СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 6](#_Toc483261697)

[**1** ОБЗОР ЛИТЕРАТУРЫ 8](#_Toc483261698)

[**1.1** Обзор аналогов 8](#_Toc483261699)

[**1.2** Обоснование выбора используемого инструментария 12](#_Toc483261700)

[**1.2.1** Ruby on Rails 12](#_Toc483261701)

[**1.2.2** Стек технологий 17](#_Toc483261702)

[**1.2.3** AngularJS 18](#_Toc483261703)

[**2** СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ 20](#_Toc483261704)

[**3** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 25](#_Toc483261705)

[**3.1** Модели и их связи в базе данных 25](#_Toc483261706)

[**3.2** Состояния 32](#_Toc483261707)

[**3.3** Директивы 38](#_Toc483261708)

[**3.4** Сервисы 49](#_Toc483261709)

[**4** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ 52](#_Toc483261710)

[**4.1** Подключение устройства 52](#_Toc483261711)

[**4.2** Управление элементом устройства 55](#_Toc483261712)

[**4.3** Запуск сценария 56](#_Toc483261713)

[**4.4** Остановка сценария 57](#_Toc483261714)

[**4.5** Создание сценария через wizard 58](#_Toc483261715)

[**5** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ 62](#_Toc483261716)

[**5.1** Применение Rspec 62](#_Toc483261717)

[**5.1** Тестирование приложения 64](#_Toc483261718)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 68](#_Toc483261719)

[ПРИЛОЖЕНИЕ А 69](#_Toc483261720)

[ПРИЛОЖЕНИЕ Б 70](#_Toc483261721)

# ВВЕДЕНИЕ

Данная дипломная работа посвящена проектированию веб-сервиса для удаленного доступа к программно-аппаратной платформе домашней автоматизации.

Развитие домашней автоматизации начиналось от одного компьютера в комнате до сети вычислительных устройств расположенных по всему дому. Многие из них – устройства, непосредственно общающиеся с пользователем, такие как планшеты, ноутбуки и т.д. Однако всё больше и больше домашняя сеть начинает содержать устройств специального назначения – «вещи» – что превращает домашнюю сеть в сеть не только для пользователей, но и для своих устройств – сеть «вещей».

С начала 2010-х годов в результате повсеместного распространения беспроводных сетей, появления облачных вычислений, освоения программно-конфигурируемых сетей и развития технологий межмашинного взаимодействия (Machine-to-Machine) начинается системное внедрение практических идей «интернета вещей» в сфере ИТ.

Новое понятие Internet - Internet of Things (IoT) - это существующая сеть Интернет, расширенная подключенными к ней вычислительными сетями различных устройств, физических предметов или вещей, которые могут самостоятельно организовывать разнообразные модели подключения или общения («Thing – Thing», «Thing – User» и «Thing - Web Object»).

Термином «вещь» в IoT обозначаются интеллектуальные, т.е. «умные» предметы или объекты, к которым относятся датчики или приводы, снабженные микроконтроллером с ОС реального времени со стеком протоколов, памятью и устройством связи, встроенные в различные объекты, например, в электросчетчики, газовые счетчики, счетчики потребления холодной и горячей воды, датчики давления, вибрации или температуры и т.д.

«Умные» объекты могут быть организованы в вычислительную сеть физических устройств, подключенных через шлюзы (хабы или специализированные IoT платформы) к традиционной сети Интернет. Сегодня эти сети слабо связанны между собой и разрозненны, каждая из них решает свою специфическую задачу. По мере того, как Интернет вещей будет развиваться, эти и многие другие сети будут подключаться друг к другу и приобретать все более широкие возможности по самоуправлению.

На технологическом уровне IoT – это способ развития инфраструктуры сети (физической основы) Интернет, в которой «умные» вещи самостоятельно, без участия человека, подключаются к сети для удаленного взаимодействия с другими устройствами (Thing - Thing) или взаимодействия с автономными или облачными ЦОДами или Data-центрами (Thing - Web Objects) с целью передачи данных на хранение, обработку и анализ данных, принятия управленческих решений, направленных на изменение окружающей среды, а также с целью взаимодействия с пользовательскими терминалами (Thing - User) для контроля и управления этими устройствами.

Концепция облачных вычислений возникла в 2006 году. Amazon.com, в то время книжный интернет-магазин, представил Amazon Web Services (AWS), положив начало движению облачных вычислений.

AWS объединяет широкий набор сервисов, таких как вычислительные мощности и хранилища данных. Впоследствии к Amazon.com присоединились Netflix, Microsoft, Google, Apple и IBM, образовав обширный рынок облачных вычислений.

Задача данного программного проекта заключена в реализации контроля всех подключённых устройств с единого, дружественного пользователю web-интерфейса в реальном времени. Поддержке общения между устройствами и решение ими определенных повседневных задач без участия человека. Предоставление пользователю возможности гибкой настройки системы под свои потребности. Подразумевается, что пользователь выбирает тип управляемых устройств среди поддерживаемых системой. Тип устройства определяет конкретные задачи, выполняемые им.

Основные возможности:

* управление различными типами устройств;
* веб-доступ с любого устройства в глобальной сети;
* редактор сценариев работы устройств;
* web-интерфейс с обновлением в реальном времени;
* push-уведомления;
* интеграция со сторонними веб-сервисами (сервис погоды);
* модель безопасности с разграничением доступа между пользователями;
* просмотр статистики работы устройств;
* пользователю доступны данные всех устройств без их сохранения в облаке;
* синхронизация состояния устройств, управляемых непосредственно и через web-интерфейс;
* plug and play (PnP) авто определение новых подключённых устройств.

Web-сервис представляет собой сервер-клиентское приложение, в котором сервер находится на облачной PaaS (Platform as a Service) платформе Heroku. Клиентская часть выполняется в браузере. Клиент выполняет запросы на сторонние сервисы, такие как сервис погоды openWeatherMap, а также устанавливает соединение с домашним сервером автоматизации, через который происходит управление устройствами и от которого приходят уведомления о состоянии системы в реальном времени.

Техгологии – RoR (Ruby on Rails), AngularJS. Приложение имеет REST (Representation state transfer) архитектурный стиль. RoR реализует паттерн MVC (Model-View-Controller), AngularJS – MVW (Model-View-Whatever).

# **1** ОБЗОР ЛИТЕРАТУРЫ

## Обзор аналогов

При проектировании системы были изучены наиболее популярные аналоги.

«MajorDoMo» – это программное обеспечение (ПО), позволяющее компьютеру выполнять функции контроллера домашней автоматики (см. рисунок 1.1).



Рисунок 1.1 – Система «MajorDoMo»

Основные особенности системы – это создание сценариев, GPS-трекинг и реакция на местоположение пользователя, голосовые уведомления и распознание голоса, управление мультимедиа, маркет дополнений, анализ состояния и самодиагностика.

К основным недостаткам системы относятся:

* для полноценной работы требует компьютер с 2 Gb оперативной памяти;
* после установки система требует настройку, возможны конфликты с приложением Skype;
* для программирования скриптов необходимо знание базового синтаксиса PHP, а также списка функций, реализованных в «MajorDoMo» для работы с автоматикой;
* доступ только в локальной сети.

«Home Assistant» – open-source платформа для домашней автоматизации. Платформа запускается либо на Raspberry Pi3 либо на компьютере (см. рисунок 1.2).



Рисунок 1.2 – Система «Home Assistant»

Основные особенности системы – это работа с устройствами различных производителей, автоматическое обнаружение устройств с последующим отображением в основной группе «Home», frontend построен на библиотеке Polymer, предоставляет WebSocket API и RESTful API.

К основным недостаткам системы относятся:

* для запуска, подключения и настройки компонентов, выключателей, сценариев, групп, триггеров пользователь должен знать YAML;
* для доступа к системе не с глобальной сети необходимо настроить port forwarding на роутере с использованием Dynamic DNS сервисов, также пользователь должен самостоятельно настроить шифрование.

«ThinkingHome» – это платформа домашней автоматизации на .NET (см. рисунок 1.3). Система позволяет работать с устройствами от разных производителей благодаря системе плагинов. Она позволяет реализовывать новый функционал в виде отдельных независимых модулей. Пользователь самостоятельно может написать плагин, при этом не заботясь о решении базовых задач.



Рисунок 1.3 – Система «ThinkingHome»

При помощи плагинов можно расширять список вероятных событий в системе. В основном вся функциональность системы как раз и обеспечивается системой плагинов. В основном сервис содержит только инфраструктуру для их работы, обеспечивает загрузку и организацию жизненного цикла, предоставляет возможность работы с БД и систему логирования. Веб-интерфейс доступен с любого устройства в домашней сети (компьютера, смартфона, планшета). Он может отображать информацию о прогнозе погоды, расписании автобусов и т.д. Через web-интерфейс доступно как ручное управление домом – управление устройствами со смартфона или планшета, так и управление через сценарии – автоматическое выполнение действий при наступлении определённых событий. Система имеет возможность работать с устройствами от разных производителей. Стандартные плагины осуществляют поддержку MQTT протокола, микроклимата, прогноза погоды, таймера, nooLite. Таким образом пользователю доступны: система плагинов, API логирования, API хранения данных в БД, инфраструктура HTTP API, инфраструктура веб-интерфейса, клиент-серверная шина сообщений, API локализации.

К основным недостаткам системы относятся:

* создание сценариев на ЯП JavaScript;
* система реализована на .NET (доступна только на Windows OS);
* скудный набор готовых плагинов;
* доступ только с устройств в домашней сети.

«OpenHub» – open source система автоматизации на Java (см. рисунок 1.4). Система позволяет интегрировать разные системы и технологии домашней автоматизации в одно единственное решение, которое позволяет накладывать правила автоматизации и предоставляет единообразный пользовательский интерфейс. «OpenHub» использует Eclipse SmartHome framework. Он написан полностью на Java и использует Apache Karaf вместе с Eclipse Equinox как Open Services Gateway Initiative (OSGi) runtime и связывает с Jetty как HTTP сервер.



Рисунок 1.4 – Система «OpenHub»

Основные особенности системы: архитектура не зависит от производителя устройств, независима от аппаратного обеспечения и протоколов, единый пользовательский интерфейс, удалённое управление и уведомления, триггеры, запускаемые событиями или таймером, механизм плагинов, сценарии.

Основные недостатки:

* нет механизма plug and play, требует технических навыков в программировании, электрике и пайке;
* упрощённый интерфейс.

«The Thing System» – это комплекс программных компонентов и сетевых протоколов для домашней автоматизации. «Домашний» сервер в нем называется steward. Steward написан на node.js и запускается как на обычном компьютере, так и на одноплатном, таком как Raspberry Pi. Дизайн и архитектура системы подчиняется правилу, что IoT должен вращаться вокруг правил, по которым «вещи» следят за событиями, в ответ на которые могут выполнять задачи. Не каждый выбор, опцию нужно предоставлять пользователю. IoT должен быть контекстно зависим, должен действовать на упреждение, а не реагировать в ответ. Таким образом Steward позиционируется как система автоматизации, хотя и предоставляет возможность удалённого контроля. Основной недостаток системы в том, что она не имеет визуализации (предоставляет только API).

## Обоснование выбора используемого инструментария

Данный проект архитектурно состоит из двух частей: backend и frontend. На стороне backend-а используется Ruby on Rails framework. На стороне, frontend-а используется AngularJS framework. Рассмотрим особенности данных технологий.

### **1.2.1** Ruby on Rails

Ruby on Rails (или коротко Rails) – это framework для веб разработки, написанный на ЯП Ruby. С появления в 2004 году, Ruby on Rails стремительно набрал популярность и стал одним из мощнейших инструментов для построения динамических веб-приложений. Ruby on Rails используется такими знаменитыми компаниями как: Airbnb, Basecamp, Disney, GitHub, Hulu, Kickstarter, Shopify, Twitter, и The Yellow Pages [1].

Преимущества Ruby on Rails:

1. Это полностью open-source проект, доступный по MIT License, в результате чего является бесплатным для скачивания и использования.
2. Реализация Model-View-Controller (MVC) паттерн для веб-приложений.
3. Обеспечение их интеграции с веб-сервером и сервером баз данных.
4. Использование REST-стиля построения веб-приложений.
5. Применение в разработке приложений следующих принципов: максимальное повторное использование кода (принцип Don’t repeat yourself); использование соглашений по умолчанию по конфигурации (принцип Convention over configuration), при котором явная спецификация конфигураций требуется только в нестандартных случаях [2].
6. Элегантный и компактный дизайн Rails, способствующий большой популярности. Используя податливый нижележащий ЯП Ruby, Rails фактически создает предметно-ориентированный язык (domain-specific language) для написания веб-приложений. В результате много общих задач веб-программирования – таких как генерирование HTML, создание моделей данных и маршрутизация URI – легко решаемы с Rails, а итоговый код программ получается кратким и выразительным.
7. Быстрая адаптация к новым веяниям в веб-технологиях. Например, в Rails одним из первых был полностью реализовал архитектурный стиль REST для веб-приложений. Создатель Rails, David Heinemeier Hansson и рабочая группа Rails используют эти новые идеи также при создании другими фреймворками новых техник. Наиболее ярким примером является слияние Rails и Merb (конкурирующая веб-платформа), так что Rails теперь получает преимущества от модульной конструкции Merb, стабильного API, а также повышенной производительности.
8. Увлечённое и разнообразное сообщество пользователей Rails, сотни open-source разработчиков, многолюдных конференций, форумов и каналов IRC (Internet Relay Chat), огромное количество гемов, богатый набор информативных блогов. Большое количество активных программистов Rails также облегчает обработку неизбежных ошибок приложений: алгоритм – «Ищи в Google сообщение об ошибке» – почти всегда добывает соответствующее сообщение в блоге или ветке форума [2].

