1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор существующих аналогов

На этапе проектирования комплекса были тщательно изучены существующие аналоги. Одним из наиболее приближенных является камера «D-Link DCS-931L» (рисунок 1.1).

Рисунок 1.1- Камера «D-Link DCS-931L» [1]

Данная беспроводная сетевая камера представляет собой полноценную систему со встроенным процессором и веб-сервером, передающим высококачественное видеоизображение. Эта IP-камера предоставляет возможности удаленного управления, обнаружения движения и звука для комплексного и эффективного решения домашней безопасности.

Входящее в комплект поставки ПО позволяет создать полноценную систему видеонаблюдения, включая запись по срабатыванию датчиков, вручную или по расписанию. Поддержка камерой сервиса mydlink позволяет всегда иметь удаленный доступ к камере через веб-браузер. К основным недостаткам системы относятся:

- нестабильная связь через Wi-Fi;
- нет поддержки карты памяти;
- низкое качество снимков при плохом освещении;
- высокая стоимость.

Еще один аналог - камера «Mio VixCam C10» (рисунок 1.2). Это еще одна беспроводная IP-камера с возможностью записи снимков на карту памяти, сетевой накопитель или облачный сервис. Также это устройство обеспечивает достойное качество снимков в условиях плохого освещения.

Рисунок 1.2 - Камера «Mio VixCam C10» [2]

Данная камера также не лишена недостатков:
- отсутствие удаленного доступа к камере;
- отсутствие проводных Ethernet и USB интерфейсов.

1.2 Обзор плат Raspberry Pi

Raspberry Pi – представляет собой одноплатный компьютер размером с банковскую карту. Изначально он разрабатывался как бюджетная система для обучения информатике, затем нашедший более широкое применение и популярность, чем предполагали его авторы. Данная платформа разрабатывается Raspberry Pi Foundation.

В микрокомпьютерах Raspberry Pi применяются ARM процессоры с частотой тактирования от 700 МГц и до 1,4 ГГц. У плат этого семейства есть такие интерфейсы как USB, Bluetooth, Wi-Fi, Ethernet, HDMI. Присутствует

графическое ядро с поддержкой аппаратного ускорения. Raspberry Pi работает в основном на операционных системах, основанных на Linux ядре.

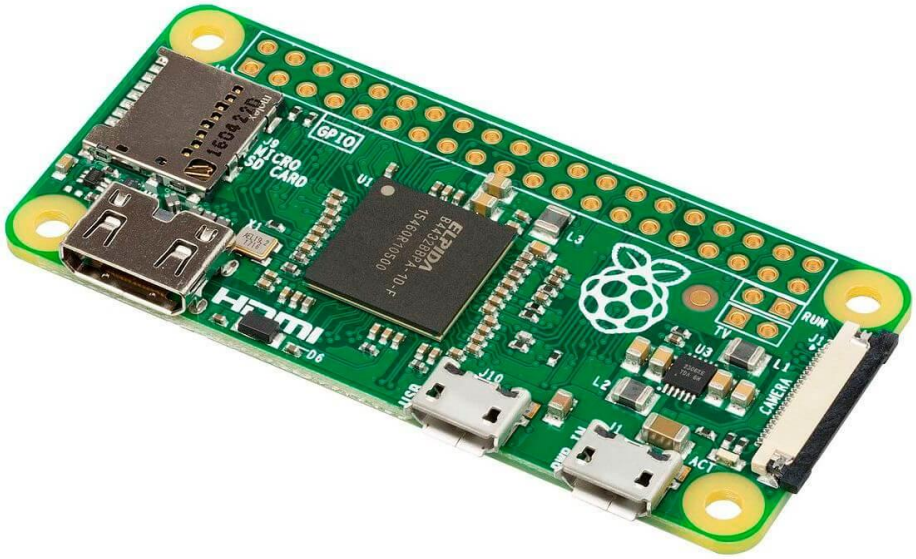

Рисунок 1.3 – Raspberry Pi Zero W [3]

Платформа включает себя 9 основных плат с разными характеристиками и размерами: A, A+, B, B+, 2B, Zero, 3B, Zero W, 3B+. Zero W представлена на рисунке 1.3.

1.3 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, а обмен информацией происходит через сеть. Преимуществом такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения являются кроссплатформенными сервисами. Книга «Гибкая разработка веб-приложений в среде Rails» [4] предоставляет информацию о том, как создать простое веб-приложение, используя Ruby, на основе базы данных с нуля. 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.

Управляющее приложение – это программа, которая будет определять поведение нашей платы и подключенных к ней модулей, а также осуществлять передачу данных на сервер.

1.4 Архитектура клиент-сервер

Архитектура является одним из важнейших аспектов в разработке веб-приложения. Наиболее эффективную работу приложений обеспечивает архитектура «клиент-сервер» [5].

