо = Сумма затрат \* 0,1 | 846561,500 |
| 11. Полная себестоимость | - | Сп = СуммЗатр + Ро + Рсо | 10158738,006 |

Расходы по статье «Машинное время» (Рмв) включают оплату машинного времени, необходимого для разработки и отладки ПП. Они определяются в машино-часах по нормативам на 100 строк исходного кода машинного времени в зависимости от характера решаемых задач и типа ПП. Определяется по формуле (4.6):

Рмв = Цмi \* Vo /100 \* Нмв, (4.6)

где Цмi – цена одного машино-часа, тыс. бел. руб. (можно принять 5-8 тыс бел. .руб.);

Vо – уточнённый общий объём функций строк исходного кода (LOC);

Нмв – норматив расхода машинного времени на отладку 100 строк кода, машино-часов. Принимается в размере 0,6-0,9**.**

Рмв = 6000 \* 10145/ 100 \* 0,7 = 426090 (бел. руб.)

Сумма выше перечисленных расходов на ПП служит исходной базой для расчёта затрат на освоение и сопровождение ПП. Определяется по формуле (4.7):

Суммазатрат=ЗПо+ЗПд+Рсоц+Рм+Рс+Рмв+Рнк+Рпр+Рнр. (4.7)

Затраты на освоение ПО (Ро). Организация-разработчик участвует в освоении ПП и несёт соответствующие затраты, на которые составляется смета, оплачиваемая заказчиком по договору. Для упрощения расчётов затраты на освоение определяются по установленному нормативу (Но = 5-10 %) от суммы затрат. Определяется по формуле (5.8):

Ро = Сумма затрат \*Но /100 (4.8)

Затраты на сопровождениеРсо. Для упрощения расчётов определяются по установленному нормативу (Нсо = 5-10 %) от суммы затрат. Определяется по формуле (4.9):

Рсо = Сумма затрат \* Нсо/100 (4.9)

## 4.3 Определение цены ПП и чистой прибыли

Для определения цены ПП необходимо рассчитать плановую прибыль.

Прибыль рассчитывается по формуле (4.10).

П = Сп \* R/100, (4.10)

где П – плановая прибыль от реализации ПП, руб;

R– уровень рентабельности ПП, % (можно принять в размере 10 - 30%).

После расчета прибыли от реализации определяется прогнозируемая цена ПП без налогов по формуле (4.11).

Цп = Сп +П. (4.11)

Отпускная цена (цена реализации) ПП включает налог на добавленную стоимость (в настоящее время НДС- 20 %) по формуле (4.12), НДС по формуле (4.13).

Цо = Сп + П + НДС, (4.12)

НДС = Цп \* НДС/100. (4.13)

Прибыль от реализации ПП за вычетом налога на прибыль (Пч) является чистой прибылью, остается организации разработчику и представляет собой ЭКОНОМИЧЕСКИЙ ЭФФЕКТ от создания нового программного продукта и определяется по формуле (4.14).

Пч = П \* (1- Нп/100), (4.14)

где Нп – ставка налога на прибыль (в настоящее время Нп = 18%).

Выполним расчет отпускной цены ПП и чистой прибыли в виде таблицы (см. табл. 5.4).

Таблица 4.4 – Расчет отпускной цены ПП и чистой прибыли

|  |  |  |  |
| --- | --- | --- | --- |
| **Наименование статей затрат** | **Норматив** | **Расчетная формула** | **Сумма затрат, бел. руб** |
| 1. Полная себестоимость | - | Сп = СуммЗатр + Ро + Рсо | 10158738,006 |
| 2. Прибыль | 15% | П = Сп \* R/100 | 1523810,700 |
| 3. Прогнозируемая цена ПП без налогов | - | Цп = Сп +П | 11682548,706 |
| 4. НДС | 20% | Цп\*НДС | 2336509,741 |
| 5. Отпускная цена | - | Цо = Сп + П +НДСНДС=Цп\*НДС | 14019058,477 |
| 7. Налог на прибыль | 18% | П \*Нп/100 | 274285,926 |
| 8. Чистая прибыль | - | Пч = П - Нп | 1249524,774 |

Полная себестоимость разработки программного продукта составляет 10158738,006 (руб.).