MVC состоит из объектов трех видов:

* модель - объект приложения;
* вид - экранное представление;
* контроллер - описывает, как интерфейс реагирует на управляющие действия пользователя.

Rails накладывает значительные ограничения на структурирование веб-приложений, которые заметно упрощают создание приложений. Rails навязывает структуру для приложения — модели, представления и контроллеры разрабатываются как отдельные функциональные блоки, a Rails при выполнении заданной программы связывает их вместе. Отличительной особенностью Rails является то, что процесс увязки базируется на использовании разумных умолчаний, которые, как правило, избавляют от написания каких-либо внешних конфигурационных метаданных, обеспечивающих взаимную работу. Приоритет соглашения над конфигурацией является примером концепции Rails [3].

Модель в Ruby on Rails предоставляет остальным компонентам приложения объектно-ориентированное отображение данных. Объекты модели могут осуществлять загрузку и сохранение данных в реляционной базе данных, при сохранении данных в которую попутно производятся проверки валидации. Модели реализуют бизнес-логику приложения.

Для хранения объектов модели в реляционной СУБД по умолчанию в Rails используется библиотека ActiveRecord.

Представление создаёт пользовательский интерфейс с использованием полученных от контроллера данных. Представление также передает запросы пользователя на манипуляцию данными в контроллер.

Контроллер в Rails — это набор логики, который запускается после получения HTTP-запроса сервером. Контроллер отвечает за вызов методов модели и запускает формирование представления.

В Rails-приложении входящий запрос сначала посылается маршрутизатору, который решает, в какое место приложения должен быть отправлен запрос и как должен быть произведен синтаксический разбор этого запроса. В результате на данном этапе где-то в коде контроллера идентифицируется конкретный метод (называемый в Rails действием). Действие может искать запрошенные данные, может взаимодействовать с моделью и может вызвать другое действие. В результате выполнения действие подготавливает информацию для представления, которое создает отображение для пользователя. В случае работы Rails в API режиме действие подготавливает JSON файл с данными [3].

Схема MVC в Rails на рисунке:



Рисунок 1.5 – Паттерн MVC

Исходя из архитектуры, построенной на MVC, RoR использует три компонента:

* ActiveRecord;
* ActionView;
* ActionController.

Сочетание последних двух известно, как Action Pack. Рассмотрим эти компоненты.

Active Record – это Модель в RoR. Модель хранит данные и предоставляет базу для работы с данными. Кроме этого Active Record также является ORM фрэймворком. ORM значит Object-relational mapping (Объектно-реляционная проекция). Собственно, Active Record делает следующие вещи:

1. Проекция таблицы на класс. Каждая таблица проецируется на один или несколько классов по принципу convention over configuration (соглашение выше конфигурации). Одно из таких соглашений – имя таблицы должно быть во множественном числе, а название класса – в единственном. Атрибуты таблицы налету проецируются в атрибуты экземпляра Руби. После того, как все проекции сделаны, каждый объект ORM класса представляет определенную строку таблицы, с которой класс был спроецирован.
2. Соединение с БД. Вы можете подключиться к базе данных, используя API, предоставляемый Active Record, который создает необходимый вам запрос непосредственно в движок БД при помощи адаптеров. У Active Record есть адаптеры для MySQL, Postgres, MS SQLServer, DB2, и SQLite. Необходимо лишь записать параметры доступа к БД в файле database.yml.
3. Операции CRUD. Это операции create (создание), retrieve (получение), update (обновление) и delete (удаление) над таблицей. Так как Active Record – это ORM фрэймворк, вы всегда работаете с объектами. Чтобы создать новую строку таблицы, вы создаете новый объект класса и заполняете его переменные экземпляра значениями. Стоит заметить, что все это Active Record делает за вас.
4. Проверка данных. Проверка данных перед помещением их в таблицу – это первый шаг в безопасности вашего проекта. Active Record предоставляет проверку Модели. Данные могут быть проверены автоматически с помощью множества готовых методов, которые, в случае необходимости, можно переписать под собственные нужды.

ActionView – это вид. Он включает в себя логику, необходимую для вывода данных Модели. Представление в Rails отвечает за создание полного или частичного ответа, отображаемого в браузере, обработанного приложением или посланного в виде электронной почты. В простейшем виде представление является фрагментом HTML-кода, отображающего какой-нибудь неизменный текст. Но чаще всего вам потребуется включить динамическое содержимое, созданное методом действия в контроллере [4]. Наиболее часто используемые функции Action View:

1. Шаблоны (Templates). Шаблоны – это файлы, содержащие заполнители (placeholders), которые буду заменены на контент. Шаблоны могут содержать HTML-код и код Ruby, встраиваемый в HTML с использованием синтакса встроенного (embedded) Ruby (ERb).
2. Помощники (helper, далее хелпер) форм и форматирования. Хелперы форм позволяют создавать такие элементы страниц, как чекбоксы, списки, используя готовые методы. В свою очередь хелперы форматирования позволяют форматировать данные необходимым нам способом, методы существуют для дат, валют и строк.
3. Макет. Макеты (layouts) определяют, как контент будет расположен на странице. Динамически создаваемая страница может содержать вложение из нескольких страниц, даже без использования таблиц и фрэймов, используя API Макета.

Action Controller. В веб-приложении контроллер регулирует поток логики приложения. Он находится на границе программы, перехватывая все запросы, на основе которых он изменяет какой-то объект модели и вызывает вид, чтобы отобразить обновленные данные. В RoR Action Controller является контроллером, вот его основные функции:

1. Поддержка сессий. Сессия – это период времени, проведенный пользователем на сайте. Его можно отследить с помощью cookie или объекта сессии. Cookie – небольшой файл, он не может содержать объекты, в отличие от объекта сессии.
2. Фильтрация. Бывают ситуации, когда необходимо вызвать определенный код, перед тем как исполнять логику Контроллера или после него, например, аутентификация пользователей, логирование событий, предоставление персонального ответа. Помогают в таких случаях фильтры, предоставляемые Action Controller. Существуют три основных фильтра: before, after и around. О них – позже.
3. Кэширование. Кэширование – это процесс, при котором наиболее запрашиваемый контент сохраняется в кэше, чтобы не было необходимости запрашивать его вновь и вновь.

Среды. RoR поощряет использование отдельных сред для каждого из этапов цикла жизни приложения: разработка (development), тестирование (testing) и эксплуатация (production), для каждого из которых создается отдельная БД. Рассмотрим каждую среду.

1. development. В среде разработки ставка делается на немедленное отображение нового варианта при изменении кода – достаточно обновить страницу в браузере. Скорость в этой среде не важна. Когда случается ошибка, она выводится на экран.
2. test. При тестировании мы обычно каждый раз наполняем БД каким-нибудь глупым текстом, чтобы убедиться, что нормальное поведение не зависит от содержания БД. Процедуры юнит-тестинга и теста функциональности в RoR автоматизированы и производятся через консоль. Тестовая среда предоставляет отдельное пространство, в которых оперируют эти процедуры. Для тестирования существуют DSL языки, такие как Rspec, Capybara, Cucumber. C их помощью можно выполнять тестирование моделей (модульное), тестирование контроллеров и acceptance-тестирование (интеграционное).
3. production. В конце концов ваше приложение выходит к финальной черте, пройдя тесты и избавившись от багов. Теперь обновления кода будут происходить редко и можно сконцентрироваться на производительности, включить кэширование. Нет необходимости писать огромные логи ошибок и пугать пользователей сообщениями об этих ошибках в браузере. Для вас – среда production.

Вокруг Rails сложилась большая экосистема плагинов, которые также называются «джемы» (gem с англ. — «самоцвет»). Для управлений 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 [5, 6].

### **1.2.2** Стек технологий

Существует много веб-приложений (особенно те, что написаны на Ruby on Rails) построенных при помощи слоёной архитектуре, которая часто называется *стек*, потому, что диаграммы обычно отображают слои как сложенные блоки (см. рисунок 1.2).

Rails представляет середину стека и является middleware. Rails – это место, где находится основная логика приложения. Дно стека – хранилище данных – место, где сохраняется значимая информация приложения. Это обычно реляционная система управления Relational Database Management System (RDBMS). Вершиной стека является пользовательский интерфейс. В веб-приложении он реализуется HTML, CSS и JavaScript выполняемый в браузере [5, 6].



Рисунок 1.6 – Обобщённый cтек технологий

Стек, построенный на выбранных технологиях выглядит так (см. рисунок 1.3):



Рисунок 1.7 – Стек технологий

PostgreSQL – это open-source SQL база данных, выпущенная в 1997 году. Она поддерживает множество продвинутых опций, которых нет в других популярных open-source базах данных таких как MySQL или коммерчески базах, таких как Microsoft SQL Server.

PostgreSQL позволяет создавать очень сложные ограничения. Например, можно потребовать, чтобы email пользователя был из определенного домена, чтобы штат в U.S. адресе был написан точно, как две буквы в верхнем регистре, или даже чтобы штат в адресе уже был в списке разрешенных государственных кодов. Это же можно сделать и при помощи Rails, но выполнение на уровне базы данных означает, что ни баг в коде, ни существующий скрипт, ни разработчик в консоли, ни программа не саможет поместить невалидную информацию в базу данных.

PostgreSQL поддерживает перечисляемые типы, массивы и словари (называются HSTOREs). Во многих базах данных необходимо иметь раздельные таблицы для таких структур данных.

Postgres поддерживает JSON тип данных, позволяя сохранять произвольную информацию в столбце. Это означает, что можно использовать Postgres в качестве хранилища для документов или сохранять данные, которые не соответствуют схеме. Используя JSONB тип данных, вы убеждаетесь, что JSON поля могут быть индексируемы также, как и структурированные поля таблицы.

### **1.2.3** AngularJS

AngularJS – это JavaScript MVC framework, созданный и поддерживаемый Google. Angular позиционирует себя как Model-View-Whatever framework, в нашем случае Whatever - это контроллер (см. рисунок 1.4). Angular воспринимает view не как статический кусок HTML, а как полномасштабное приложение. Angular предоставляет мощные средства по организации кода и позволяет структурировать разметку для создания выразительного, тестируемого, управляемого frontend кода [7].



Рисунок 1.8 – MVC в AngularJS

Angular помогает чисто разделить код и представление. Angular организует frontend как приложение со своими собственными путями, контроллерами и представлениями. Это упрощает frontend и позволяет легко организовать JavaScript код.

Чистое, декларативное представление. Angular представление – это просто HTML. Angular добавляет специальные атрибуты, называемые директивами, которые позволяют чисто соединить данные и функции с разметкой. Нет необходимости встраивать код или скрипты, существует чистое разделение между представлением и кодом.

Директивы в Angular служат для добавления новой функциональности html-разметке. Связь моделей и отображения идет именно через директивы. С помощью директив AngularJS можно создавать пользовательские HTML-теги и атрибуты, чтобы добавить поведение некоторым элементам. Angular имеет набор встроенный директив:

1. Ng-app. Объявление корневого элемента приложения.
2. Ng-bind. Служит для автоматического замещения текста HTML-элемента на значение переданного выражения.
3. Ng-model. Похож на ng-bind, только обеспечивает двустороннее связывание данных. Если изменится содержимое элемента, то и изменится значение модели. Изменения в модели отображаются на содержимое элемента.
4. Ng-class. Позволяет динамически загружать css-классы.
5. Ng-controller. Присоединяет контроллер к участку DOM.
6. Ng-view. Базовая директива, отвечает за обработку маршрутов.

Angular предоставляет двустороннее связывание данных. Сервис $scope следит за изменениями в модели и изменяет раздел HTML-выражения в представлении через контроллер. Кроме того, любые изменения в представлении отражаются в модели. Это позволяет обойти необходимость манипулирования DOM.

Angular контроллер – это JavaScript объект, создаваемый посредством стандартного конструктора.

