

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных Технологий
Кафедра Информационных систем и технологий
Специальность 1-40 01 01 «Программное обеспечение информационных технологий»
Специализация 1-40 01 01 10 «Программное обеспечение информационных технологий
(программирование интернет-приложений)»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту на тему:

Телеграмм-бот помощник в сфере недвижимости

Выполнил студент Фомин Александр Фёдорович
(Ф.И.О.)

Руководитель проекта асс. Дубовик М.В.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Заведующий кафедрой к.т.н., доц. Смелов В.В.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Консультанты асс. Дубовик М.В.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Нормоконтролер асс. Дубовик М.В.
(учен. степень, звание, должность, подпись, Ф.И.О.)

Курсовой проект защищен с оценкой _____

Реферат

Пояснительная записка курсового проекта содержит 43 страниц пояснительной записки, 52 иллюстрации, 28 источников литературы, 8 приложения.

Ниже будут перечислены основные технологии, которые использованы в данном курсовом проекте.

AWS SERVICES, TELEGRAM API, ANGULAR API.

Целью курсового проекта является создание Телеграмм-бота и веб-приложения, которое предназначено для помощи поиска объявлений в сфере недвижимости.

В первой главе проводится постановка задачи, аналитический обзор прототипов по тематике курсового проекта, находится описание алгоритма решений.

Вторая глава посвящена процессу проектирования приложения, проектирование базы данных, проектирование структуры серверной части, проектирование структуры клиентской части, а также обзор диаграммы использования.

В третьей главе описывается процесс разработки проекта, реализации поставленных задач.

Четвёртая глава посвящена тестированию приложения.

В пятой главе описывается руководство программиста, благодаря которому пользователю будет легче понять интерфейс приложения и лучше в нём ориентироваться, а также развернуть приложение самостоятельно.

Содержание

Реферат.....	2
Введение	4
1. Постановка задачи	5
1.1 Обзор прототипов	5
1.1.1 Onliner.....	5
1.1.2 Аренда Квартир Бот.....	6
1.1.3 FlattyBy Бот.....	7
1.2 Алгоритмы решения	8
2. Проектирование	10
2.1 Проектирование серверной части	10
2.2 Проектирование базы данных	16
2.3 Проектирование клиентской части	18
3. Разработка.....	21
3.1 Настройка среды разработки	21
3.2 Деплой и управления инфраструктурой серверной части	23
3.3 Описание разработки бэка	26
3.4 Разработка Angular client-а и его настройка его CI/DI	32
4. Тестирование.....	35
5. Руководство программиста.....	42
Заключение	43
Список использованной литературы	44
Приложение А.....	46
Приложение Б.....	48
Приложение В	49
Приложение Г	54
Приложение Д.....	56
Приложение Е	58
Приложение Ж	62
Приложение З.....	63

Введение

В данном курсовом проекте разработаны телеграмм-бот «Taphut», телеграмм-бот помощник в сфере недвижимости» и веб-приложение «Taphut». Данные приложения представляют собой серверную часть, разработанную с помощью программной платформы Serverless на языке TypeScript, а также два клиентских приложения, где первый клиент разработан с использованием Telegram API[24], а второй с помощью frontend framework Angular[25] соответственно. Логически клиентские и серверная части представляют собой единое приложение.

Сфера по продаже и аренде недвижимости крайне быстро развивается, обретая всё большую популярность. Для поиска объявлений, вы можете воспользоваться газетами, обратиться к агентству недвижимости, частому агенту, либо же искать на бесконечном количестве платформ, предоставляющие возможность для размещения данного рода объявлений.

В агентствах недвижимости специалистов принято называть риелтор, агент, брокер, маклер, посредник. Это не слова синонимы, по виду деятельности они похожи, но у каждого есть своё индивидуальное значение. Риелтор – индивидуальный предприниматель или юридическое лицо, профессионально занятое посредничеством при заключении сделок купли-продажи, аренды коммерческой и жилой недвижимости путём сведения партнёров по сделке и) получения комиссионных. Маклер — торговый посредник. Как правило, маклер профессионально занимается посредничеством при покупке и продаже товаров, ценных бумаг, услуг, страховании, способствует заключению сделок купли-продажи путём сведения партнёров. В СССР в основном были известны квартирные маклеры. По мере становления капитализма большинство из них начали заниматься недвижимостью. Брокер – юридическое или физическое лицо, выполняющее посреднические функции между продавцом и покупателем.

Основной упор у агентств по продаже и аренде недвижимости делается на поиск новых предложений, который будут подходить под их параметры запроса. Считается, что чем быстрее агентство заключит договоров на оказание услуг по продаже, аренде, поиску недвижимости, тем больше шанс качественно оказания данной услуги и реализации договоренностей в договоре по реализации услуг.

Основными задачами считаются быстрое реагирование на появление объявлений по продаже, либо аренде недвижимости, а также быстрое уведомление клиента о новых объявлениях, которые будут подходить под его критерии поиска.

1. Постановка задачи

Для того, чтобы чётко поставить задачу, необходимо вернуться к названию курсового проекта – «Телеграмм-бот помощник в сфере недвижимости».

Как уже было сказано ранее, основой, фундаментом любого агентства недвижимости, является быстрое реагирование на появление новых объявлений.

Также важно эффективно собирать и хранить объявления с разных платформ, для их последующего анализа.

Также, для дополнительно функционала и более дружелюбного интерфейса взаимодействия, будет реализация клиента в веб браузере.

Таким образом, основе всего вышеизложенного можно сформулировать основные требования этого курсового проекту:

- возможность регистрации и авторизации пользователя;
- разделение ролей (администратор и обычный пользователь);
- создание фильтров;
- возможность редактирования и удаления фильтра его автором;
- просмотр всех объявлений, хранящихся в базе данных веб-приложения;
- парсинг объявлений;
- анализ объявлений;
- уведомления пользователя об объявлении, если оно подходит под его критерии поиска в фильтре.

1.1 Обзор прототипов

Количество платформ, которые предоставляют услуги по размещению объявлений по продаже и аренде недвижимости, в последнее время неуклонно растет. Но для эффективного своевременного уведомления клиента, который находится в поиске наилучшего для него предложения, до сих пор требуется наличие человеческого фактора. На данный момент работники агентств по продаже и аренде недвижимости вручную обрабатывают все платформы, анализируют, и уведомляют клиента, если предложение может его заинтересовать.

Будут приведены несколько популярных сайтов и популярных телеграмм ботов, которые позволяют размещать у себя объявления по продаже и аренде недвижимости. В их число входят:

- Onliner;
- Аренда Квартир Бот;
- FlattyBy Бот.