Особенностью данной архитектуры является то, что само веб-приложение находится и выполняется на сервере, клиенту при этом доступны только результаты работы. Работа приложения основана на получении запросов от пользователя (клиента), их обработке и выдаче результата. Передача запросов и результатов их обработки происходит через Интернет.

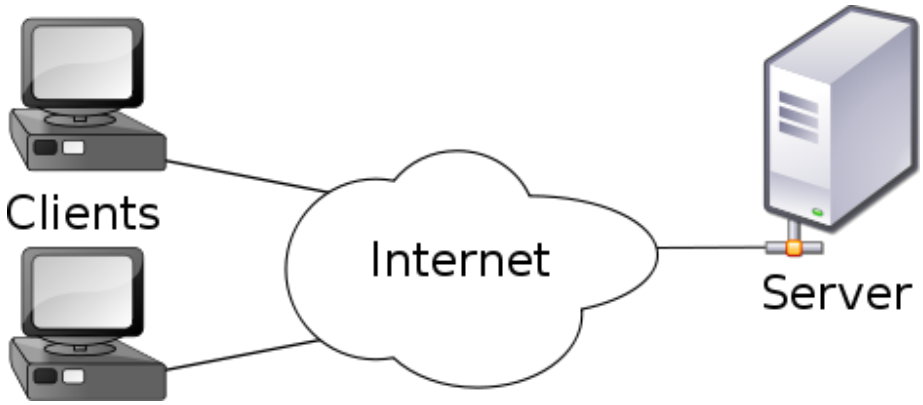

Рисунок 1.4 - Архитектура клиент-сервер

Отображением результатов запросов, а также приемом данных от клиента и их передачей на сервер обычно занимается специальное приложение – браузер (InternetExplorer, Mozilla, Opera, Chrome и т. д.). Функцией браузера является отображение данных, полученных из Интернета, в виде страницы, описанной на языке HTML. Таким образом, результат, передаваемый от сервера к клиенту, также должен быть представлен на этом языке.

На стороне сервера веб-приложение выполняется специальным программным обеспечением (веб-сервером), который принимает запросы, обрабатывает их, формирует ответ в виде страницы, описанной на языке

HTML и передает его клиенту.

При обработке запроса пользователя веб-приложение формирует ответ на основе исполнения программного кода, работающего на стороне сервера, веб-формы, страницы HTML, другого содержимого, включая графические файлы.

Как было сказано ранее, в результате формируется HTML-страница, которая отправляется клиенту. Таким образом, 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. Из-за наличия исполняемой части, веб-приложения способны выполнять практически те же операции, что и обычные Windows-приложения, с тем лишь отличием, что код исполняется на сервере, роль интерфейса системы исполняет 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, обычно, защищён гораздо лучше большинства клиентов;

– на сервере легче 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 сотрудника для администрирования данной системы. В случае использования централизованной системы, неработоспособность основного сервера может сделать неработоспособным всё приложение. Также, важным фактором является высокая стоимость оборудования.

1.5 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 [6].