Сервисы AngularJS представляют специальные объекты или функции, выполняющие некоторые общие для всего приложения задачи. В AngularJS имеюся ряд встроенных сервисов, такие как $http, $q и ряд других. Кроме того, имеется возможность создавать свои сервисы для выполнения специфических задач. Чтобы создать сервис, нам надо использовать метод factory. Данный метод, как говорит название, представляет фабрику для создания сервиса. Причем сервис реализует паттерн синглтон, что значит, что для всего приложения будет создан только один экземпляр, и из любого места приложения будет идти обращение к этому единичному объекту.

Angular с самого начала поддерживал unit-тестирование JavaScript кода.

Angular имеет большую экосистему компонентов и модулей благодаря своей популярности. Множество типичных вопросов имеют решение в экосистеме Angular [7].

## 

# **2** СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Системы домашней автоматизации, интернета вещей очень популярны на данный момент. Данный проект является интерфейсом пользователя к программно-аппаратной платформе удаленного управления устройствами.

Пользователь, заходит на домашнюю страницу, откуда может посмотреть краткую информацию о проекте, имеет возможность перейти на страницу регистрации.

В процессе регистрации помимо почты и пароля пользователь может опционально ввести идентификационный номер домашнего сервера автоматизации. Проводятся валидации полей формы, пользователь оповещается о неправильно заполненных полях и ему предоставляется возможность исправить эти поля и отправить форму заново.

Если пользователь зарегистрирован, то он видит dashboard проекта, на котором находятся карточки со статистикой:

* кол-во подключенных датчиков в системе;
* кол-во логических областей, которым принадлежат датчики;
* кол-во пользовательских скриптов;
* кол-во скриптов находящихся на выполнении;
* кол-во новоподключённых устройств.

Постоянными элементами интерфейса являются Navbar и Vertical Navigation.

Navbar содержит следующие элементы слева направо:

1. Hamburger Menu. Иконка меню служит для открытия и скрытия вертикальной навигационной панели.
2. Logo. SVG изображение.
3. Application Title. Содержит имя продукта
4. Notification Icon. Через нее доступна Notification Drawer панель критических событий сервера. Это самодостаточная система, которая может быть просмотрена без необходимости перехода на другие страницы приложения. На самой иконке находится badge, отображающий кол-во новых уведомлений с домашнего сервера автоматизации.
5. Help Icon. При его нажатии появляется выпадающее меню с обязательной опцией «About», которая запускает модальное окно с информацией о продукте.
6. User Icon. Показывает имя зарегистрированного пользователя. По нажатии на нее появляется выпадающее меню с обязательной опцией «Logout».

Vertical Navigation – это глобальная навигационная панель, отображаемая по левой стороне страницы. Вертикальная навигационная панель имеет до трех уровней вложенности. На ней расположены ссылки на страницу скриптов, страницу логических областей, или выбрать страницу конкретной логической области, содержащей датчики, также присутствует ссылка на dashboard.

Страница логических областей (areas). Датчики, подсоединённые к системе, делятся на логические области, т.е. они принадлежат конкретной области. Логические области служат только лишь для удобной группировки устройств и не отражают физического строения распределённой системы устройств. Логическая область – это удобная абстракция, позволяющая абстрагироваться от деления устройств, например, по местонахождению (комната, дом, и т.д.). Вместо этого название и, возможно, описание area позволяет применять систему в более общих ситуациях автоматизации, не привязываясь к определённым понятиям. Таким образом, area имеет уникальное имя, задаваемое пользователем при создании. Можно редактировать имя и описание логической области. Также доступны все CRUD операции над ней. Датчик, подсоединенный к системе принадлежит к area по умолчанию. Пользователь может менять принадлежность датчика к area в любое время. При удалении area, датчики, принадлежащие к нему, переносятся в area по умолчанию.

Страница логической области (area). Эта страница содержит список датчиков, принадлежащих конкретной логической области. Над датчиками можно производить действие по перенесению их в другие логические области. Для каждого датчика помимо имени и типа отображается некоторая уточняющая информация. Для каждого датчика в списке есть ссылка на индивидуальную страницу датчика.

Страница датчика. Шаблоны страниц датчика зависят от типа датчика. Со страницы датчика производится управление им. Имеется возможность просмотреть полную информацию о нем. Посмотреть, в каких скриптах он задействован. Добавление новых устройств происходит при подсоединении их к системе, вся информация о них передается через домашний сервер автоматизации. Новое устройство помещается изначально в логическую область по умолчанию, откуда может быть перенесено в любую логическую область. Устройство удаляется при отсоединения его от системы.

Notification Drawer для критических событий сервера автоматизации. При первом запросе к домашнему серверу автоматизации клиенту передается список критических событий, таких как появление нового устройства в системе, ошибка в устройстве, ошибка выполнения скрипта. Вместе с информацией в критическом сообщении показывается метка времени – когда это событие произошло.

Страница скриптов. Показывает список всех скриптов. Можно создать скрипт. Создание скрипта основано на wizard-е. Каждый шаг в процессе создания скрипта влияет на действия, возможные в дальнейшем. После прохода всех шагов, wizard-а генерируется скрипт, который сохраняется в базе данных и передается на сервер автоматизации. Имеется возможность запустить отдельный скрипт. При запуске скрипта выполняется попытка его выполнения на сервере автоматизации, а состояние выполнения доступно для отслеживания пользователю. Выполнение скрипта – это процесс, которым пользователь может управлять через интерфейс.

Страница Действий (actions). Содержит список действий, инициализированных пользователем, таких как управление отдельным датчиком или запуск скрипта и показывает результат или состояние действий. У действия имеется метка времени его запуска и окончания.

Страница аккаунта. Содержит данные пользователя, а также идентификатор сервера автоматизации. Имеется возможность редактировать эти данные. Основная информация может быть заполнена через форму регистрации.

Toast Notifications. Тост-уведомления показываются в верхнем правом углу приложения. Они служат для показа происходящих событий в реальном времени. Эти уведомления пропадают с течением времени. Они не блокируют информацию, находящуюся за ними и отображаются достаточное время, чтобы пользователь успел прочитать сообщение. Это уведомление не пропадает, если пользователь «завис» над ним.

*Блок пользовательского интерфейса*. Пользовательский интерфейс организован как Single page application (SPA) – единственный HTML-документ используется в качестве оболочки всех веб-страниц и организует взаимодействие с пользователем через динамически подгружаемые HTML, CSS, JavaScript посредством AJAX и WebSocket. SPA приложение передаёт весь необходимый код JavaScript (модули, виджеты, контроллеры) вместе с загрузкой самой страницы. SPA-приложение типичный представитель HTML5. SPA-приложения работают на большом количестве устройств (компьютеры, планшеты, смартфоны). SPA-приложения имеют богатый и насыщенный пользовательский интерфейс, так как веб-страница одна. Намного проще хранить информацию о сеансе, управлять состояниями представлений и управлять анимацией. Некоторые функции, такие как routing переносятся со стороны backend-а на клиентскую, что позволяет не обращаться с запросом по каждой странице, а запрашивать только необходимые данные. Также при заполнении форм, валидации выполняются как на backend-е, так и на frontend-е. AngularJS адоптирован для поддержки SPA принципов. AngularJS фреймворк для клиентской стороны. Angular использует двустороннее связывание данных в пользовательском интерфейсе (UI), связывая UI-элементы м моделью. Для двустороннего связывания Angular применяет паттерн Наблюдатель. Двустороннее связывание позволяет автоматически обновлять представления, как только изменяется модель и наоборот. В традиционном подходе - генерировании HTML на стороне сервера контроллер и модель взаимодействуют внутри процесса на сервере для генерации HTML представлений. В приложении, использующем AngularJS контроллер и модель находятся у клиента в браузере, поэтому новые страницы могут быть сгенерированы без какого-либо взаимодействия с сервером. Angular использует технологию AJAX. Преимущественно используется XMLHttpRequest объект в JavaScript, который предоставляет возможность делать HTTP-запросы из JavaScript на сервер без перезагрузки данных. Результатом запросы к серверу является сырые данные в формате JSON или XML или же новая HTML страница. В случае возвращения HTML как ответа сервером JavaScript на стороне клиента обновляет частичный участок Document Object Model (DOM). В случае прихода сырых данных JavaScript на стороне клиента обычно генерирует из сырых данных HTML, который затем используется для обновления частичного участка DOM.

*Блок связи с сервером автоматизации* расположен на клиентской стороне (frontend). Он выполняет связь с домашним сервером автоматизации по полнодуплексному протоколу WebSocket. Для получения url адреса сервера автоматизации пользователю необходимо зарегистрировать сервер автоматизации в приложении. Он способен это сделать непосредственно во время регистрации или позже, зайдя в настройки аккаунта. Именно этот блок инициализирует REST операции по добавлению нового устройства и другие команды пришедшие с сервера автоматизации.

*Блок обновления состояний устройств*. Данный блок реализует поведенческий паттерн «Посредник». Он обеспечивает взаимодействие множества объектов, находящихся на стороне сервера автоматизации со множеством соответствующих объектов на стороне данного веб-приложения. При этом получается слабая связанность и устройства избавляются от необходимости явно ссылаться друг на друга. Команда о подключении нового устройства подаётся через блок связи с сервером автоматизации. Информация о новом устройстве заносится в базу данных. Домашний сервер автоматизации шлёт информацию о состоянии каждого устройства при подключении клиента к серверу и при каждом изменении состояния устройства. При управлении устройствами из пользовательского интерфейса в блок обновления состояния устройств приходит реакция с домашнего сервера. Также страница действий содержит текущую информацию о состоянии устройств.

*Блок удаленного управления устройствами*. Этот блок непосредственно связан с блоком пользовательского интерфейса. Его логика связана с пользовательскими действиями над устройствами. Он формирует новое состояние устройства на основе пользовательских действий в интерфейсе. Существует необходимость двусторонней связи между веб-сервисом и домашним сервером автоматизации, так как пользователю необходимо понять, выполнена ли его команда с удалённым устройством или нет. Для этого, после передачи команды серверу автоматизации веб-сервис ждёт ответ с результатом выполнения команды. Ответ принимается асинхронно, чтобы создать иллюзию немедленного выполнения пользовательской команды. В случае неудачи пользователь оповещается о нештатной ситуации. Передаёт новое состояние устройства в блок связи с сервером автоматизации.

*Блок аутентификации пользователя и регистрации сервера автоматизации*. Так как веб-сервис предоставляет доступ к конфиденциальной информации (показания датчиков и устройств), а также к управлению данными устройствами, то строгая аутентификация является актуальным вопросом. Базовая аутентификация – включение имени пользователя и пароля в состав HTTP POST запроса не подходит для данного приложения, так как любой, кто перехватит пакет узнает секретную информацию, поэтому использована дайджест-аутентификация. Методика заключается запросе у пользователя пароля, возможно с подтверждением и сохранения зашифрованной версии пароля в базе. Сравнение зашифрованных паролей вместо непосредственного сравнения даёт дополнительное преимущество – есть возможность аутентифицировать пользователей без хранения в базе данных самих паролей, тем самым избегается проблема системы безопасности приложения. Производит действия с формой при регистрации пользователя. Блок выполняет валидации полей формы.

*Блок формирования и управления скриптами*. Скрипты позволяют пользователю определить последовательность выполняемых действий, которые будут выполнены при запуске скрипта. Таким образом, скрипты – удобный механизм автоматизации управления. Аналоги представляют скрипты в разной форме, например, описание скрипта в виде YAML формате. Этот вариант является довольно сложным и запутанным для использования пользователем. Вариант с написанием пользователем кода на существующих скриптовых языках, таких как javascript, php и т.д. также не является оптимальным, так как требует хотя бы какого-то знакомства с программированием. Поэтому был выбран вариант создания скрипта через выполнение пользователем пошаговых действий в wizard-е. Этот подход достаточно прост для пользователя. Он скрывает внутреннюю реализацию сценария и тем самым сокращает время, необходимое на его создание. Данный подход также достаточно гибок, так как wizard предоставляет возможность разбить определение сложного сценария на более простые компоненты и может предоставлять пользователю разные шаги в зависимости от решений пользователя на предыдущих шагах wizard-а. Сформированный скрипт отправляется в базу данных и передаётся на домашний сервер автоматизации.

*Блок обработки критических сообщений с домашнего сервера*. Критические сообщения могут влиять на любую часть пользовательского интерфейса. Они помещаются в Notification Drawer. Также с ними связана логика, при таких асинхронных событиях, приходящих с сервера автоматизации как удаление устройства из системы, сбои в работе устройств, сбои в работе скриптов и т.д.

*Блок обработки данных с сервиса прогноза погоды*. Данные с сервиса могут отображаться на widget-е, а также пользователь может просматривать погоду в конкретном городе. Блок использует API сервиса погоды.

*База данных*. Хранит данные о пользователях, о домашних серверах автоматизации и скрипты, логические области, данные о датчиках.

Структурная схема, иллюстрирующая перечисленные блоки и связи

между ними приведена на чертеже ГУИР. 400201.003 C1.