1.1.1 Onliner

Onliner[1] – белорусский вебсайт, включающий пять основных разделов (технологии, авто, недвижимость, люди, форум). Функционирует также как маркетплейс товаров и услуг. По данным исследовательской компании Gemius, в 2020 году портал охватывал 52,08 % белорусской интернет-аудитории. Onliner.by является независимым, негосударственным медиа и позиционирует себя как платформу, на которой каждый белорус может высказать своё мнение и решить

насущные проблемы. Портал получил регистрацию СМИ 26 августа 2019 года. В 2019 году сайт имел самый высокий рейтинг свободы от пропаганды среди всех медиа страны. Согласно статистике, 60 % его аудитории представляют люди в возрасте от 25 до 34 лет. На рисунке 1.1 представлен функционал данного сайта.

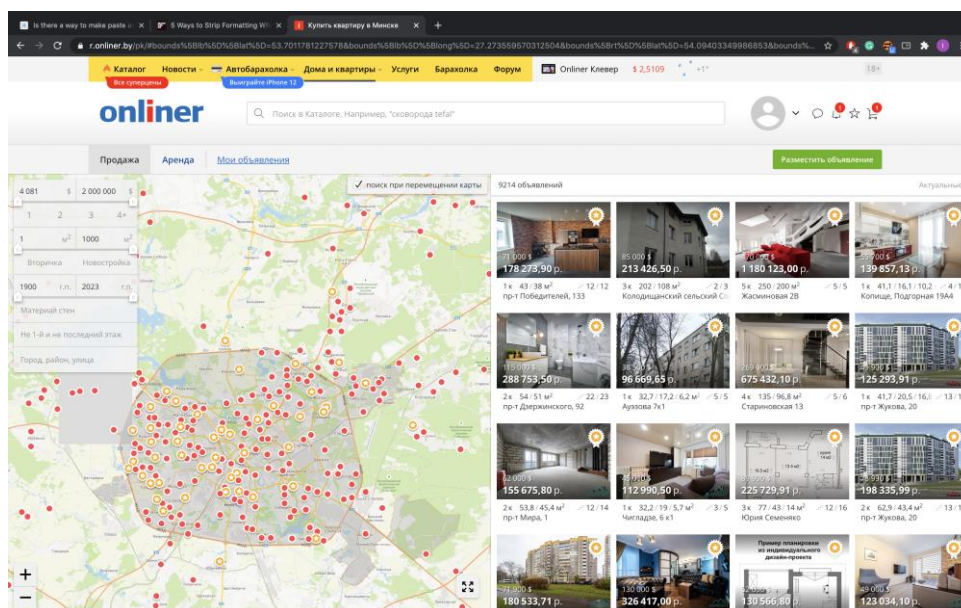


Рисунок 1.1 – «Onliner.by»

Функции и возможности раздела недвижимости платформы Onliner:

- карта с объявлениями;
- возможность разместить свое объявление;
- фильтры для поиска объявлений.

База объявлений регулярно обновляется, из-за чего обычному пользователю поиск новых объявлений значительно усложняется, ему приходится снова и снова перебирать уже просмотренные объявления, и искать среди них новые.

1.1.2 Аренда Квартир Бот

Бот написан на Ruby и «развёрнут» на бесплатном хостинге Heroku. Telegram-бот присылает уведомления через три минуты после того, как хозяин очередной квартиры (подходящей по заданным параметрам) выкладывает объявление в сеть.

Под каждым объявлением, что приходит от бота, есть кнопка «посмотреть на сайте» – она перенаправляет на специализированный ресурс. Сейчас бот использует только базу квартир на Onliner, в будущем разработчик обещает добавить и другие.

Также у бота присутствует кнопка «это агент». Работает это так: если пользователь убеждается, что объявление, которое ему прислал бот, разместило агентство, он нажимает на кнопку. 3 жалобы — и бот заносит контактный номер телефона в чёрный список.

Под каждым объявлением, что приходит от бота, есть кнопка «посмотреть на сайте» — она перенаправляет на специализированный ресурс. Сейчас бот использует только базу квартир на Onliner, в будущем разработчик обещает добавить и другие.

На рисунке 1.2 представлен пример приходящего уведомления о квартире.

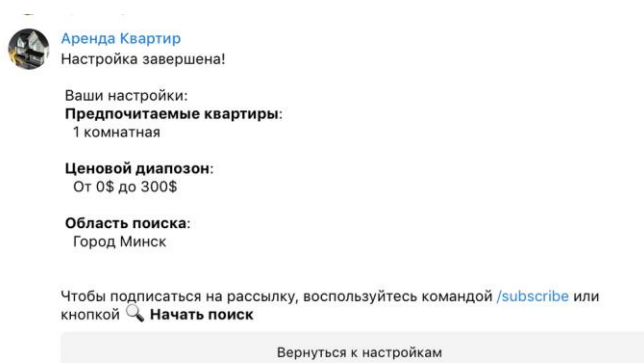


Рисунок 1.2 – «Аренда Квартир Бот»

Монетизации у программы нет, но в какой-то момент появится кнопка «поблагодарить» — чтобы пользователи при желании могли помочь проекту.

1.1.3 FlattyBy Бот

Данный бот предоставляет обширную функционал.

Индекс недоверия номеру телефона в объявлении от 0 до 5. Чем он выше, тем вероятнее, что объявление разместил агент.

Если позвонил(а) на номер, а там агент — пиши комментарий под постом, оперативно отметим недоброжелателя и не пропустим в следующий раз с этим номером.

Если объявление с ошибкой — пиши, всё исправят.

Ближайшая станция метро (если есть).

Строка — ссылка на источник.

Под каждым постом, если в объявлении была указана геолокация, можно найти ссылку на карту.

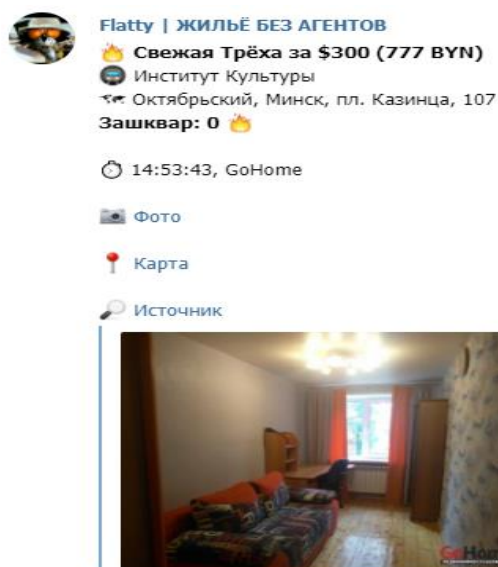


Рисунок 1.2 – «Аренда Квартир Бот»

На рисунке 1.3 предоставлен пример уведомления в телеграмме.

1.2 Алгоритмы решения

После исследования задачи рассматриваются шаги, которые требуются для решения, и порядок, в котором они должны быть выполнены. Шаги, которые необходимы для решения задачи и их последовательность, – это и есть алгоритм.

Первым шагом будет разработка архитектуры всего приложения для реализации микросервисной архитектуры. Выбор облачного провайдера, который предоставляет облачные услуги. Определиться со списком необходимых сервисов для полного функционирования приложения. Реализовать управления всей облачной инфраструктурой.