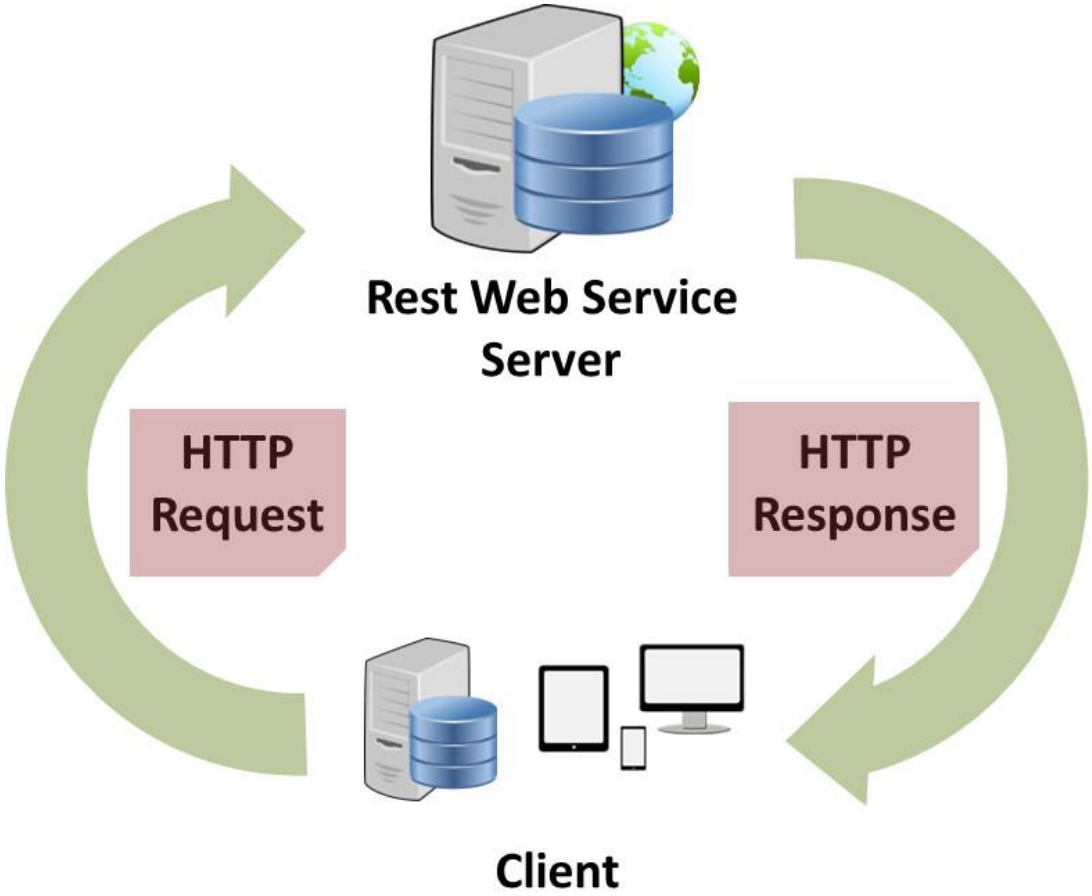

Рисунок 1.5 - REST-сервис

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. Данный принцип означает, что запросы, которые обрабатываются клиентом или сервером, обязаны 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 нескольких последних лет разработчики создавали REST-сервисы для своих

RubyOnRails-приложений, используя самые разнообразные технологии. Архитектура REST отличается своей простотой, требуя от приложений возможности приема сообщений с HTTP-заголовками. Эта функция достаточно просто реализуется простыми контроллерами в RubyOnRails MVC.

1.6 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. Разработчикам было необходимо придумать архитектурное решение, которое позволило бы разделить графический интерфейс, бизнес логику и данныу.

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 представление данных и правила, которые используются для работы с ними. Они 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 X символов, и так далее).

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).

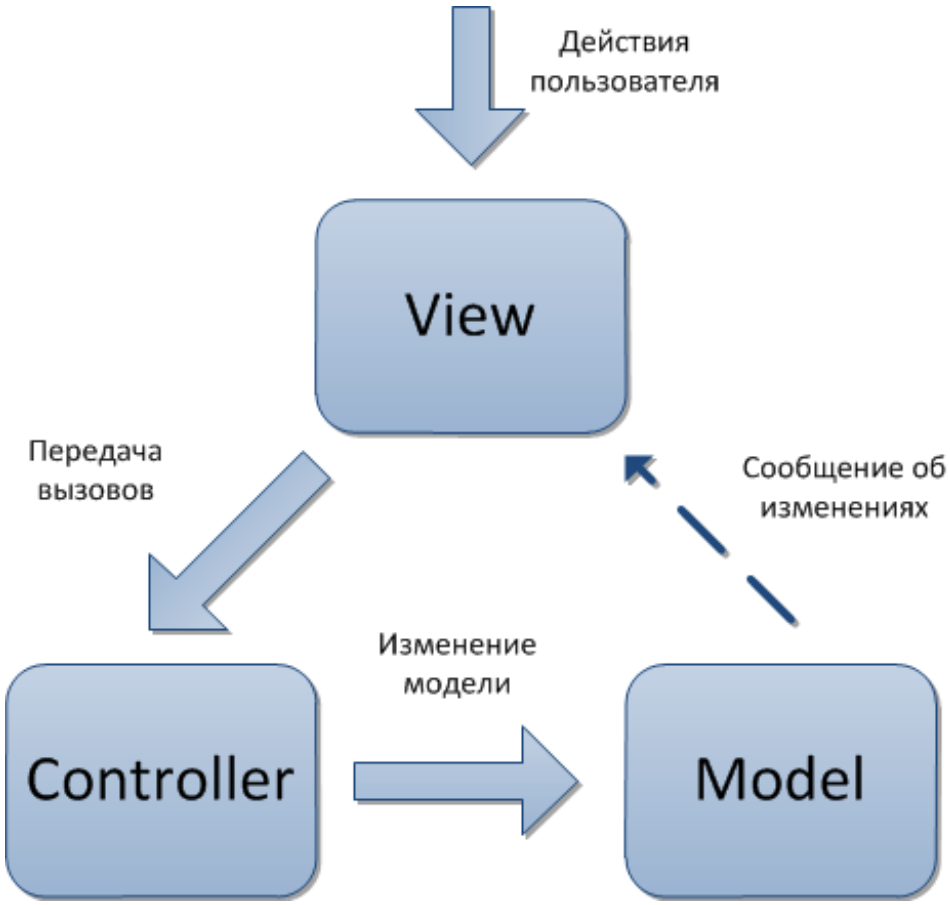

Рисунок 1.6 - Шаблон проектирования MVC

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.