# **3** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описывается функционирование web-приложения. Рассматриваются основные объекты, потоки данных и алгоритмы приложения на основе модулей, выделенных в структурном проектировании.

Описание системы производится, начиная от основных блоков приложения, заканчивая точным описанием задач отдельных частей приложения, их алгоритмов и данных.

Логика web-приложения в основном находится в angular-контроллерах на стороне frontend-а. Контроллеры и модели на стороне backend-а в основном служат для работы с бд.

## **3.1** Модели и их связи в базе данных

**3.1.1** user

Данная модель хранит такие данные о пользователе как логин, пароль, почта. Сервис предоставляет возможность выполнять регистрацию, вход и выход из аккаунта. Пользователь при регистрации обязан ввести токен сервера автоматизации, чтобы иметь к нему доступ и получать данные с датчиков. Модели пользователя принадлежат:

* сервера автоматизации (automation\_servers);
* логические области (areas);
* устройства (devices);
* сценарии (scropts);
* сообщения (notifications);
* тикеты (tickets);
* комментарии (comments);

Т.е. каждый пользователь имеет доступ только к своим моделям.

Модель пользователя имеет следующие поля:

* email - почта пользователя;
* encrypted\_password - пароль пользователя;
* reset\_password\_token- токен переустановки пароля
* reset\_password\_sent\_at-время посылки письма для переустановки пароля;
* remember\_created\_at- время создания пользователя;
* sign\_in\_count- кол-во раз пользователь заходил;
* current\_sign\_in\_ip-ip-адрес, с которого выполнен вход пользователя;
* created\_at, время создания модели пользователя;

**3.1.2** area

Данная модель представляет из себя логическую область, которая является контейнером для устройств. Устройство возможно переносить из одной области в другую. Логические области могут представлять из себя помещения, комнаты, или любые иные понятия, по которым пользователю удобно структурировать его устройства. Понятие логическая область выделяется только на web-сервисе, т.е. остальная система не придерживается логического структурирования, но использует физическое (подключение устройств к одному контроллеру, например).

Модель имеет следующие поля:

* name, имя логической области;
* default - флаг, определяющий, является ли данная логическая область областью по умолчанию;
* user\_id – внешний ключ пользователя, владеющего данной областью, связывает с таблицей user;
* created\_at- время создания модели логической области;
* updated\_at- время последнего изменения модели области;

**3.1.3** automation\_server

Модель хранит данные о сервере автоматизации, принадлежащем пользователю. Для установления связи web-приложением, сервер автоматизации обязан получить доменное имя. Доменное имя может изменяться при перезагрузке сервера, так как он использует сторонний сервис для преодоления Network Address Translation (NAT). Поэтому при каждом обновлении доменного имени автономный сервер отправляет на сервер web-приложения POST-запрос в контроллер automation\_server\_controller, в котором передает новый url. Идентификация сервера автоматизации происходит при помощи уникального токена token. Данные о сервере обновляются в базе. Web-приложение готово для получения и отправки данных на сервера автоматизации.

Модель для сервера автоматизации имеет следующие поля:

* url- url сервера автоматизации, может меняться;
* token- идентификационный токен сервера автоматизации;
* user\_id- внешний ключ, связывающий модель с пользователем user;
* created\_at- время создания модели сервера автоматизации;
* updated\_at- время последнего изменения модели сервера;

**3.1.4** device

Модель устройства хранит данные о физическом устройстве, например, таком, как лампочка или привод. Само по себе устройство представляет собой композицию элементов управления (controls), которые в свою очередь и осуществляют управление отдельными элементами устройства.

Таким образом, device также является контейнером, только уже для элементов управления. В отличие от area, понятие device активно используется остальной частью системы (сервером автоматизации и распределенной системой микроконтроллеров). Устройство имеет адрес dev\_id, по которому можно получить доступ с как с сервера автоматизации, так и с web-приложения. Устройства делятся на типы при помощи поля label. Тип устройства служит только для корректного отображения на странице, так как устройства отличаются лишь набором стандартных элементов управления (controls). Представления устройства в виде контейнера для шаблонных элементов управления позволяет быстро внедрять новые типы устройств в системы. Со стороны же web-приложения никаких дополнительных действий или конфигураций производить не нужно. Новые устройства подключаются PnP. Имя устройства name необходимо только для отображения пользователю и пользователь может легко его редактировать. Режим редактирования имени устройства доступен по двойному щелчку на имени, т.е. для редактирования используется механизм Edit-in-place или click-to-edit. Вид устройства и работы данного механизма продемонстрирована на рисунке.

Модель устройства имеет следующие поля:

* dev\_id- уникальный адрес устройства в системе;
* name- имя устройства;
* label- тип устойства;
* area\_id-внешний ключ, связывающий устройство с логической областью;
* user\_id-внешний ключ, связывающий устройство с пользователем;
* updated\_at-время последнего изменения модели устройства;
* created\_at- время создания модели устройства;

**3.1.5** Модель control

Модель служит для представления элемента управления, принадлежащего устройству. Поле type содержит вид элемента управления.

В системе используются следующие виды:

* toggle- простой переключатель;
* dimmer-элеменнт управления в виде диммера, имеет значения в процентах;
* switch\_state- сложный переключатель, имеет более двух состояний;
* num\_value- простое цифровое значение;
* sym\_value-простое символьное значение;

Текущие положения элементов управления не сохраняются в базе, а приходят с сервера автоматизации. Это сделано в целях увеличения быстродействия сервиса и безопасности.

Модель имеет следующие поля:

* name- имя элемента управления;
* ctrl\_id- идентификатор элемента управления в системе;
* type- тип элемента;
* updated\_at-время последнего изменения модели элемента;
* created\_at- время создания модели элемента;

**3.1.6** Модель geolocation

Данная модель хранит информацию о местоположении сервера автоматизации. Поля lon и lat используются для отображения местонахождения сервера на yandex-карте.

Модель имеет следующие поля:

* country-страна, в которой находится сервер автоматизации;
* country\_code- код страны;
* lat- широта сервера;
* lon- долгота сервера;
* region- код региона;
* region\_name- имя региона;
* timezone- часовой пояс;
* created\_at- время создания модели геолокации;
* updated\_at- время последнего изменения модели устройства;

**3.1.7** Модель notification

Модель служит для хранения информации о событиях в целой системе, как произошедших на сервере автоматизации или в системе микроконтроллеров, так и пользовательских событий в web-приложении. Поэтому, для различия источников происхождения, сообщения делятся по полю origin. В системе выделяются два значения поля origin:

* user-сообщение сгенерировано непосредственно в web-приложении;
* automation\_server-сообщение сгенерировано на сервере автоматизации;

Сообщения показываются в отдельной закладке, у каждого сообщения есть метка времени created\_at, которая позволяет проследить хронологию возникновения событий.

Модель имеет следующие поля:

* category-тип сообщения;
* details-непосредственно информация, передаваемая сообщением;
* origin-источник возникновения сообщения;
* user\_id-внешний ключ, связывающий с моделью user;
* read-флаг, устанавливаемый после просмотра сообщения;
* created\_at- время создания модели сообщения;
* updated\_at- время последнего изменения модели сообщения;

Тип сообщения (category) могут быть следующими:

* error-сообщение с таким типом возникает при возникновении ошибки в системе;
* warning-сообщение с таким типом возникает при возникновении неправильных значений в устройстве;
* info- общая информация, сообщения такого типа возникают при правильном ходе выполнения программы;

**3.1.8** Модель script

Модель содержит всю необходимую информацию о сценарии. Сценарии создаются с помощью wizard-а в web-приложении. Для выполнения сценарии передаются на сервер автоматизации. Сценарии имеют специально разработанный DSL язык. Текст программы на данном языке храниться в поле code.

Модель имеет следующие поля:

* name-имя сценария;
* code-исходный код сценария;
* user\_id-внешний ключ, соединяющий с моделью пользователя;
* start\_time-время запуска сценария на сервере автоматизации;
* status-статус сценария;
* created\_at- время создания модели сценария;
* updated\_at- время последнего изменения модели сценария;

Статус сценария модет принимать следующие значения:

* stopped-сценарий не выполняется;
* running- сценарий выполняется на сервере автоматизации;

**3.1.9** Модель ticket

Модель содержит данные о вопросе, заданном пользователем по работе системы. Любой пользователь может задать интересующий его вопрос и получить ответы от других пользователей либо из поддержки.

Модель содержит следующие поля:

* user\_id-внешний ключ, соединяющий с модель с пользователем user;
* title-заголовок тикета;
* content- тело тикета;
* tags-тэги тикета;
* open- флаг, определяющий, открыт тикет или закрыт;
* created\_at- время создания модели тикета;
* updated\_at- время последнего изменения модели тикета;

**3.1.10** Модель comment

Модель служит для хранения информации о комментарии на тикет.

Модель содержит следующие поля:

* user\_id-внешний ключ, соединяющий с пользователем;
* ticket\_id- внешний ключ, соединяющий с тикетом, на который был написан данный ответ;
* content-тело комментария;
* updated\_at-время последнего изменения модели комментария;
* created\_at- время создания модели комментария;

**3.1.11** Модель weather\_map

Модель хранит данные о стороннем сервисе погоды OpenWeatherMap.

Для получения данных о погоде необходимо иметь api-key. Api-key для данного сервиса предоставляется бесплатно. Бесплатный тарифный план подразумевает:

* до 60 обращений в минуту;
* api для получения текущей погоды;
* прогноз на 5 дней;
* данные по загрязнению воздуха;
* погодные предупреждения;
* доступность 95 %;
* обновление данных менее чем каждые два часа;
* поддержка только текущей версии api;

Модель имеет следующие поля:

* user\_id-внешний ключ, соединяющий с пользователем;
* appid- api-key для обращения к сервису;
* updated\_at-время последнего изменения модели;
* created\_at- время создания модели;

**3.1.12** Модель yandex\_maps

Модель хранит данные, необходимые для обращения к сервису Yandex Maps. Для использования API Yandex Maps не обязательно иметь api-key, но его наличие предоставляет дополнительные бонусы от использования сервиса.

Модель имеет следующие поля:

* user\_id-внешний ключ, соединяющий с пользователем;
* apikey- api-key для обращения к сервису;
* updated\_at-время последнего изменения модели;
* created\_at- время создания модели;

**3.1.13** Модель control\_settings

Модель служит для хранения информации о настройках сбора статистики отдельного элемента управления. Эти настройки доступны пользователю для редактирования.

Модель содержит следующие поля:

* control\_id-идентификатор элемента управления в системе;
* sampling\_frequency- частота выборки данных элемента для статистики;
* storage\_period- период сбора статистики;
* max\_samples- максимальное кол-во статистических записей;
* drop\_duplicates- флаг, определяющий, отбрасывать ли повторяющиеся значения для статистики;
* created\_at- время создания модели;
* updated\_at- время последнего изменения модели;

Частота выборки данных (sampling\_frequency) может принимать следующие значения:

* realtime-статистические данные берутся с каждого нового показания элемента управления;
* minutely- запись статистических данные происходит раз в минуту, значение усредняется;
* hourly- статистические данные заносятся каждый час, значения усредняются;
* daily- статистические данные заносятся каждый день, значения усредняются;

По умолчанию частота выборки берется равным realtime. При такой настройке кол-во хранимых данных может достигать колоссальных размеров, поэтому вводится поле max\_samples, которое ограничивает кол-во записей, хранимых в бд на сервере автоматизации. При realtime значение поля storage\_period игнорируется и данные являются плавающими.

Период сбора статистики (storage\_period) принимает следующие значения:

* day-статистика собирается за день;
* week-статистика собирается за неделю;
* month-статистика собирается за месяц;

Период сбора статистики по умолчанию принимает значение week. Это значит, что статистические данные разбиваются по неделям.

**3.1.14** Модель device\_settings

Модель содержит информацию о настройках статистики на устройстве.

Модель имеет следующие данные:

* device\_id-идентификатор устройства в системе;
* statistics\_collection- флаг, определяющий, сохранять ли статистику на сервере автоматизации;
* start\_time- начало последнего периода сбора статистики;
* created\_at- время создания модели;
* updated\_at- время последнего изменения модели;

По умолчанию сбор статистики для устройства выключен, при этом поле statistics\_collection принимает значениение false.

Данные настройки статистики предоставляются для каждого устройства в системе.

## **3.2** Состояния

Для роутинга используется angular-ui-router. Этот модуль использует понятие «состояние» для предоставления роутинга. Состояние – angular-сервис, предоставляющий текущий url страницы, параметры url и т.д. Опишем состояния приложения.