Далее необходимо реализовать базу данных. В данном курсовом проекте будет использована не реляционная NoSQL база данных DynamoDB[13]. В базе данных должны храниться фильтры пользователей: имя фильтра, критерии для поиска, идентификатор создателя фильтра. Для каждой платформы, с которой мы будем собирать объявления, будет своя коллекция, в которой будет храниться информация об объявлениях: уникальный идентификатор объявления, дата создания, дата, когда объявление надо будет удалить из базы данных, статус, дата изменения объявления, информация о самом объявлении.

На следующем шаге надо реализовать API для последующего взаимодействия клиентов с сервером. Для телеграмм-бота будет реализован REST endpoint с помощью AWS APIGateway[17]. Для браузерного клиента будет реализовано взаимодействие посредством GraphQL с помощью AWS AppSync[15] сервиса.

После следует приступить к проектированию первой клиентской части, а именно бота. Для этого потребуется настроить бот с помощью Telegram-API.

После имплементации телеграмм-бота, можно приступить к реализации браузерного клиента с помощью frontend фреймворка Angular, а также его последующей настройки C\DI с помощью Vercel[9].

Когда мы будем удовлетворены тем, как выполняется задание, мы можем снова вернуться к предыдущим шагам для их улучшения. И так этот цикл может повторяться несколько раз, пока мы не будем удовлетворены полностью. В компьютерной терминологии такой цикл называют циклом разработки программного обеспечения.

Для реализации решено использовать AWS. Надо для начала перечислить все за и против.

Самая большая сила Amazon – это доминирование на рынке публичных облаков. В своем отчете Magic Quadrant for Cloud Infrastructure as a Service, Worldwide, Gartner отмечает, что «AWS является лидером рынка облачных IaaS более 10 лет».

Одна из причин его популярности, несомненно, заключается в огромных масштабах его деятельности.

AWS обладает огромным и постоянно растущим набором доступных сервисов, а также самой разветвленной сетью мировых центров обработки данных. В отчете Gartner это резюмируется: «AWS – это наиболее зрелый, готовый к работе поставщик услуг с самыми широкими возможностями для управления большим количеством пользователей и ресурсов».

Большая слабость Amazon связана с ценой.

Несмотря на то, что AWS регулярно снижает свои цены, многим предприятиям сложно понять структуру затрат компании и эффективно управлять этими расходами при выполнении большого объема рабочих нагрузок на сервисе.

В целом, однако, эти минусы более чем перевешиваются сильными сторонами Amazon, и организации любого размера продолжают использовать AWS для самых разных рабочих нагрузок. График доминанции AWS над всеми представлен на рисунке 1.4.



Рисунок 1.4 – График популярности облачных систем.

Выбор пал на изучение AWS, так как он предоставляет самый лучший функционал среди конкурентов, имеет более 174 сервисов, и покрывает более чем 85% рынка и выгоднее всех своих конкурентов в качестве в плане цена/качество.

2. Проектирование

Данный курсовой проект предоставляет микросервисную архитектуру – принципиальная организация распределенной системы на основе микросервисов, их взаимодействия друг с другом и со средой по сети, а также принципов, направляющих проектирование архитектуры, её создание и эволюцию. Таким образом, этап проектирования можно логически разбить на три основные части:

- проектирование серверной части;
- проектирование базы данных;
- проектирование «клиента» (клиентской части).

Ниже будут рассмотрены процессы определения архитектуры и прочие характеристики каждой из этих частей.

2.1 Проектирование серверной части

Было определено, что приложение будет реализовано на микросервисной архитектуре. В качестве облачного провайдера был выбран Amazon Web Services (в дальнейшем AWS), так как данная платформа самая распространенная в мире облачная платформа с широчайшими возможностями, предоставляющая более 175 удобных, полнофункциональных сервисов для центров обработки данных по всей планете. Миллионы клиентов, в том числе стартапы, ставшие лидерами по скорости роста, крупнейшие корпорации и передовые правительственные учреждения, используют AWS для снижения затрат, повышения гибкости и ускоренного внедрения инноваций.

Serverless Framework[7] помогает разрабатывать и развертывать функции AWS Lambda[14] вместе с необходимыми ресурсами инфраструктуры AWS. Это интерфейс командной строки, который предлагает структуру, автоматизацию и передовые методы прямо из коробки, позволяя вам сосредоточиться на создании сложных, управляемых событиями, бессерверных архитектур, состоящих из функций и событий.

Бессерверная платформа отличается от других платформ, тем что:

- Она управляет вашим кодом, а также вашей инфраструктурой;
- Она поддерживает несколько языков;
- Core Concepts.

Есть ряд концепций Serverless Framework. Далее будут приведены эти концепции и какое отношение они имеют к AWS и Lambda.

Функции AWS Lambda. Это независимая единица развертывания, подобная микросервису. Это просто код, развернутый в облаке, который чаще всего пишется для выполнения одной задачи, например:

- Сохранение пользователя в базе;
- Обработка файла в базе данных;
- Выполнение запланированной задачи.

Вы можете выполнять несколько задач в своем коде, но мы не рекомендуем делать это без уважительной причины. Framework разработан, чтобы помочь вам легко разрабатывать и развертывать функции, а также управлять многими из них.

События – все, что вызывает выполнение функции AWS Lambda. К таким событиям на AWS относят следующее:

- Запрос конечной точки HTTP AWS API Gateway;
- Загрузка корзины AWS S3;
- Таймер CloudWatch[21];
- AWS SNS;
- Предупреждение CloudWatch.

Когда вы определяете событие для своих функций AWS Lambda в Serverless Framework, платформа создает любую инфраструктуру, необходимую для этого события, и настраивает ваши функции AWS Lambda для его прослушивания.

Ресурсы – это компоненты инфраструктуры AWS, которые используются вашими функциями, например:

- Таблица AWS DynamoDB;
- AWS S3 Bucket;
- AWS SNS.

Все, что можно определить в CloudFormation[18], поддерживается Serverless Framework.

Бессерверная платформа не только разворачивает ваши функции и события, которые их запускают, но также разворачивает компоненты инфраструктуры AWS, от которых зависят ваши функции.

Сервисы – это организационная единица Serverless Framework. Вы можете думать об этом как о файле проекта, хотя у вас может быть несколько сервисов для одного приложения. Здесь вы определяете свои функции, события, которые их запускают, и ресурсы, которые используют ваши функции. Службу можно описать в формате YAML или JSON, используя соответственно файл с именем `serverless.yml` или `serverless.json`.

