**1** ОБЗОР ЛИТЕРАТУРЫ

## **1.1** Обзор существующих аналогов

(Важно! Все источники в СПИСКЕ ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ должны нумероваться в порядке упоминания в ПЗ)

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



Рисунок 1.1- Приложение «Daily Budget»

Данный проект – это система для ведения медицинского обслуживания клиентов. В ней реализована функциональность записи на прием, которая позволяет регистрировать первичных клиентов удобным способом, находить повторных, обзванивать перед визитом, а также вести расчеты с ними. Для ведения финансовых активностей существует отдельный модуль, который выполняет функции транзакций и платежей.

Также для планирования в системе предусмотрена функционал создания кабинетов, распределение врачей и ассистентов в соответствии с их графиками работы. Исходя из этого, система «ClinicIQ» полностью соответствует своим целям, но она не ориентирована на индивидуальное взаимодействие с пациентом, а скорее предназначена для управления медицинским центром. Также, к основным недостаткам системы относятся:

* отсутствие анализа и сбора статистики пациентов;
* необходимость в администрировании системы;
* отсутствие дистанционного взаимодействия пациента с доктором;
* ограниченный набор функций системы в мобильном приложении.

Еще один аналог - веб-приложение «Clinic365» (рисунок 1.2). Это удобная медицинская информационная система, которая позволяет вести базу пациентов (медицинские карточки - осмотры, анализы, диагнозы), формировать расписание приемов у врачей, учет сотрудников и зарплаты, организовывать маркетинговые акции.



Рисунок 1.2 - Веб-приложение «Clinic365»



Рисунок 1.3 - Веб-приложение «Clinic365»

Данное веб-приложение также не лишено недостатков:

* сложный пользовательский интерфейс;
* отсутствие анализа и сбора статистики пациентов;
* нет мобильного приложения.

## **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. КнигаСэммаРуби «Гибкая разработка веб-приложений»[1] содержит информацию о том, как создать простое веб-приложение на основе базы данных с нуля, используя Ruby. В книгу включены описание простейшего рабочего процесса (с использованием текстового редактора и системы контроля версий), основы технологий клиентской стороны (HTML, CSS, jQuery, Javascript), основы серверных технологий (RubyOnRails, HTTP, базы данных), основы облачного развертывания (CloudFoundry) и несколько примеров правильной практики написания кода (функции, MVC, DRY). С её помощью можно изучить фундаментальные основы языка Ruby, научиться программировать, используя объекты и массивы, а также ментальные модели, которые соответствуют этому типу разработки ПО.

Мобильное приложение – это программа, установленная и запущенная на телефоне, смартфоне и т.д. Сейчас, в связи с развитием cотовой связи и беспроводных технологий (Wi-fi, WiMax, 4G) мобильные приложения ушли далеко вперёд.

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 создаются мобильные приложения – Android, iOS, BlackBerry и WindowsMobile.

Каналы распространения: специализированный портал – AppStore, AndroidMarket, MicrosoftStore, самостоятельный поиск и 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» [2].

Особенность данной архитектуры заключается в том, 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. Передача запросов и результатов их обработки происходит через Интернет.



Рисунок 1.4 - Архитектура клиент-сервер

Отображением результатов запросов, а также приемом данных от клиента и их передачей на сервер обычно занимается специальное приложение – браузер (InternetExplorer, Mozilla, Opera и т. д.). Как известно, 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), который и 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, выполняемым веб-приложениями, относятся:

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

Основными достоинствами архитектуры «клиент-сервер» являются:

* возможность, в большинстве случаев, распределить функции вычислительной системы между несколькими независимыми компьютерами в сети, что позволяет упростить обслуживание вычислительной системы;
* все данные хранятся на сервере, который, как правило, защищён гораздо лучше большинства клиентов;
* на сервере проще обеспечить контроль полномочий, чтобы разрешать доступ к данным только клиентам с соответствующими правами доступа;
* позволяет объединить различные клиенты;
* использовать ресурсы одного сервера часто могут клиенты с разными аппаратными платформами, операционными системами и т.п.

Среди недостатков можно выделить необходимость квалифицированного профессионала для администрирования данной системы. В случае использования централизованной системы, неработоспособность основного сервера может сделать неработоспособным всё приложение. Также, 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 [3].