**3.2.1** Состояние Application

Application – это абстрактное состояние (не имеет отображения, имеет только контроллер). Application – это родительское состояние для всего приложения. Переход в состояние Application происходит только один раз при загрузке приложения в браузере. Оно служит для получения данных с backend-а и предоставления этих данных дочерним состояниям. Application ожидает следующие данные:

1. Список логических областей(areas). Данный список приходит с контроллера backend-а areas\_controller.
2. Список устройств (devices). Устройства агрегируют принадлежащие им элементы управления уже на стороне backend-а для упрощения управления устройствами. Данный список приходит с контроллера backend-а devices\_controller.
3. Список событий (notifications). Контроллер backend-а notifications\_controller.
4. Данные о сервере автоматизации (automationServer). Эти данные агригируют информацию модели geolocation для уменьшения количеств обращений к серверу. Контроллер backend-а automation\_server\_controller.

Application состояние объединяет контроллер mainCtrl и отображение layout.html, которые являются родительскими для всех остальных состояний (всего приложения). layout.html содержит директиву notificationDrawer, которая служит для отображения и управления сообщениями приложения.

Контроллер mainCtrl использует следующие зависимости:

* $scope-объект модели, связывает представление и контроллер;
* $window-angular-сервис, предоставляющий доступ к глобальному объекту window браузера;
* Auth- angular-сервис авторизации для Device;
* Area-список сообщений;
* autoServSock- сервис, использующий Websocket для связи с сервером автоматизации;
* deviceHelper-сервис, содержащий методы для работы с объектом устройства;
* Script-список сценариев;
* scriptHelper-сервис, содержащий методы для работы с моделью сценария;

Контроллер mainCtrl содержит следующие поля и методы:

* areas- список логических устройств;
* notifs-список сообщений;
* scripts-список сценариев;
* areNewNotifs()-метод, определяющий, есть ли новые сообщения;
* logout()-метод для выполнения выхода пользователя;

**3.2.2** Dashboard

Dashboard – это состояние, отвечающее за отображение карточек статистики по серверу автоматизации, статистики для устройств, статистики для сценариев и погоды. Карточки расположены в три колонки на большом экране (desktop, 1200px и выше), в две колонки для средних устройств (992px и выше) и в один столбик для малых устройств (768px и выше). Каждая карточка имеет собственную иконку, характеризующую ее назначение. Dashboard имеет контроллер dashboardCtrl. Контроллер получает данные от родительского состояния Application. Данные для карточки сервера автоматизации, данные для карточки статистики устройств, данные для карточки статистики скриптов. Это соответственно список устройств areas, список скриптов scripts и объект, хранящий информацию о сервере автоматизации automationServer.

Контроллер dashboardCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* automationServer-сервис, предоставляющий объект сервера автоматизации с агрегированным объектом geolocation;
* autoServSock- сервис, использующий Websocket для связи с сервером автоматизации;
* areas –список всех доступных пользователю логических областей;
* scripts - список всех доступных пользователю сценариев;

**3.2.3** Scripts

Scripts – состояние, отвечающее за отображение и взаимодействие со сценариями. Контроллер scriptsCtrl является дочерним состоянием для application. Он получает от Application-состояния список сценариев scripts и отображает его в виде списка. Контроллер также ответственен за создание визарда для создания нового скрипта. Код сценария храниться на сервере web-приложения и при запуске передается на сервер автоматизации на выполнение. В данной версии сервиса логическая группировка сценариев не предусмотрена.

Контроллер dashboardCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* Script – сервис, который предоставляет доступ к модели сценария;
* ModalService – сервис, предоставляющий возможность отображать модальное окно;
* notifier – сервис, предоставляющий методы для создания сообщений;
* Area- сервис, который предоставляет доступ к модели логической области;

Контроллер dashboardCtrl содержит следующие поля и методы:

* scripts – список сценариев;
* ScriptTitle() – заглавие для списка сценарриев с учетом их количества;
* addNewScript() – метод по добавлению пового сценария в список сценариев scripts;
* destroy(item) – метод для удаления сценария из списка сценариев scripts;
* responsive – настройки для элемента angular-slick-carousel;

Настройка для элемента angular-slick-carousel принимает вид:

[

{

breakpoint: 768,

settings: {

arrows: false,

centerMode: true,

centerPadding: '40px',

slidesToShow: 3

}

},

{

breakpoint: 480,

settings: {

arrows: false,

centerMode: true,

centerPadding: '40px',

slidesToShow: 1

}

}

]

**3.2.4** Areas

Areas – это состояние, отвечающее за отображение и взаимодействие с логическими областями. Area изображены в виде карточек-контейнеров для устройств. При первой регистрации пользователь имеет логическую область по умолчанию с именем Area 0. Все новые устройства попадают именно в логическую область по умолчанию. Устройства накже отображаются в этом состоянии. Они отображаются элементами списка, вложенного в логическую область. Контроллер areasCtrl получает список логических областей от Application-состояния. AreasCtrl также ответственен за создание областей.

Контроллер areasCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* Area - сервис, который предоставляет доступ к модели логической области;
* ModalService – сервис, предоставляющий возможность отображать модальное окно;
* notifier- сервис, предоставляющий методы для создания сообщений;

Контроллер areasCtrl содержит следующие поля и методы:

* AreaTitle()-заглавие для списка логических устройств с учетом их количества;
* addNewArea()- метод для создания модели новой логической области;
* removeAreaOnly()- метод для удаления только логической области, устройства, принадлежащие удаляемой области переносятся в область по умолчанию.

**3.2.5** automation\_server\_location

Данное состояние представляет страницу с Yandex Map. На карте обозначено местоположение сервера автоматизации. Данные о местоположении берутся из сервиса automationServer, который агрегирует все данные, касающиеся сервера автоматизации. Местоположение сервера берется из объекта geolocation. Широта – поле lat. Долгота – поле lon. Переход в данное состояние осуществляется при нажатии на карточку статистики для сервера автоматизации.

Контроллер состояния locationCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* automationServer- сервис, предоставляющий объект сервера автоматизации с агрегированным объектом geolocation;

Контроллер locationCtrl содержит следующие поля и методы:

* geolocation- данные для выставления метки на карте, показывающей расположение сервера автоматизации;
* serverCoords- массив, содержащий широту и долготу сервера;
* map- объект, агрегирующий поля, необходимые для директивы отображения yandex-карты yandex-map;

Объект map имеет следующие поля:

* center- координаты точки на карте,
* coordinates- координаты метки на карте, отображающей местонахождение сервера автоматизации;
* balloonContentHeader-заголовок подсказки, отображаемой при наведении на метку сервера;
* balloonContentBody- текст, отображаемый при нажатии на метку на карте, представляет собой значение долготы и широты;
* zoom- увеличения на карте, по умолчанию равно 10;

**3.2.6** devices\_statistics

Состояние, отображающее детальную статистику по устройствам, подключенным к системе. Устройства вне имеют конкретных типов, а характеризуются набором элементов управления. Для различного отображения устройств в UI служит поле label. По этому же полю происходит подсчет устройств каждого типа с выводом результата в данное состояние.

Контроллер состояния devStatCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* deviceInfo- сервис, предоставляющий информацию о разных типах устройств;
* Area- сервис, который предоставляет доступ к модели логической области;

DeviceInfo сервис не используется для определения логики для разных типов устройств, но только для их уникального отображения. DeviceInfo, например, хранит поле icon для устройств, если в поле label, тип устройства неизвестный, то берутся атрибуты для неизвестного устройства из сервиса, а icon принимает вид восклицательного знака.

Контроллер devStatCtrl содержит следующие поля и методы:

* areas- список всех доступных пользователю логических областей, в котором область агрегирует список принадлежащих ей устройств;
* devices- список всех устройств в системе из всех логических областей;
* labels- список уникальных типов устройств в системе;
* devNumberByLabel- объект, представляющий информацию в виде ключ-значение для статистики по устройствам;
* watchDevicesChanges()- метод, вызываемый в контролере, служит для слежением за изменениями кол-ва устройств в системе;
* devIcon()- метод для получения информации об устройстве по его полю label;

Метод watchDevicesChanges при каждом изменении количества устройств в системе обновляет статистику по типам устройств, находящимся в системе;

DevNumberByLabel может принимать значения из поля label. На текущий момент поле label, принимает следующие значение:

* lamp-лампочка;
* bulb-источник света, лампа;
* temperature\_sensor-датчик температуры;
* light\_sensor-датчик освещенности;
* door\_sensor-датчик открытия дверей;
* smoke\_detector-датчик задымленности;
* water\_leakage\_sensor-датчик протечки воды;
* gas\_sensor-датчик утечки газа;
* motion\_or\_sound\_sensor- совмещенные датчики движения и звука;
* unknown\_device-неизвестное устройство;

Устройство считается неизвестным, если его нету в списке devices сервиса deviceInfo.

## **3.3** Директивы

Директивы – это переиспользуемые элементы приложения. Директива имеет собственный контроллер, вид и получает параметры. Директивы помещаются в DOM в качестве пользовательских HTML-тегов и атрибутов. Директивами, например, бывают однородные элементы списка с одинаковыми параметрами или логически выделенная часть приложения, такая как Notification Drawer, что позволяет разделять логику приложения. Директивы способствуют модульному проектированию и переиспользование компонентов, а также улучшают читаемость кода и упрощают разработку, так как задача может быть разбита на части.

**3.3.1** Директива automationServerCard

Директива получает данные о сервере автоматизации и отображает их в виде карточки на dashboard-е. При нажатии на карточку происходит переход в состояние devices\_statistics.

Контроллер директивы automationServerCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* moment- сервис, предоставляющий доступ к библиотеке, работающей с датой и временем;
* $interval- angular-сервис, являющийся оберткой к js функции window.setInterval, позволяющей вызывать функции периодически;
* automationServer-сервис, предоставляющий объект сервера автоматизации с агрегированным объектом geolocation;
* autoServSock- сервис, использующий Websocket для связи с сервером автоматизации;

Контроллер automationServerCtrl содержит следующие поля и методы:

* country- страна, в которой находится сервер автоматизации;
* geoloc- объект модели geolocation для сервера автоматизации;
* regionName- название региона, в котором находится сервер автоматизации;
* timezone- часовой пояс сервера автоматизации;
* url- адрес сервера автоматизации;
* connection- состояние подключения сервера автоматизации;

Поле connection может принимать следующие значения:

* CONNECTING- попытка настроить websocket-соединение;
* OPEN- websocket-соединение готово, есть возможность получать и отправлять пакеты на сервер автоматизации;
* CLOSING- websocket-соединение в процессе закрытия;
* CLOSED- websocket-соединение закрыто, т.е нету возможности отправки и принятия сообщений с сервера автоматизации;
* RECONNECT\_ABORTED-неудачная попытка переподсоединитьcя, websocket-соединение закрыто;

Также на карточке отображается время в часовом поясе сервера автоматизации. Также контроллер директивы следит за изменениями состояния web-сокета при помощи метода $watch сервиса $scope. Web-сокет соединяет web-сервис и сервер автоматизации и может находится в разных состояниях. Контроллер директивы отображает изменения состояния на карточке с помощью поля connection и соответствующей ему иконки.

**3.3.2** Директива weatherCard

Директива служит для отображения карточки погоды на dashbpard-е.

Контроллер директивы использует сервис weatherMap для получения погоды с web-сервиса OpenWeatherMap.

Контроллер директивы weatherCardCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* weatherMap- сервис, предоставляющий доступ к данным о текущей погоде;

Контроллер automationServerCtrl содержит следующие поля и методы:

* humidity- влажность в процентах;
* pressure- давление в hPa;
* degrees- в градусах цельсия;
* description- словесное описание погоды;
* icon- png изображение, соответствующей текущей погоде;
* name- название местности, в которой показывается погода;
* weather- объект с данными о погоде;

**3.3.3** Директива scriptsStatCard

Директива служит для отображения статистики по подключенным устройствам к системе. Она также отображает общее кол-во сценариев, принадлежащих пользователю.

Контроллер директивы scriptsStatCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* scriptsStat- сервис, предоставляющий доступ к статистическим данным о текущем количестве подключённых устройств;
* scripts – список пользовательских сценариев;
* Script – сервис, который предоставляет доступ к модели сценария;

Контроллер scriptsStatCtrl содержит следующие поля и методы:

* starting– число сценариев, начинающих работу на сервере автоматизации;
* running– число сценариев, выполняющихся на сервере автоматизации;
* stopping– число останавливающихся сценариев на сервере автоматизации;
* stopped– число остановленных или незапущенных сценариев на сервере автоматизации;

**3.3.4** Директива devStatCard

Директива для отображения карточки статистики по устройствам, подключенным к системе. По нажатию на карточку выполняется переход в состояние devices\_statistics c более подробной статистикой.

Контроллер директивы deStatCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* deviceInfo- сервис, предоставляющий информацию о разных типах устройств;
* Area- список всех доступных пользователю логических областей, в котором область агрегирует список принадлежащих ей устройств;

Контроллер deStatCtrl содержит следующие поля и методы:

* areas- список всех доступных пользователю логических областей, в котором область агрегирует список принадлежащих ей устройств;
* devices- список всех устройств в системе из всех логических областей;
* labels- список уникальных типов устройств в системе;

**3.3.5** Директива notificationDrawer

Директива служит для отображения Notification Drawer – выподающего списка с сообщениями.