Terraform[8] – это инструмент от компании Hashicorp, помогающий декларативно управлять инфраструктурой. В данном случае не приходится вручную создавать инстансы, сети и т.д. В консоли вашего облачного провайдера; достаточно написать конфигурацию, в которой будет изложено, как вы видите вашу будущую инфраструктуру. Такая конфигурация создается в человеко-читаемом текстовом формате. Если вы хотите изменить вашу инфраструктуру, то редактируете конфигурацию и запускаете `terraform apply`. Terraform направит вызовы API к вашему облачному провайдеру, чтобы привести инфраструктуру в соответствие с конфигурацией, указанной в этом файле

Если перенести управление инфраструктурой в текстовые файлы, то открывается возможность вооружиться всеми излюбленными инструментами для управления исходным кодом и процессами, после чего переориентируем их для работы с инфраструктурой. Теперь инфраструктура подчиняется системам контроля версий, как и исходный код, ее можно точно так же рецензировать или откатывать к более раннему состоянию, если что-нибудь пойдет неправильно.

Если вы используете Terraform для личного проекта, хранение состояния в одном `terraform.tfstate` на вашем компьютере - это нормальная практика. Но если вы будете использовать Terraform в продакшене, вы столкнетесь с несколькими проблемами:

- Общее хранилище. Чтобы использовать Terraform для обновления инфраструктуры, каждому члену команды необходим доступ к одним и тем же файлам состояния Terraform. Это означает, что вам нужно хранить эти файлы в общем месте.

- Блокировка. Когда данные передаются, вы сталкиваетесь с новой проблемой: блокировка. Без блокировки, если два члена команды одновременно запустят Terraform, это приведет к конфликтам, потере данных и повреждению файлов состояний.

- Изоляция. При внесении изменений в инфраструктуру рекомендуется изолировать различные среды. Например, при внесении изменений в среду test или среду stage вы должны быть уверены, что работа в среде prod не нарушится.

Лучший способ управления общим хранилищем файлов состояния - использовать встроенную поддержку Terraform'ом удаленных бэкендов. Бэкенд определяет, как Terraform загружает и сохраняет состояние.

Удаленные бэкенды решают такие проблемы:

- Человеческий фактор. После того, как будет настроен удаленный бэкенд, Terraform будет автоматически загружать файл состояния из этого бэкенда каждый раз, когда вы будете запускать `terraform plan` или `terraform apply`.

- Блокировка. Запустив `terraform apply`, Terraform автоматически включит блокировку. И если в это время кто-то уже запустил `terraform apply`, то вам придется ждать. Но вы можете запустить `terraform apply` с параметром `-lock-timeout=<TIME>`, чтобы Terraform подождал заданное время. Пример: `-lock-timeout=10m` сообщит Terraform подождать 10 минут.

- Шифрование секретов. Удаленные бэкенды изначально поддерживают шифрование при передаче и шифрование самого файла состояния. Более того, эти бэкенды предоставляют способы настройки разрешений доступа (например, использование политик IAM с Amazon S3 Bucket), чтобы вы могли контролировать, кто имеет доступ к вашим файлам состояния и секретам, которые они могут содержать.

Сервис Amazon Cognito[11] предоставляет решения для контроля доступа к серверным ресурсам из приложения. Это позволяет определять роли, а также назначать различные роли пользователям, чтобы приложение получало доступ только к ресурсам, доступным конкретному пользователю.

Двумя основными компонентами Amazon Cognito являются пулы пользователей и пулы удостоверений. Пулы пользователей – это каталоги пользователей, которые предоставляют возможность регистрации и входа для пользователей вашего приложения. Пулы удостоверений позволяют предоставлять пользователям доступ к другим сервисам AWS. Вы можете использовать пулы удостоверений и пулы пользователей по отдельности или вместе.

Пулы пользователей предоставляют:

- Услуги регистрации и авторизации.
- Встроенный настраиваемый веб-интерфейс для входа пользователей.
- Социальный вход через Facebook, Google, вход через Amazon и вход через Apple, а также через поставщиков удостоверений SAML и OIDC из вашего пользовательского пула.

- Управление каталогами пользователей и профили пользователей.
- Функции безопасности, такие как многофакторная аутентификация (MFA), проверка на наличие взломанных учетных данных, защита от перехвата учетной записи, а также проверка телефона и электронной почты.
- Индивидуальные рабочие процессы и миграция пользователей с помощью триггеров AWS Lambda.

На рисунке 2.1 показан типичный сценарий Amazon Cognito. Здесь цель состоит в том, чтобы аутентифицировать вашего пользователя, а затем предоставить ему доступ к другому сервису AWS.

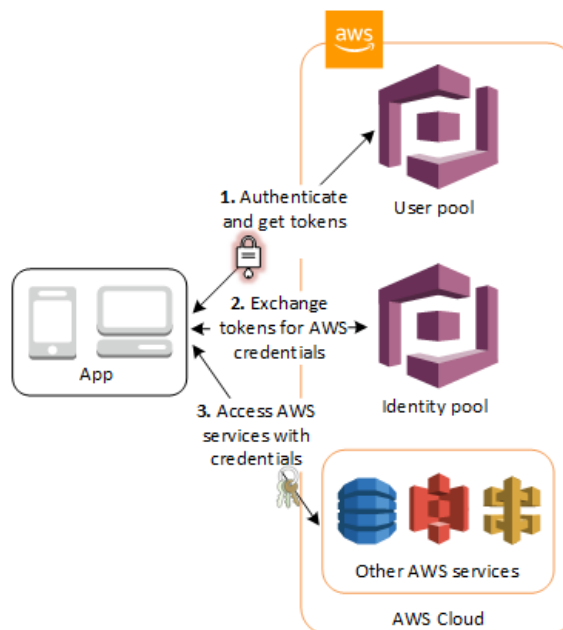


Рисунок 2.1 – сценарий Amazon Cognito

AWS Lambda – это сервис бессерверных вычислений, который запускает программный код в ответ на определенные события и отвечает за автоматическое выделение необходимых вычислительных ресурсов. Основные возможности:

- расширьте возможности сервисов AWS с помощью собственного программного кода;
- создавайте собственные серверные сервисы;
- использование собственного кода;
- полностью автоматизированное администрирование;
- встроенная отказоустойчивость;
- автоматическое масштабирование и т.д.

AWS AppSync является полностью управляемым сервисом, что упрощает разработку API GraphQL благодаря встроенной обработке сложных задач по подключению к AWS DynamoDB, Lambda и многим другим. Дополнительные кэши позволяют повысить производительность, подписки поддерживают обновления в реальном времени, а хранилища данных на стороне клиента позволяют легко сохранять синхронизацию для клиентов без постоянного подключения к Интернету. После развертывания AWS AppSync автоматически масштабирует подсистему

выполнения API GraphQL вверх или вниз в соответствии с текущим объемом запросов к API. Шаблон AWS AppSync представлен на рисунке 2.2.