Чистая прибыль остается в распоряжении организации разработчика, представляет собой экономический эффект от создания нового программного продукта и составляет 1249524,774 (руб.). В сравнении со схожими программными продуктами полная стоимость продукта примерно соответствует аналогам, однако данный программный продукт был спроектирован с учетом специфики производственного процесса рассматриваемого предприятия, тем самым являясь предпочтительней для приобретения, нежели «усредненные» аналоги.

# 5 РЕСУРСО- И ЭНЕРГОСБЕРЕЖЕНИЕ

Разработанное web-приложение экономит человеческие ресурсы. Это обеспечивается автоматизацией большей части бизнес-процессов в виде получения, отображения и визуализации данных.

Благодаря быстродействию и оптимизации web-приложения, выполнение типичных операций занимает меньшее количество времени, что в свою очередь уменьшает количество времени работы за компьютером, тем самым экономит электроэнергию.

Грамотно построенный интерфейс уменьшает порог вхождения в приложение для новых продавцов и менеджеров.

ЗАКЛЮЧЕНИЕ

Целью данной дипломной работы была разработка приложения для обработки и визуализации статистики продаж в виде масштабируемого web-приложения. В ходе работы над проектом были пройдены следующие этапы:

* проанализированы основные подходы к проектированию масштабируемых web-приложений;
* определены инструментарии, с помощью которого возможно создание приложения;
* спроектирована архитектура собственного приложения, основные его компоненты и механизм их взаимодействия;
* реализовано и развернуто web-приложение;
* произведено тестирование приложения.

Разработанная система была выполнена с использованием передовых архитектурных решений и технологий, существующих на данный момент.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. HTML & XHTML: The Definitive Guide / C. Musciano, B. Kennedy. — 6 ed. — Sebastopol, California: O’Reilly Media 2006. — 680p.
2. CSS: The Definitive Guide / E. A. Meyer. — 3 ed. — Sebastopol, California: O’Reilly Media 2006. — 538p.
3. JavaScript: The Definitive Guide / D. Flanagan. — 6 ed. — Sebastopol, California: O’Reilly Media 2011. — 1026p.
4. The Little Book on CoffeeScript / A. MacCaw. — 1 ed. — Sebastopol, California: O’Reilly Media 2012. — 62p.
5. Syntactically Awesome Style Sheets [Электронный ресурс] / Н. Вейзенбаум, К. Эппстейн, Х. Кэтлин. — Бостон, 2007.— Режим доступа: <http://sass-lang.com/>. — Дата доступа: 15.12.2007.
6. jQuery [Электронный ресурс] / Д. Резиг. — Рочестер, 2003.— Режим доступа: <https://jquery.com/>. — Дата доступа: 07.09.2003.
7. CoffeeScript [Электронный ресурс] / Д. Ашкенас. — Нью-Йорк, 2010.— Режим доступа: <http://coffeescript.org/>. — Дата доступа: 21.02.2010.
8. Underscore.js [Электронный ресурс] / Д. Ашкенас. — Нью-Йорк, 2013.——Режим доступа: <http://underscorejs.org/>. — Дата доступа: 06.04.2013.
9. Backbone.js [Электронный ресурс] / Д. Ашкенас. — Калифорния, 2010. — Режим доступа: <http://backbonejs.org>/. — Дата доступа: 13.10.2010.
10. Marionette.js [Электронный ресурс] / М. Бриггс. — Нью-Йорк, 2009 —Режим доступа: <http://marionettejs.com>/. — Дата доступа: 11.08.2009.