Директива отображает два списка сообщений. Вид директивы содержит две вложенные директивы notificationAccordion для отображения пользовательских сообщений и сообщений сервера автоматизации.

Контроллер директивы notifDrawerCtrl использует следующие зависимости:

* $scope- модель, связывает представление и контроллер;
* Notif - сервис, содержащий список сообщений;

Поля и методы контроллера notifDrawerCtrl:

* deleteAll()- метод для удаления всех сообщений, как пользовательских (возникших в web-приложении), так и системных;

**3.3.6** Директива notificationAccordion

Директива служит для отображения элемента accordion для списка сообщений.

Контроллер директивы производит фильтрацию сообщений по полю модели сообщения origin и отображает только те сообщения, источник происхождения которых задан для директивы, в приложении это:

* user;
* automation\_server;

Контроллер директивы notifAccordionCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* $filter- angular-сервис для фильтрации объекта по определенному свойству;
* Notif- сервис, содержащий список непрочитанных сообщений системы;

Контроллер notifAccordionCtrl содержит следующие поля и методы:

* deleteAll()- метод для удаления всех сообщений, как системных так и пользовательских;
* formatDeleteText()- метод для форматированного вывода текста для ссылки на удаление сообщений;

**3.3.6** Директива notificationItem

Директива служит для отображения и управления отдельным сообщением.

Контроллер директивы notificationCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* notif- модель отдельного сообщения;
* notifier- сервис, предоставляющий методы для создания сообщений;
* onDelete()- функция для удаления данного сообщения;

Директива отображает следующие атрибуты:

* createdAt c фильтром по дате - дата создания сообщения;
* createdAt по времени – время создания сообщения;
* details-описание события, для которого создано сообщение;
* category- тип сообщения, отображается разными иконками;

**3.3.7** Директива areaCard

Контроллер директивы предназначен для управления отдельной логической областью (area). Логическая область является контейнером для устройств. Контроллер имеет методы для изменения имени area и принимает от контроллера areasCtrl метод по удалению area. Эти действия доступны в kebab-меню в виде. Только вид логической области по умолчанию не содержит пункт меню по ее удалению.

Контроллер директивы areaCardCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* Area- сервис, который предоставляет доступ к модели логической области;
* area- конкретная логическая область, представляемая данной директивой;
* onEdit()- метод, вызываемый при изменении модели логической области;
* onDestroy()- метод, вызываемый при удалении данной логической области;

Контроллер areaCardCtrl содержит следующие поля и методы:

* areas- список всех доступных пользователю логических областей, в котором область агрегирует список принадлежащих ей устройств;
* updateName()- метод для изменения имени модели логического устройства;
* link-функция, выполняемая при компиляции директивы;

**3.3.7** Директива areaDeviceItem

Контроллер директивы отвечает за отображение и манипулирование устройством. Устройство отображается внутри родительского area. Методы контроллера позволяют перемещать устройство из одного area в другой, редактировать имя устройства, редактировать имена элементов управления (control) устройства. Вид устройства содержит следующий набор директив элементов управления устройства:

* dimmerControl, представляет элемент управления диммер;
* numValueControl, представляет отображение цифрового значения;
* switchStateControl, элемент мультипереключатель;
* symValueControl, отображает символьное значение;
* toggleControl, элемент переключатель;

Контроллер директивы areaDeviceCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* deviceInfo- сервис, предоставляющий информацию о разных типах устройств;
* ModalService-сервис, предоставляющий возможность отображать модальное окно;
* Area- сервис, который предоставляет доступ к модели логической области и ее методам;
* notifier- сервис, предоставляющий методы для создания сообщений;
* device - объект модели устройства, представляемый данной директивой;

Контроллер areaDeviceCtrl содержит следующие поля и методы:

* info – информация о устройстве, берется из сервиса deviceInfo;
* areas- список логических областей вместе с принадлежащими им устройствами;
* changeArea() –метод по изменению логической области для устройства;
* editModal() – метод для вызова модального окна для редактирования полей модели устройства;
* updateName() – метод для редактирования имени устрйства;
* show – флаг, определяющий, показывать ли элементы управления, принадлежащие данному устройству;
* toggleControlsAppearance() – метод, вызываемый для скрытия, показа элементов управления;
* updateControlName() – метод для редактирования имени элемента управления;

**3.3.8** Директива dimmerControl

Директива отвечает за манипулирование элементом управления вида диммер пользователем.

Контроллер директивы dimmerControlCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* deviceInfo- сервис, предоставляющий информацию о разных типах устройств;
* deviceHelper-сервис, содержащий методы для работы с объектом устройства;
* device - объект модели устройства, представляемый данной директивой;
* control - объект модели элемента управления, представляемый данной директивой;

Контроллер использует метод sendDeviceChange сервиса deviceHelper, при помощи которого сервер автоматизации оповещается о управлении пользователем диммером. Контроллер имеет соответствующую callback-функцию с именем onHandleUp, вызываемую при манипуляции диммером.

**3.3.9** Директива numValueControl

Директива служит для отображения числовым значением элемента управления.

Контроллер директивы numValueCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* deviceInfo- сервис, предоставляющий информацию о разных типах устройств;
* deviceHelper-сервис, содержащий методы для работы с объектом устройства;
* device - объект модели устройства, представляемый данной директивой;
* control - объект модели элемента управления, представляемый данной директивой;

Контроллер numValueControl содержит следующие поля и методы:

* info-информация для данного типа устройства;

**3.3.10** Директива switchStateControl

Директива служит для отображения текущего состояния сложного переключателя (имеющего более двух состояний) и манипуляции данным элементом управления.

Контроллер директивы switchStateControl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* deviceInfo- сервис, предоставляющий информацию о разных типах устройств;
* deviceHelper-сервис, содержащий методы для работы с объектом устройства;
* device - объект модели устройства, представляемый данной директивой;
* control - объект модели элемента управления, представляемый данной директивой;

Контроллер использует метод sendDeviceChange сервиса deviceHelper, при помощи которого сервер автоматизации оповещается о произошедшей манипуляции данным элементом управления. Контроллер имеет соответствующую callback-функцию с именем switch, вызываемую при манипуляции переключателем.

**3.3.11** Директива symValueControl

Директива служит для отображения текущего состояния символьного значения.

Контроллер директивы symValueCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* deviceInfo- сервис, предоставляющий информацию о разных типах устройств;
* deviceHelper-сервис, содержащий методы для работы с объектом устройства;
* device - объект модели устройства, представляемый данной директивой;
* control - объект модели элемента управления, представляемый данной директивой;

Контроллер symValueControl содержит следующие поля и методы:

* info-информация для данного типа устройства;

**3.3.11** Директива toggleControl

Директива служит для отображения текущего состояния простого переключателя (имеющего два состояния) и манипуляции данным элементом управления.

Контроллер директивы toggleCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* deviceInfo- сервис, предоставляющий информацию о разных типах устройств;
* deviceHelper-сервис, содержащий методы для работы с объектом устройства;
* device - объект модели устройства, представляемый данной директивой;
* control - объект модели элемента управления, представляемый данной директивой;

Контроллер использует метод sendDeviceChange сервиса deviceHelper, при помощи которого сервер автоматизации оповещается о произошедшей манипуляции данным элементом управления. Контроллер имеет соответствующую callback-функцию с именем onChange, вызываемую при манипуляции переключателем.

**3.3.12** Директива btnGroup

Директива используется для отображения элемента группы кнопок. Группа кнопок представляет выбор из предоставленных возможных значений. При нажатии на определенную кнопку происходит запоминание выбора в переменную selectedIndex.

Контроллер директивы btnGroupCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* list- список возможных значений на выбор;
* model-значение выбранного элемента;
* selectedIndex- индекс выбранного элемента;
* disapear- флаг, определяющий, исчезает ли группа кнопок после произведенного выбора;
* onClickList- список функций, определенная функция вызывается при нажатии на кнопку с ее индексом в списке;

Контроллер btnGroupCtrl содержит следующие поля и методы:

* show- флаг, отвечающий за отображение директивы;
* selectedIndex- индекс выбранной опции;
* select- метод, вызываемый при нажатии выбираемой опции;

**3.3.13** Директива controlInput

Директива используется при вводе значения для элемента управления.

Контроллер директивы controlInputCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* type – тип элемента управления, директива позволяет вводить значения для всех типов элементов управления;
* model – модель, сохраняющая вводимые данные;

Контроллер controlInputCtrl содержит следующие поля и методы:

* data - модель, сохраняющая вводимые данные;
* watch() – метод, следящий за изменениями модели;

**3.3.14** Директива controlSelect

Директива используется для выбора элемента управления. Выбор происходит в следующей последовательности:

* выбирается логическая область
* выбирается устройство
* выбирается элемент управления

Контроллер директивы controlSelectCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* areas- список логических областей вместе с принадлежащими им устройствами;
* device - объект модели устройства, представляемый данной директивой;
* control - объект модели элемента управления, представляемый данной директивой;

Контроллер controlSelectCtrl содержит следующие поля и методы:

* watchAreaChanges() – метод, вызываемый при выборе логической области;
* watchDeviceChanges() – метод, вызываемый при выборе устройства;

**3.3.15** Директива scriptListItem

Директива используется для отображения сценария как элемента списка сценариев в состоянии scripts. Директива отображает всю необходимую информацию о сценарии и предоставляет методы для его управления.

Контроллер директивы scriptListItemCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* scriptHelper-сервис, содержащий методы для работы с моделью сценария;
* ModalService – сервис, предоставляющий возможность отображать модальное окно;
* script –сценарий, отображаемый данной директивой;
* destroy – делегируемый метод для удаления сценария;
* Контроллер scriptListItemCtrl содержит следующие поля и методы:
* updateName()- метод для обновления имени сценария;
* updateDescription()- метод для обновления описания сценария;
* shedule()- метод для планирования сценария, его выполнение происходит на сервере автоматизации через определенный промежуток времени, указанный в поле модели сценария start\_time;
* run()- метод для запуска сценария, выполнение сценария происходит на сервере автоматизации;
* stop()- остановка запущенного сценария;

**3.3.16** Директива scriptModal

Директива используется для отображения wizard-а для создания пользовательских сценариев. Сценарии для выполнения передаются серверу автоматизации.

Контроллер директивы scriptModalCtrl использует следующие зависимости:

* $scope- объект модели, связывает представление и контроллер;
* Script – сервис, который предоставляет доступ к модели сценария;
* areas- список логических областей вместе с принадлежащими им устройствами;
* close- список логических областей вместе с принадлежащими им устройствами;
* WizardHandler- angular-сервис для управления модальным окном;
* scriptHelper-сервис, содержащий методы для работы с моделью сценария;
* controlInfo- сервис, предоставляющий информацию по всем типам элементов управления;

Контроллер scriptModalCtrl содержит следующие поля и методы:

* wizard()- метод для доступа к объекту wizard сервиса WizardHandler.
* script- модель создаваемого сценария;
* wlist-список с сохраненными шагами wizard-а;
* finish()- метод, вызываемый для инициализации модели скрипта из сохраненных шагов wizard-а;
* stepBack()- метод, для возвращения wizard-а на шаг назад;
* isCurrentStep()- метод,определяющий индекс текущего шага wizard-а;
* defaultsForControl()- метод, возвращающий значения по умолчанию для элемента управления;

Wlist содержит объекты-шаги сценария. Элемент списка wlist не имеет обязательных полей. Если элемент имеет метод template, то данные из этого элемента списка компилируются в участок исходного кода на DSL-языке для выполнения сценариев.

## **3.4** Сервисы

Сервисы – это модули, выполняющие некоторую задачу. Разделяемые данные и алгоритмы программы рекомендуется выносить в сервисы, так как контроллеры могут использовать сервисы через зависимости. Сервисы способствуют повторному использованию кода.

**3.4.1** Сервис deviceHelper

Сервис служит для хранения методов работы с websocket-ом и устройством (device).

Сервис содержит следующие поля и методы:

* buildDeviceFromPack()- метод для создания модели устройства с json-пакета pack, пришедшего с websocket-а.
* createDevice()- метод для создания устройства и его сохранения в базе данных;
* convertStateFromStr()- метод для перевода значений элементов управления из строкового представления при необходимости;
* checkStateUpdate()- метод для валидации обновлений значений элементов управления;
* checkStateUpdate()- метод для валидации обновлений значений элементов управления;
* setControlsValues()- метод для установки значений элементов управления с пакета данных, пришедшего с websocket-а.
* findDeviceById()- метод для нахождения устройства по полю id в списке устройств devices.
* applyChanges()- метод для обновления значений элементов управления, данные приходят с websoket-а, подсоединенного к серверу автоматизации;
* buildChangePackFromDevice()- метод для создания пакета с обновлениями элементов управления, обновления создаются пользователем через web-сервис;
* sendDeviceChange()- метод для отправки изменений значений элементов управления, произведенных пользователем через UI;

**3.4.2** Сервис scriptHelper

Сервис служит для хранения методов по работе сценариев с websocket-ом.