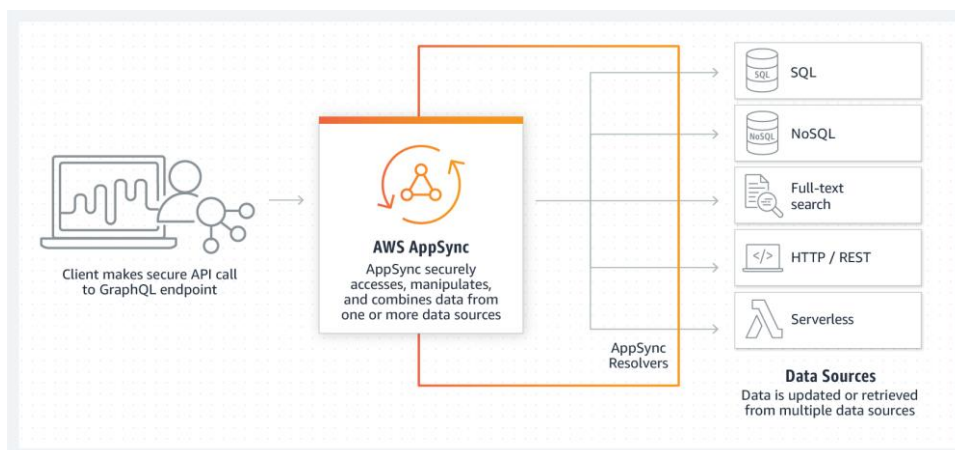


Рисунок 2.2 – Шаблон AWS AppSync.

В AWS AppSync есть два типа преобразователей, которые используют шаблоны сопоставления немного по-разному: преобразователи модулей и конвейеров. Они представлены на рисунке 2.3

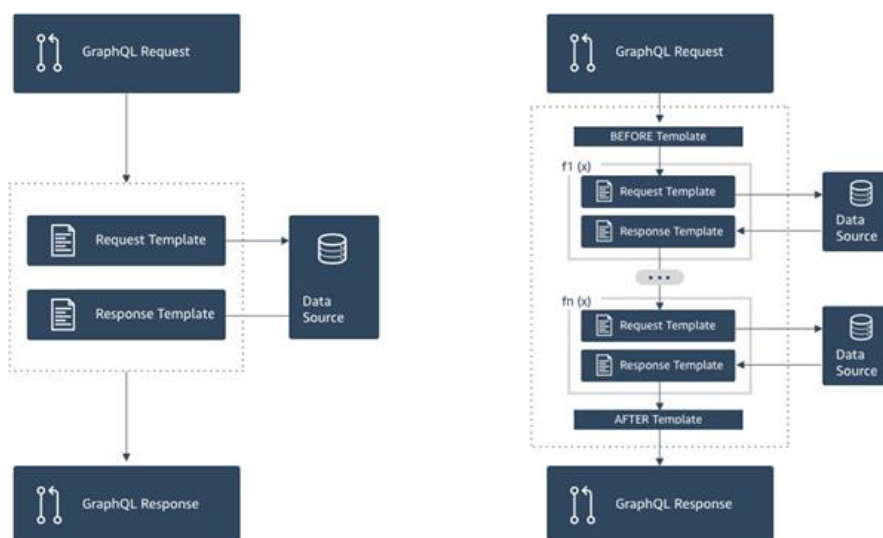


Рисунок 2.3 – Преобразователи модулей и конвейеров.

Amazon API Gateway – это полностью управляемый сервис для разработчиков, который упрощает публикацию, обслуживание, мониторинг, защиту и использование API в любых масштабах. API Gateway позволяет создавать API RESTful и WebSocket, которые являются главным компонентом приложений для двусторонней связи в режиме реального времени. AWS API Gateway поддерживает рабочие нагрузки в контейнерах и бессерверные рабочие нагрузки, а также интернет-приложения.

AWS API Gateway берет на себя все задачи, связанные с приемом и обработкой сотен тысяч одновременных вызовов API, включая управление трафиком, поддержку CORS, авторизацию и контроль доступа, регулирование количества запросов, мониторинг и управление версиями API. Работа с API Gateway

не требует минимальных платежей или стартовых вложений. Принцип работы представлен на рисунке 2.4.

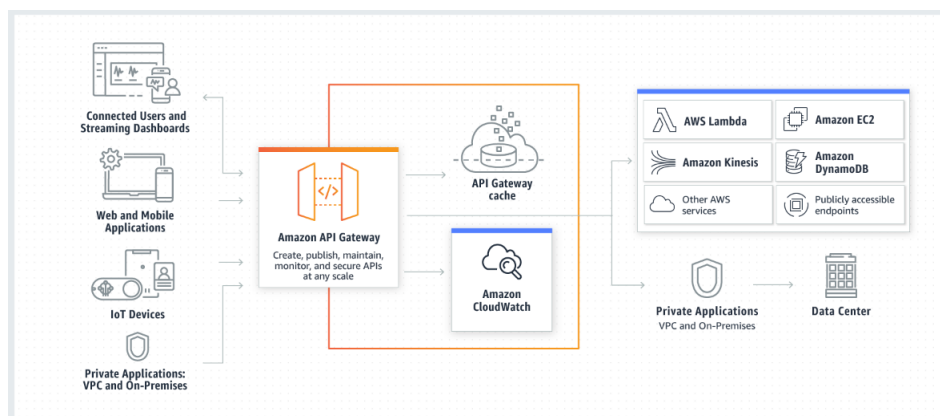


Рисунок 2.4 – Принцип работы API Gateway.

Из выше приведенных объяснений становится очевидно преимущество использования микросервисов AWS перед использованием других технологий.