Рисунок 1.5 - REST-сервис

Примером такой системы может служить WorldWideWeb. В REST определяется строгое разделение ответственности между компонентами клиент-серверной системы, облегчающее реализацию необходимых актеров (actors). Другой целью REST является упрощение семантики взаимодействия компонентов сетевых систем, что позволяет улучшить масштабируемость и повысить производительность. В основу REST заложен принцип автономности запросов, означающий, что запросы, обрабатываемые клиентом или сервером, должны включать всю контекстную информацию, необходимую для их понимания.

При работе REST-систем для обмена данными стандартных медиа-типов 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 RubyOnRails-приложений, используя самые разнообразные технологии. Архитектура REST отличается своей простотой, требуя от приложений обеспечить только возможность приема сообщений с HTTP- заголовками. Эта функция легко реализуется простыми контроллерами в RubyOnRailsMVC.

## **1.5** Шаблон проектирования MVC

MVC (Model-View-Controller, «модель-представление-контроллер») – схема использования шаблонов проектирования, с помощью которых модель приложения, пользовательский интерфейс и взаимодействие с пользователем разделены на три отдельных компонента таким образом, чтобы модификация одного из компонентов оказывала минимальное воздействие на остальные.

Впервые паттерн MVC появился в языке SmallTalk. Разработчики должны были придумать архитектурное решение, которое позволяло бы отделить графический интерфейс от бизнес логики, а бизнес логику от данных.

Популярность данной структуры в веб приложениях сложилась благодаря её включению в две среды разработки, которые стали очень популярными: Struts и RubyonRails. Эти две среды разработки наметили пути развития для сотен рабочих сред, созданных позже.

Идея, которая лежит в основе конструкционного шаблона MVC, очень проста: нужно чётко разделять ответственность за различное функционирование в наших приложениях. Приложение разделяется на три основных компонента, каждый из которых отвечает за различные задачи (принцип единой ответственности).

Контроллер управляет запросами пользователя (получаемые в виде запросов HTTP GET или POST, когда пользователь нажимает на элементы интерфейса для выполнения различных действий). Его основная функция – вызывать и координировать действие необходимых ресурсов и объектов, нужных для выполнения действий, задаваемых пользователем. Обычно контроллер вызывает соответствующую модель для задачи и выбирает подходящий вид.

Модель – это данные и правила, которые используются для работы с данными, которые представляют концепцию управления приложением. В любом приложении вся структура моделируется как данные, которые обрабатываются определённым образом. Что такое пользователь для приложения – сообщение или книга? Только данные, которые должны быть обработаны в соответствии с правилами (дата не может указывать в будущее, email должен быть в определённом формате, имя не может быть длиннее Х символов, и так далее).



Рисунок 1.6 - Шаблон проектирования MVC

Модель даёт контроллеру представление данных, которые запросил пользователь (сообщение, страницу книги, фотоальбом, и тому подобное). Модель данных будет одинаковой, вне зависимости от того, как мы хотим представлять их пользователю. Поэтому мы выбираем любой доступный вид для отображения данных.

Модель содержит наиболее важную часть логики нашего приложения, логики, которая решает задачу, с которой мы имеем дело (форум, магазин, банк, и тому подобное). Контроллер содержит в основном организационную логику для самого приложения (очень похоже на ведение домашнего хозяйства).

Стоит отметить, что в данном случае описан подход с «толстой» моделью и «тонким» контроллером. Очень часто практикуется подход наоборот – «тонкая» модель и «толстый» контроллер – когда бизнес-логика заключена в контроллере, а модель является лишь данными.

Вид обеспечивает различные способы представления данных, которые получены из модели. Он может быть шаблоном, который заполняется данными. Может быть несколько различных видов, и контроллер выбирает, какой подходит наилучшим образом для текущей ситуации.

Веб-приложение обычно состоит из набора контроллеров, моделей и видов. Контроллер может быть устроен как основной, который получает все запросы и вызывает другие контроллеры для выполнения действий в зависимости от ситуации.

Самое очевидное преимущество, которое мы получаем от использования концепции MVC – это чёткое разделение логики представления (интерфейса пользователя) и логики приложения.

Поддержка различных типов пользователей, которые используют различные типы устройств является общей проблемой наших дней. Предоставляемый интерфейс должен различаться, если запрос приходит с персонального компьютера или с мобильного телефона. Модель возвращает одинаковые данные, единственное различие заключается в том, что контроллер выбирает различные виды для вывода данных.

Помимо изолирования представления от логики приложения, концепция MVC существенно уменьшает сложность больших приложений. Код получается гораздо более структурированным, и, тем самым, облегчается поддержка, тестирование и повторное использование решений.