Сервис содержит следующие поля и методы:

* buildStartPack()- метод для создания пакета запуска сценария;
* sendStart()- метод для отправки сценария через websocket;
* sendStop ()- метод для создания пакета с командой остановки сценария;
* buildStopPack()- метод создания пакета остановки сценария;
* applyChanges()- метод по обновлению данных о скрипте, обновления приходят с сервера автоматизации из websocket-а;

Обновления содержат текущий статус сценария (поле status), состояния могут быть следующие:

* starting– статус сценария, при котором он начинает инициализацию на сервере автоматизации;
* running– стату сценария, при котором он выполняется на сервере автоматизации;
* stopping– статус сценария, при котором он заканчивает работу на сервере автоматизации;
* stopped– статус сценария, при котором он не запущен на сервере автоматизации;

**3.4.3** Сервис Notif

Сервис служит для работы с моделью notification.

Сервис содержит следующие поля и методы:

* notifs – список сообщений;
* success()- метод по созданию сообщения с типом success;
* error()-метод по созданию сообщения с типом error;
* info()-метод по созданию сообщения с типом info;
* areRead()- метод, позволяющий определить, прочитаны ли все соовщения;
* filterByOrigin()- метод для фильтрации сообщений по полю модели origin;
* markReadByGroup()- метод для пометки сообщений как прочитанных в определенной группе;

**3.4.4** Сервис controlsInfo

Сервис служит для предоставления методов и информации о элементах управления.

Сервис содержит следующие поля и методы:

* validDimmerState()- метод определяет, валидны ли данные для элемента управления dimmer;
* validToggleState()-метод определяет, валидны ли данные для элемента управления простой переключатель (toggle);
* validSymValueState()-метод определяет, валидны ли данные для символьного значения;
* validNumValueState()-метод определяет, валидны ли данные для числового значения;
* validSwitchState()-метод определяет, валидны ли данные для элемента сложного переключателя;
* convertToggleStateFromStr()- метод для перевода значения из строки для элемента управления простой переключатель (toggle);

**3.4.4** Сервис scriptsStat

Сервис служит для хранения и оправления статистики по сценариям.

Сервис содержит следующие поля и методы:

* running– количество запущенных сценариев;
* stopped– количество остановленных сценариев;
* calc()– метод для подсчета кол-ва запущенных и остановленных сценариев;
* update()– метод, вызываемый для обновления статистики по сценариям;

**3.4.4** Сервис weatherMap

Сервис служит для получения данных с web-сервиса погоды OpenWeatherMap.

Сервис содержит следующие поля и методы:

* cityWeather()–метод для получения текущей погоды по названию города;
* convertToCelsius()– перевод данных по температуре в градусы Цельсия;

Данные с сервиса погоды приходят в следующем виде:

# РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## **4.1** Подключение устройства

Подключение устройства происходит автоматически (механизм PnP). Далее приведен алгоритм и детали реализации данного механизма:

Пакет с данными о новом устройстве приходит на websocket automationSocket. Формат пакета «dev\_hello»:

{

type:’dev\_hello’,

  dev\_id: ‘erg545gsg4334g’,

  label: ‘lamp’,

  controls: [

   {

      name: ‘Light Switch’,

    ctrl\_id: ‘67’,

      type: {

name: ‘switch\_state’,

optional: {

names: [ ‘off’, ‘onn’, ‘zoff’]

}

}

    },

   {

      name: ‘Bulb 1’,

    ctrl\_id: ‘67’,

      type: { ‘name’: ‘toggle’,

‘optional’: {}

}

    }

],

  time\_stamp: '01.01.1970'

}

В пакет входят следующие поля:

* type–тип пакета;
* dev\_id– идентификатор устройства;
* label– метка устройства (тип);
* controls – список элементов управления вместе с их атрибутами;
* name – название элемента управления;
* ctrl\_id –идентификатор элемента управления;
* type – подобъект в элементе управления, описывающий параметры типа;
* type.name – название типа;
* optional – опции, доступные конкретному типу;
* optional.names – список для выбора в элементе управления «сложный переключатель»;
* time\_stamp – время подключения нового устройства к серверу автоматизации;

Обработка пакетов начинается с их разделения по типам пакетов (поле type). Если поле type принимает значение dev\_hello, то пакет распознается как пакет, оповещающий о подключении нового устройства к системе. В коде распознавание пакета выглядит так:

autoServSock.onMessage(function(message){

var pack = JSON.parse(message.data);

switch(pack.type){

case 'dev\_hello':

deviceHelper.createDevice(pack).then(function (response) {

notifier.info(

{title: 'New device',

subject: response.name,

origin: 'automation\_server'

});

}).catch(function (response) {

notifier.error(

{ title: 'New device',

subject: pack.dev\_id,

errors: response.errors,

origin: 'automation\_server'

});

});

break;

}

После того, как пакет был распознан как оповещающий о приходе нового устройства, то вызывается метод createDevice сервиса deviceHelper, в котором производится создание модели устройства из данных пакета и инициализация показаний элементов управления устройства первыми значениями:

helper.createDevice = function(hello\_pack) {

var device = helper.buildDeviceFromPack(hello\_pack);

return new Device(device).create().then(function (response) {

var defaultArea = Area.default();

defaultArea.devices.unshift(response);

helper.setControlsValues(response,

hello\_pack.changes\_packet);

return response;

});

};

Метод buildDeviceFromPack() создает модель из данных пакета. Метод create() сервиса Device сохраняет устройство в базе данных и возвращает promise-объект. Promise-объект содержит метод then(), вызываемый при удачном сохранении модели в бд. В методе then() выполняется начальная инициализация значений элементов управления в вызове метода setControlsValues и устройство добавляется в логическую область по умолчанию Default Area defaultArea.devices.unshift(response). Метод для построения модели устройсва buildDeviceFromPack из пакета имеет вид:

helper.buildDeviceFromPack = function(pack) {

var device = \_.cloneDeep(pack);

// give a default unique device name device.name = \_.lowerCase(device.label) + " " +device.dev\_id;

delete device['type'];

delete device['changes\_packet'];

delete device['time\_stamp'];

return device;

};

В этом методе формируются поля модели устройства. Имя устройства по умолчанию формируется из идентификатора устройства и поля label. Большинство полей пакета просто копируется в модель устройства кроме следующих полей:

* type–тип пакета;
* changes\_packet–подобъект пакета, хранящий инициализационные данные для элементов управления устройства;
* time\_stamp–время подключения устройства к серверу автоматизации;

## **4.2** Управление элементом устройства

Управление элементом устройства происходит в контроллерах следующих директив:

* dimmerControl, представляет элемент управления диммер;
* numValueControl, представляет отображение цифрового значения;
* switchStateControl, элемент сложный переключатель;
* symValueControl, отображает символьное значение;
* toggleControl, элемент простой переключатель переключатель;

Рассмотрим алгоритм управления элементом переключателя toggle. При изменении позиции переключателя в UI вызывается функция noChange() сервиса deviceHelper в контроллере директивы.

$scope.onChange = function () {

deviceHelper.sendDeviceChange($scope.control,

$scope.device);

};

Метод sendDeviceChange() берет измененное значение элемента управления, управляемого через данную директиву, формирует пакет данных и отправляет на сервер автоматизации.

helper.sendDeviceChange = function (control, device) {

automationSocket.then(function (sock) {

var changes\_pack =

helper.buildChangePackFromDevice(control, device);

console.log(changes\_pack);

sock.send(changes\_pack);

});

};

Метод buildChangePackFromDevice выполняет формирование пакета с обновлениями состояния элемента управления. Пакет обновления выглядит так:

{

type: "dev\_changes",

dev\_id: "Unique\_str",

controls: [

{

ctrl\_id: "Unique\_Str",

state: "on"

},

{

ctrl\_id: "Unique\_Str",

state: "7"

}

],

time\_stamp: "01/01/1970"

}

Поля пакета изменения состояния:

* type–тип пакета;
* controls– список данных для элементов управления;
* dev\_id– идентификатор устройства;
* ctrl\_id– идентификатор элемента управления;
* state– текущее состояние элемента управления;

Пакет с такой структурой отправляется на сервер автоматизации.

## **4.3** Запуск сценария

Запуск сценария производится из контроллера директивы scriptListItem при нажатии кнопки start в UI. Метод контроллера, запускаемый при нажатии кнопки использует метод sendStart сервиса scriptHelper:

$scope.run = function () {

scriptHelper.sendStart($scope.script);

};

Метод sendStart формирует пакет запуска сценария, в который помещает код сценария на специально разработанном DSL-языке:

h.sendStart = function (script) {

automationSocket.then(function (sock) {

var scriptPack = h.buildStartPack(script);

console.log(scriptPack);

sock.send(scriptPack);

}).catch(function () {

});

};

Пакет формируется функцией buildStartPack, принадлежащей сервису scriptHelper. Функция выглядит следующим образом:

h.buildStartPack = function(script) {

var pack = {

type: "user\_script",

time\_stamp: moment().format()

};

\_.assign(pack, script);

return pack;

};

Пакет, формируемый функцией выглядит следующим образом:

{

type: "user\_script",

id: “34”

name: "Unique\_str",

code: "some\_text",

  start\_time: "01/01/1970",

time\_stamp: "01/01/1970"

}

Данный пакет отправляется на сервер автоматизации на выполнение. Он содержит следующие поля:

* type–тип пакета;
* controls– список данных для элементов управления;
* dev\_id– идентификатор устройства;
* ctrl\_id– идентификатор элемента управления;
* state– текущее состояние элемента управления;

## Остановка сценария

Остановка сценария производится из контроллера директивы scriptListItem при нажатии кнопки stop в UI. Метод контроллера, запускаемый при нажатии кнопки использует метод sendStop сервиса scriptHelper:

$scope.stop = function () {

scriptHelper.sendStop($scope.script);

};

Метод sendStop формирует пакет остановки сценария, в который помещает идентификатор сценария:

h.sendStart = function (script) {

automationSocket.then(function (sock) {

var scriptPack = h.buildStartPack(script);

console.log(scriptPack);

sock.send(scriptPack);

}).catch(function () {

});

};

Пакет формируется функцией buildStopPack, принадлежащей сервису scriptHelper. Функция выглядит следующим образом:

h.buildStopPack = function (script) {

var pack = {

type: "script\_changes",

name: script.name,

status: "stopped",

time\_stamp: moment().format()

};

return pack;

};

Пакет, формируемый функцией выглядит следующим образом:

{

type: "script\_changes",

name: "script\_name",

status: "stopped",

time\_stamp: "01/01/1970"

}

Данный пакет отправляется на сервер автоматизации на выполнение. Он содержит следующие поля:

* type–тип пакета;
* name – названия сценария;
* status – статус сценария;
* time\_stamp – метка времени формирования пакета;

Полю status сформированного пакета присваивается значение stopped.

## **4.5** Создание сценария через wizard

При нажатии на кнопку New Script в UI пользователь переходит в модальное окно с wizard-ом. Шаг wizard-а представляется объектом, который хранит все необходимые опции для перехода на следующий шаг. Опции нулевого шага выглядят следующим образом:

{

template: function () {

return "if { ";

}

}

Метод template возвращает некоторую часть кода сценария, при завершении прохождения wizard-а выполняется проход по всем шагам wizard-а с последующим соединением частей кода в сценарий.

Первый шаг сценария имеет следующие опции:

{

inputTypesList: ['Time moment', 'Device control value',

'Weather'],

selectedIndex: null

}

Первый шаг wizard-а – это выбор типа условия, при котором происходит определенное событие. InputTypesList – список опций для выбора пользователю. Виды опций:

* Time moment– условием сценария является некоторый момент во времени;
* Device control value – условием сценария является определенное условие, налагаемое на значение элемента управления;
* Weather – условием сценария является показания погоды;

На втором шаге wizard-а пользователю показываются только опции того условия, которое он выбрал на первом шаге wizard-а.

Если пользователь выбрал условием Time moment, то опции второго шага следующие:

{

time: null,

template: function(){

return "(time={{{time}}}";

}

}

Time– модель для объекта, представляющего время, выбранное пользователем.

Если пользователь выбрал условием Device control value, то опции второго шага следующие:

{

area: ctrlDefaults.area,

device: ctrlDefaults.device,

control: ctrlDefaults.control,

template: function(){

return "( {{device.dev\_id}}#{{control.ctrl\_id}}";

}

}

Объект ctrlDefaults содержит значения по умолчанию для логической области, для устройства и для элемента управления.