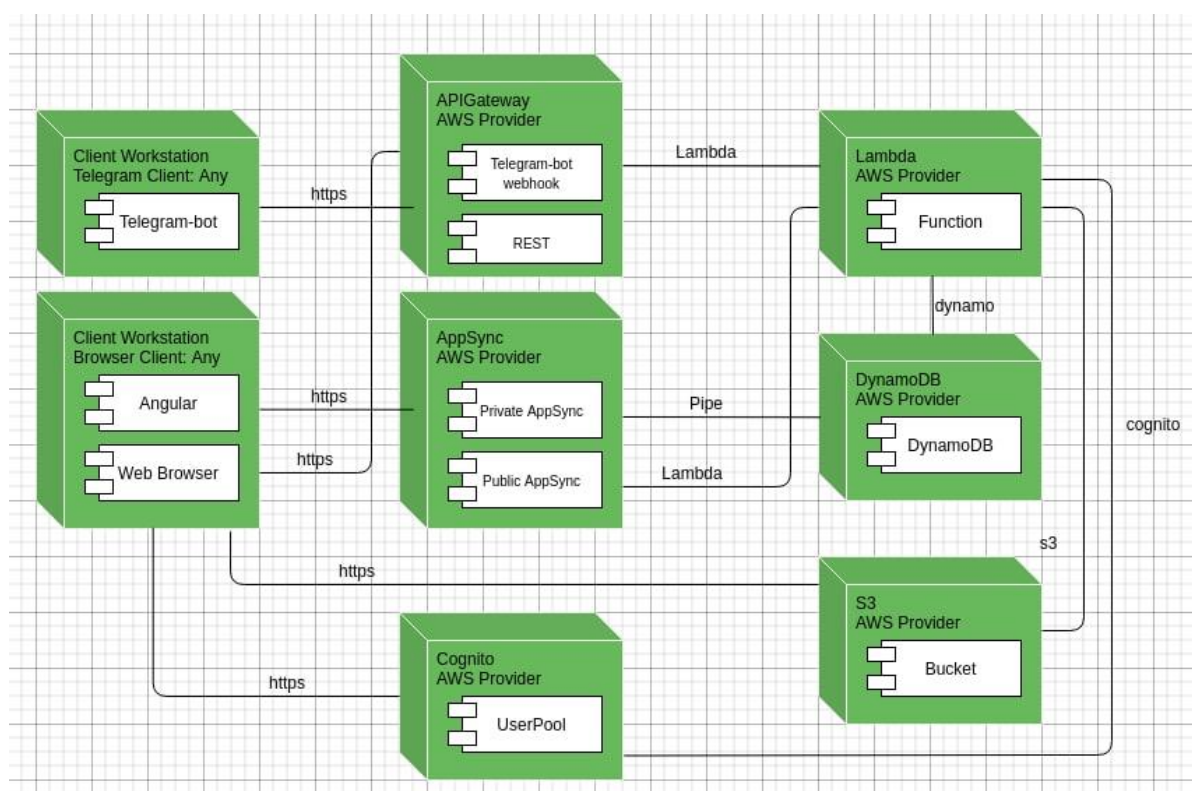


Рисунок 2.5 – Обобщенная диаграмма развертывания, совмещенная с диаграммой компонентов

Следовательно, можно подвести общую схему серверной части (рисунке 2.5) представляющую из себя диаграмму развертывания, совмещенную с диаграммой компонентов, на которой четко прослеживается взаимодействие микросервисов.

2.2 Проектирование базы данных

В качестве базы данных в данном курсовом проекте будет использована DynamoDB. Это база данных пар «ключ-значение» и документов, которая обеспечивает задержку менее 10 миллисекунд при работе в любом масштабе. Это надежная полностью управляемая база данных для приложений в масштабе всего Интернета, которая работает в нескольких регионах с несколькими ведущими серверами и обладает встроенными средствами обеспечения безопасности, резервного копирования и восстановления, а также кэширования в памяти. DynamoDB может обрабатывать более 10 трлн запросов в день и справляться с пиковыми нагрузками, превышающими 20 млн запросов в секунду.

Многие из наиболее активно развивающихся компаний в мире, например Lyft, Airbnb и Redfin, а также крупные корпорации, такие как Samsung, Toyota и Capital One, используют масштабируемый и высокопроизводительный сервис DynamoDB для выполнения критически важных рабочих нагрузок.

AWS DynamoDB – это полностью управляемая служба базы данных NoSQL, которая обеспечивает быструю и предсказуемую производительность с бесшовной масштабируемостью. Если вы разработчик, вы можете использовать Amazon DynamoDB для создания таблицы базы данных, которая может хранить и извлекать любой объем данных и обслуживать любой уровень трафика запросов. Amazon DynamoDB автоматически распределяет данные и трафик для таблицы по достаточному количеству серверов, чтобы обрабатывать объем запросов, указанный клиентом, и объем хранимых данных, сохраняя при этом стабильную и высокую производительность. Все элементы данных хранятся на твердотельных дисках (SSD).

В данном курсовом проекте будет реализовано две коллекции: TelegramUserFilters и OnlinerApartment.

OnlinerApartment:	
id	Number
expirationTime	Number
status	String
createdAt	String
updatedAt	String
apartment	Object

Рисунок 2.6 – Коллекция «OnlinerApartment»

Схематично коллекцию «OnlinerApartment» можно представить, как показано на рисунке 2.6.

Коллекция - OnlinerApartment:

- id – это Primary key, то есть hash_key Attribute;
- expirationTime – это TTL Attribute
- status – принимает значения "OLD" и "NEW" для взаимодействия с TTL;
- createdAt – время, когда была создана запись;
- updatedAt – время, когда была обновлена запись;
- apartment – объект в котором приходит вся информация с Onliner.by.

Был упомянут TTL Attribute. Он нужен для эффективного удаления объявлений из коллекции. Это удаление производится автоматически и не снижает пропускную способность бд и эти вызовы бесплатны, в отличие от тех, когда мы будем вручную выполнять Delete операции.

TelegramUserFilters представлена на рисунке 2.7

chatId	String	
filterName	String	
filter	city	String
	currency	String
	filterName	String
	maxPrice	Number
	minPrice	Number
	roomsNumber	Number
createdAt	String	
updatedAt	String	

Рисунок 2.7 – Коллекция «TelegramUserFilters»

На уровне взаимодействия кода и бд существует следующая логика:

- chatId – идентификатор чата в телеграмме между пользователем и ботом;
- filterName – имя фильтра;
- createdAt – время, когда была создана запись;
- updatedAt – время, когда была обновлена запись;
- filter – хранит объект, в котором следующие поля:
 - city – город, по которому поиск;
 - currency – валюта, в которой будет чекаться min/max Price;
 - filterName – тот же filterName что и range_key;
 - minPrice – минимальная цена для фильтрации объявлений;
 - maxPrice – максимальная цена для фильтрации объявлений;
 - roomsNumber – сколько комнат, для фильтрации объявлений.

2.3 Проектирование клиентской части

Для проектирования телеграмм-бота, архитектура которого представлена на картинке 2.8, использовался Telegram API, а также был разработан CLI интерфейс, для взаимодействия с пользователем.

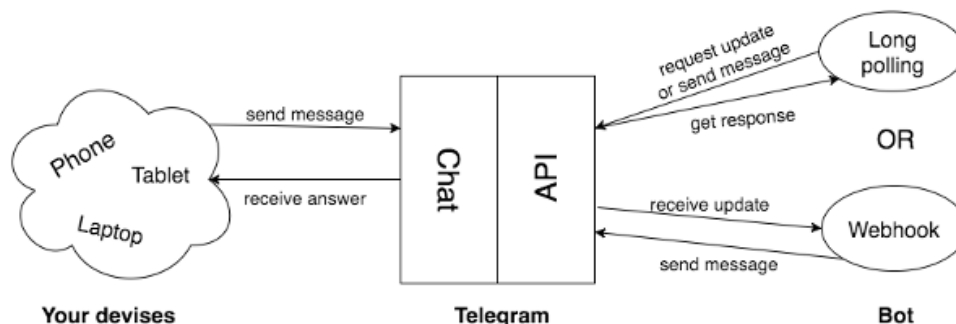


Рисунок 2.8 – Архитектура телеграмм-бота

Bot API – это интерфейс основанный на HTTP, созданный для разработчиков, заинтересованных в создании ботов для Telegram.

Боты Telegram – это особые учетные записи, для которых не требуется дополнительный номер телефона. Эти учетные записи служат интерфейсом для кода, выполняемого где-то на вашем сервере.

Чтобы использовать это, вам не нужно ничего знать о том, как работает протокол шифрования MTProto - промежуточный сервер на стороне Telegram будет обрабатывать все шифрование и связь с Telegram API за вас. Вы общаетесь с этим сервером через простой HTTPS-интерфейс, который предлагает упрощенную версию Telegram API.

Принцип взаимодействия через Telegram API представлен на рисунке 2.9.

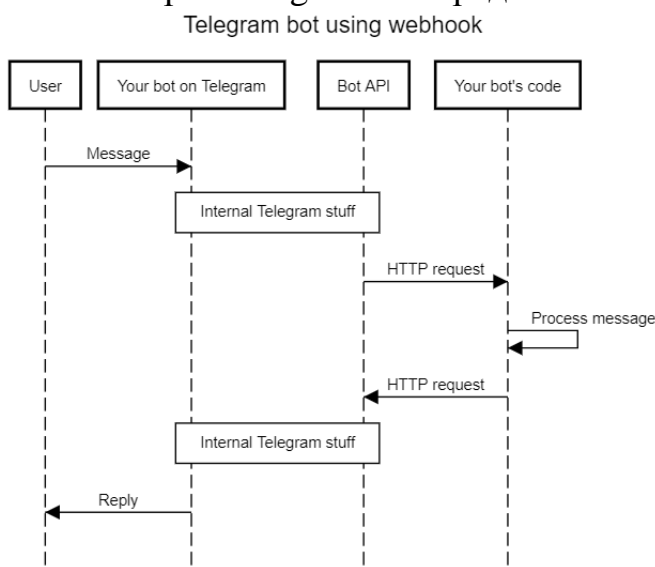


Рисунок 2.9 – Диаграмма последовательности.

Пользователи могут взаимодействовать с ботами двумя способами:

- отправляйте сообщения и команды ботам, открывая с ними чат или добавляя их в группы;
- отправляйте запросы прямо из поля ввода, вводя @username бота и запрос.

Это позволяет отправлять контент от встроенных ботов прямо в любой чат, группу или канал.

Сообщения, команды и запросы, отправленные пользователями, передаются в программное обеспечение, работающее на ваших серверах. Наш промежуточный сервер обрабатывает все шифрование и связь с Telegram API за вас. Вы общаетесь с этим сервером через простой HTTPS-интерфейс, который предлагает упрощенную версию Telegram API. Мы называем этот интерфейс нашим Bot API.

Клиентская часть веб-приложения будет реализована с помощью Angular. Angular – JavaScript-библиотека с открытым исходным кодом для разработки пользовательских интерфейсов. На клиентской части представлен минимальный функционал для тестирования и взаимодействия с Onliner API.

Angular позволяет вам из "коробки" создавать большие и сложные по части бизнес-логики приложения. Angular было полным переосмыслением AngularJS, наверное, это было самое болезненное, но оно того стоило, сам фреймворк стал куда чище и гибче, более enterprise-подобным и с этой точки зрения обладает высокой масштабируемостью.

Какие плюсы можно выделить:

- поддержка Google, Microsoft;
- инструменты разработчика (CLI);
- единая структура проекта;
- TypeScript из "коробки" (вы можете писать строго типизированный код);
- реактивное программирование с RxJS;
- единственный фреймворк с Dependency Injection из "коробки";
- шаблоны, основанные на расширении HTML и т.д.

Также будет использован NgRX – это группа библиотек, «вдохновленная» библиотекой Redux, которая, в свою очередь, «вдохновлена» шаблоном Flux. Это означает, что шаблон Redux является упрощенной версией шаблона Flux, а NGRX является версией шаблона redux с использованием Angular и RxJS.

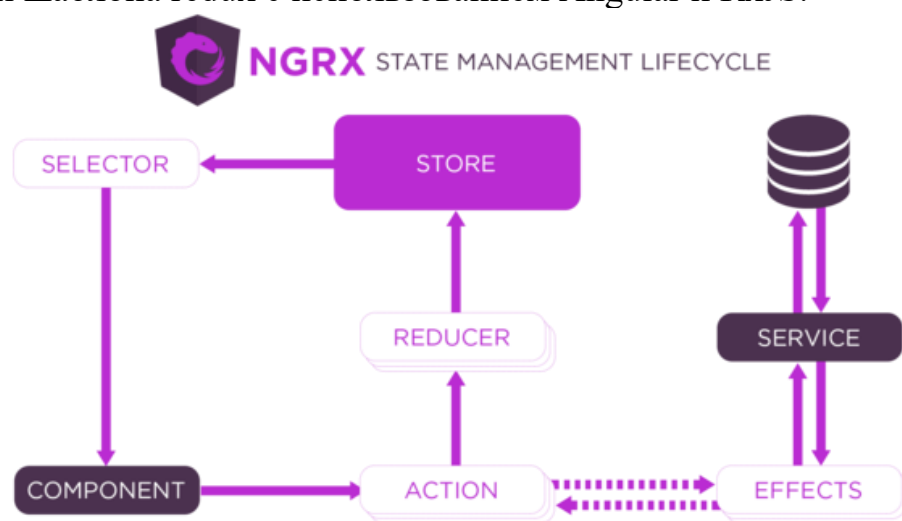


Рисунок 2.10 – Жизненный цикл NgRX.

На рисунке 2.10 представлен жизненный цикл NgRX.

Это основные строительные единицы жизненного цикла NgRX:

- действия (actions);
- редукторы (reducers);
- эффекты (effects);
- селекторы (select);
- хранилище (store).

Редукторы – это чистые функции, принимающие два аргумента: предыдущее состояние (state) и действие (action). Когда отправляется действие, NgRX проходит через все редукторы, передавая в качестве аргументов предыдущее состояние и действие, в порядке, в котором редукторы были созданы, пока не найдет обработчик для этого действия.

Эффекты прослушивают отправленные действия, и, также как и редукторы, проверяют, имеются ли у них обработчик для них. Затем выполняется побочный эффект. Обычно это получение или отправка данных посредством API.

Хранилище – это объект (экземпляр класса NgRX/Store), который объединяет вещи, о которых мы упоминали ранее (действия, редукторы, селекторы). Например, когда через его функции отправляется действие, то хранилище находит и выполняет соответствующий редуктор.

Хранилище NgRX предоставляет нам функцию «селектор» для получения фрагментов нашего хранилища. А если нам нужно применить некоторую логику к этому фрагменту перед использованием данных в компонентах то здесь нам понадобятся селекторы.

Они позволяют нам обрабатывать данные фрагмента состояния вне компонента. Функция «select» хранилища принимает в качестве аргумента чистую функцию, она и является нашим селектором.

В объекте хранилища (store object) у вас есть функция для отправки/запуска (dispatch/trigger) действий. Действия — это классы, которые реализуют интерфейс действий NGRX/Actions. Эти классы действий имеют два свойства:

- тип (type): это обычная строка только для чтения, описывающая, что означает действие;
- полезная нагрузка (payload): тип этого свойства зависит от того, какой тип данных это действие необходимо отправить редуктору (reducer).