Если пользователь выбрал условием Weather, то опции второго шага следующие:

{

enableTemp: true,

temperature: null,

enableHumid: true,

humidity: null,

enablePressure: true,

pressure: null,

template: function(){

var templ = "";

if(this.enableTemp){

templ +=

"(weather.temperature={{temperature}}";

}

if(this.enableHumid){

if(!\_.isEmpty(templ)){

templ += ") and ";

}

templ += "(weather.humidity={{humidity}}";

}

if(this.enablePressure){

if(!\_.isEmpty(templ)){

templ += ") and ";

}

templ += "(weather.pressure={{pressure}}";

}

return templ;

}

}

Объект опций содержит следующие поля:

* enableTemp –флаг, отвечающий за чекбокс, при установлении которого температура добавляется в условие сценария;
* temperature –модель для ввода условия для температуры;
* enableHumid –флаг, отвечающий за чекбокс, при установлении которого влажность добавляется в условие сценария;
* humidity –модель для ввода условия для влажности;
* enablePressure –флаг, отвечающий за чекбокс, при установлении которого давление добавляется в условие сценария;
* pressure–модель для ввода условия для влажности;

Третий шаг сценария – это шаг после введения условия времени, пропускаем шаги, связанные с введением влажности и данных для элементов управления и переходим на восьмой шаг. Восьмой шаг – это шаг, на котором происходит выбор между повторным добавлением еще одного условия или переходом к действиям, выполняемым при соблюдении условия, созданного на первых шагах.

$scope.step3 = function () {

$scope.step8();

$scope.wizard().goTo(7);

};

Четвертый шаг сценария – это шаг после введения условия с элементом управления. Данный шаг как и третий выполняет переход на восьмой шаг:

$scope.step4 = function () {

$scope.step8();

$scope.wizard().goTo(7);

};

Пятый шаг сценария – это шаг введения оператора сравнения для условия с элементом управления:

{

compList: controlsInfo[ctrlType].comp,

comp: null,

template: function() { return this.comp }

}

Объект опций содержит следующие поля:

* compList– список всевозможных сравнений, совместимых с данным типом элемента управления.
* comp– модель для оператора сравнения.

Шаг шестой – шаг выбора источника ввода значения, значения для сравнения могут быть взяты:

* value from input– значение введенное пользователем.
* value from device control– значение взятое с другого элемента управления.

{

inputTypesList: ['value from input', 'value from device

control'],

selectedIndex: null

}

На конечном шаге wizard-а происходит компиляция кода сценария:

$scope.script.code = \_.reduce($scope.wlist, function

(result, elem) {

if(\_.has(elem, 'template')&&\_.isFunction(elem.template)){

result += Mustache.render(elem.template(), elem);

}

return result;

}, "");

Контроллер модального окна возвращает сформированную модель сценария;

# **5** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

Фреймворк Ruby on Rails имеет стандартный DSL-язык для написания тестов. В данном проекте был использован сторонний пакет rspec. Rspec предоставляет лучшую читаемость тестов и дополнительные возможности. Rspec используется для модульного и интеграционного тестирования. Также для тестирования моделей используется пакет Shoulda Matchers.

Rspec – это behavior-driven development (BDD) фреймворк. Он очень широко распространен в среде Rails-разработчиков. Основная концепция BDD основывается на test-driven development(TDD), в котором сперва пишутся тесты, а разработка основывается на написании достаточного количества кода для прохождения этих тестов. Далее проводится рефакторинг. RSpec содержит собственный фреймфорк mock-ов, которая полностью интегрирована в RSpec и основана на JMock. Простота синтаксиса RSpec делает его одним из популярнейших средств тестирования для Ruby приложений. RSpec может быть использована после установки rspec гема (gem).

RSpec содержит 3 других гема:

* rspec-core;
* rspec-expectation;
* rspec-mock;

Shoulda Matchers совместимые с Rspec тесты для Rails, охватывающие общую функциональность. Тесты в большинстве своем однострочные, что увеличивает читаемость теста.

## **5.1** Применение Rspec

Как упоминалось выше, RSpec предоставляет DSL-язык для описания поведения объектов. Зарезервированные слова, используемые в RSpec похожи таковым в других TDD фреймфорках. Например, если зарезервированные слова используются в Test::Unit, то считается, что они могут быть переведены на RSpec. Синтаксис RSpec легко читаем и описывает поведение кода, тем самым предоставляет свободу программирования. Каждый фреймворк для тестирования работает по следующей схеме- дает некоторый контекст, когда возникает событие, проверяются ожидания. Методы, наподобие describe(), context() и it() формируют основу тестового кода.

**5.1.1** Метод describe

Метод describe() используется для описания класса, метода или группы. Он является внешним блоком, который содержит тестовый код и отображает характеристики кода, заключенного в нем. Этот метод принимает любое количество параметров и необязательных блоков. Как бы то ни было, обычно один или два параметра используются для описания поведения группы. Первый параметр является ссылкой на класс или модуль, в то время как второй аргумент является опциональным и его тип данных – String. Группы могут быть также вложенными.

**5.1.2** Блок context

Блок context() используется для описания контекста в котором используется класс или метод, упомянутый в блоке describe. Он может считаться псевдонимом слову describe в сценарии, и они оба могут быть использованы попеременно. В основном, describe() используется для определения сущностей, а context()- для контекстов. Это помогает получать различные результаты в различных сценариях.

Использование describe и context выглядит следующим образом:

describe User do

context "has no account balance" do

....

end

end

**5.1.3** Метод it

Метод it()-это метод, который описывает спецификации теста в контексте. Блок it() принимает строку как аргумент и строка после слова it считается функцией, которую блок должен выполнить или другими словами – это test case.

Пример применение метода it.

describe User do

context "has no account balance" do

it "is not allowed to sanction a housing loan" do

puts "The loan cannot be sanctioned due to no

balance in the account."

end

end

end

**5.1.4** Ожидания RSpec.

Ожидание в Rspec – это утверждение, выражающее, что тестовое выражение считается правдивым. RSpec использует простой фреймворк и ключевые слова, такие как should() и should\_not() для выражения ожидания. Он поддерживает сопоставления, то есть объекты, которые пытаются соответствовать ожидаемому результату, как для обычных операций, так и для необычных выражений.

## **5.1** Тестирование приложения

Тесты приложения имеют структуру, отображенную на рисунке 5.1:

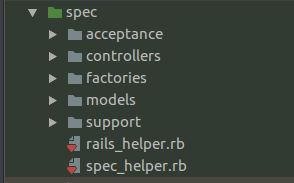


Рисунок 5.1 – структура тестов RSpec

Acceptance-тесты (высокоуровневые):

* end-to-end или full stack тестирование (интеграционное);
* black-box тестирование;
* эмулируют взаимодействие пользователя с приложением;
* пишуться с точки зрения пользователя;
* медленные (секунды, минуты);

В отличие от unit-тестов, acceptance-тесты покрывают весь стек, уровни и слои предложения. Тестирование разных слоев происходит одновременно и интегрированным образом. Rails приложение построено на основе паттерна MVC. Acceptance-тесты тестируют модели, контроллеры и виды в связке. Acceptance-тестирование также является black-box тестированием, т.е. тестированием по принципу черного ящика. Не зная, как работает система, подаются на ее вход некие данные, система реагирует на эти данные определенным образом. Acceptance-тесты пишутся с точки зрения пользователей и при этом происходит эмуляция взаимодействия пользователя с приложением.

Выполнение acceptance тестов похожа на тестирование пользователем. При этом запускается эмуляция браузера в памяти и тесты ведут себя как пользователь:

* нажимаются определенные кнопки;
* выполняются переходы по ссылкам;
* заполняются поля форм;

При этом приложение работает точно так же, как и в браузере у пользователя.

Unit-тесты (низкоуровневые):

* тестирую лишь малую часть приложения (метод, класс, модуль, блок);
* изолированные, применяются заглушки для модулей, с которыми взаимодействует тестируемый код;
* пишутся с точки зрения разработчика и системы;
* быстрые (миллисекунды);

Unit-тесты – это тесты моделей, контроллеров и других классов. Их отличие от acceptance-тестов в том, что они всегда тестируют лишь малую часть логики: метод, блок и т.д. Unit-тесты не тестируют связи между модулями и пишутся изолированно, поэтому с ними используются заглушки вместо нетестируемых других частей приложения. Unit-тесты пишутся с точки зрения разработчика и системы, т.е. в этих тестах уместно делать такие системные проверки, как изменение количества объектов моделей в базе данных.

Из двух уровней тестов acceptance и unit складывается полная схема тестирования по принципу BDD.

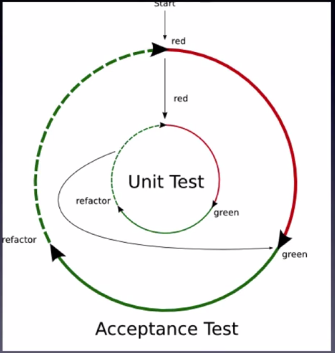


Рисунок 5.2 – схема BDD

Внешний уровень тестов – это accetance тесты, внутренний – unit. Разработка по схеме BDD идет следующим образом:

1. Все начинается с написание верхнеуровневого acceptance теста для реализации некоторого функционала.
2. Данный тест не проходит, так как функционала еще нету. На рисунке 5.2 это дуга, выделенная красным (от слова red до green).
3. Переход на более низкий уровень – уровень unit тестов.
4. Написание unit-тестов для моделей, контроллеров, видов, который сразу не проходит. На рисунке 5.2 это внутренняя красная дуга (от слова red до green).
5. Написание реализации функционала.
6. Запуск unit-тесты, тесты проходят.
7. Запуск acceptance-тестов, тесты проходят. Прохождение acceptance тестов является показателем того, завершилась ли история и работает ли система так, как ожидается. Если unit-тест не прошел, то происходит возвращение к шагу 4.
8. Возвращение к первому пункту, разработка ведется такими итерациями.

В acceptance-тестах тестируется:

* видимый пользователю функционал;
* различные сценарии использования;
* детальные требования;
* можно тестировать сразу несколько контроллеров;

Структура acceptance-теста:

* in order to <цель>;
* as a <роль пользователя>;
* i want <действие>;

Сценарии (тесты):

* то, что дано, данные доступные для теста;
* последовательность действий пользователя;
* проверка результата (ожидания);

Пример acceptance-теста для входа пользователя:

feature 'User sign in', %q{

In order to be able to manipulate devices

As an authenticated user

I want to be able to sign in

} do

given(:user) { create(:user) }

scenario 'Registered user try to sign in' do

sign\_in(user)

expect(page).to have\_content 'Signed in successfully.'

expect(current\_path).to eq root\_path

end

scenario 'Non-registered user try to sign in' do

visit new\_user\_session\_path

fill\_in 'Email', with: 'wrong@test.com'

fill\_in 'Password', with: '12345678'

click\_on 'Log in'

expect(page).to have\_content 'Invalid Email or password.'

expect(current\_path).to eq new\_user\_session\_path

end

end

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Руби С. Гибкая разработка веб-приложений в среде Rails. 4-е изд. / С. Руби, Д. Томас, Д. Хэнссон. – СПб.: Питер,2012. – 464 с.: ил.

[2] The Ruby On Rails Tutorial Learn web development with Rails [Электронный ресурс]. – Электронные данные. – Режим доступа: http://railstutorial.ru/.

[3] Ruby on Rails Guides [Электронный ресурс]. – Электронные данные. – Режим доступа: http://guides.rubyonrails.org.

[4] Coperland D. Powerful, Effective, and Efficient Full-Stack Web Development / D. Coperland. – Dallas, Texas, 2016.

[5] Coperland D. Powerful, Effective, and Efficient Full-Stack Web Development Second Edition / D. Coperland. – Raleigh, North Carolina, 2016.

[6] AngularJS API Reference [Электронный ресурс]. – Электронные данные. – Режим доступа: https://docs.angularjs.org/api.

[7] AngularJS with Ruby on Rails [Электронный ресурс]. – Режим доступа: http://angular-rails.com/.

Флэнаган Д. Язык программирования Ruby. / Д. Флэнаган,  
Ю. Мацумото. – СПб.: Питер, 2011. – 496с.

Thomas D. The Pragmatic Programmers’ Guide / D. Thomas. – Dallas, Texas, 2013.

Официальный веб-сайт Ruby [Электронный ресурс]. – Электронные данные. – Режим доступа: https://www.ruby-lang.org/.

Флэнаган Д. JavaScript. Подробное руководство. / Д. Флэнаган. – СПб: Символ-Плюс,2008. –992 с., ил.

Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. / Э. Гамма, Р. Хелм, Р. Джонсон – СПб: Питер, 2001. –368 с.: ил.

# ПРИЛОЖЕНИЕ А

*(рекомендуемое)*

Вводный плакат

# ПРИЛОЖЕНИЕ Б

(обязательное)

Схема структурная