Составив все необходимые требования, определив алгоритмы решения, можно приступить к разработке веб-приложения и телеграмм бота.

3. Разработка

Составив все необходимые требования, определив алгоритмы решения, можно приступить к разработке веб-приложения «Телеграмм-бот помощник в сфере недвижимости».

3.1 Настройка среды разработки

Основным языком разработки выбран TypeScript[2]. Файл с конфигурацией TypeScript представлен на рисунке 3.1.

```
{
  "compilerOptions": {
    "removeComments": true,
    "noUnusedParameters": true,
    "sourceMap": true,
    "outDir": ".build",
    "noImplicitAny": true,
    "allowJs": true,
    "checkJs": true,
    "moduleResolution": "node",
    "module": "commonjs",
    "target": "es2017",
    "skipLibCheck": true,
    "downlevelIteration": true,
    "strictNullChecks": true,
    "noUnusedLocals": true,
    "lib": [
      "es2017",
      "es2017.object",
      "es2015.promise",
      "es2015.symbol",
      "es2015.symbol.wellknown",
      "es2015.collection",
      "es2015.iterable",
      "dom",
      "scripthost"
    ],
    "baseUrl": "./src"
  },
  "include": [".src/**/*.ts"],
  "exclude": [
    "node_modules/**/*",
    ".serverless/**/*",
    ".webpack/**/*",
    "_warmup/**/*",
    ".vscode/**/*",
    ".src/scripts",
    ".config/webpack.config.js"
  ]
}
```

Рисунок 3.1 – Конфигурационный файл tsconfig.json.

Преимущества TypeScript. Если углубиться в технические детали, TypeScript дает вам:

- строгий набор текста (все остается так, как мы это определим);
- структурная типизация (незаменим, когда вы заботитесь о полном определении фактической структуры, которую вы используете);
- введите аннотации (удобный способ явно указать используемый тип);
- вывод типа (неявная типизация выполняется самим TypeScript).

Так же понадобится Webpack[5] который, позволяет компилировать TypeScript в JS. Этот конфигурационный файл указан в рисунке 3.2.

```
module.exports = {
  stats: {
    colors: true,
  },
  devtool: 'source-map',
  entry: entries,
  context: appDirectory,
  mode: slsw.lib.webpack.isLocal ? 'development' : 'production',
  target: 'node',
  externals: [nodeExternals({ modulesDir })], // ignore all modules in node_modules folder
  resolve: {
    modules: [resolveApp('./src'), modulesDir],
    extensions: ['.ts', '.tsx', '.js'],
  },
  module: {
    strictExportPresence: true,
    rules: [
      {
        test: /\.ts$/,
        exclude: [/node_modules/, /\.test\.ts/],
        use: {
          loader: 'ts-loader',
          options: { happyPackMode: true },
        },
      },
    ],
  },
  plugins: [
    new ForkTsCheckerWebpackPlugin({
      eslint: resolveApp('./.eslintrc.js'),
      eslintOptions: {
        cache: true,
      },
    }),
  ],
  output: {
    libraryTarget: 'commonjs',
    path: path.join(__dirname, '.webpack'),
    filename: '[name].js',
  },
};
```

Рисунок 3.2 – Конфигурационный файл webpack.config.js.

Webpack – это сборщик модулей. Он служит для упаковки кода для использования браузером. Он позволяет использовать последние возможности JavaScript с помощью Babel или использовать TypeScript и компилировать его в кроссбраузерный унифицированный код. Он также позволяет импортировать статические ресурсы в JavaScript.

Так же нам надо использовать определенный менеджер пакет такой как Yarn[6]. Yarn это новый менеджер пакетов, совместно созданный Facebook, Google, Exponent и Tilde.

Как можно прочитать в официальной документации, его целью является решение целого ряда проблем, с которыми столкнулись разработчики при использовании npm, а именно:

- установка пакетов не была достаточно быстрой и последовательной;
- существовали проблемы с безопасностью, так как npm позволяет пакетам запускать код при установке.

Независимо от того, устанавливается ли пакет с помощью npm или Yarn решаются набор определенных задач. В npm эти задачи выполняются последовательно и отдельно для каждого пакета, это значит, что пока пакет не установлен полностью, следующий пакет будет ждать. В Yarn эти операции выполняются параллельно, что улучшает производительность.

Так же был выбран определенный линтер. Линтер – полезный инструмент, который подсказывает, когда ты накосячил в коде допустил неточность, помогает поддерживать код в одном стиле и сам исправляет многие замечания.

Так как основным языком выбран TypeScript, его линтер TSLint[4] уже не используется, поэтому вместо него мы будем использовать ESLint[3]. Файл для его конфигурации представлен на картинке 3.3.

```
module.exports = {
  extends: [
    'eslint:recommended',
    'plugin:@typescript-eslint/eslint-recommended',
    'plugin:@typescript-eslint/recommended',
    'prettier/@typescript-eslint',
    'plugin:prettier/recommended',
  ],
  env: {
    node: true,
  },
  parser: '@typescript-eslint/parser',
  plugins: ['@typescript-eslint'],
  settings: {
    'import/parsers': {
      '@typescript-eslint/parser': ['.ts', '.tsx'],
    },
    'import/resolver': {
      typescript: {},
    },
  },
  parserOptions: {
    project: './tsconfig.json',
    tsconfigRootDir: './',
    sourceType: 'module',
    ecmaVersion: 2019,
  },
  rules: {
    '@typescript-eslint/no-explicit-any': 'off',
  },
};
```

Рисунок 3.3 – Файл .eslintrc.js.

ESLint это крутой инструмент, который позволяет проводить анализ качества вашего кода, написанного на любом выбранном стандарте JavaScript. Он приводит код к более-менее единому стилю, помогает избежать глупых ошибок, умеет автоматически исправлять многие из найденных проблем и отлично интегрируется со многими инструментами разработки.

3.2 Деплой и управления инфраструктурой серверной части

Для деплоя и управления были использованы следующие Framework:

- Serverless;
- Terraform.

Serverless Framework помогает разрабатывать и разворачивать функции AWS Lambda вместе с необходимыми ресурсами инфраструктуры AWS.

В приложении управление распределено между данными ресурсами: управление базой данных за Terraform, Lambda Функции с API Gateway и AppSync управляются Serverless.

На рисунке 3.4 приведен пример деплоя с помощью Servless.

```
alexander@af-Vostro-3568:~/files/university/taphut(master)$ yarn sls dev src/appsync/public/ deploy -v
yarn run v1.22.10
$ bin/slsctl dev src/appsync/public/ deploy -v
AppSync Plugin: GraphQL schema valid
Serverless: Packaging service...
Serverless: Uploading CloudFormation file to S3...
Serverless: Uploading artifacts...
Serverless: Validating template...
Serverless: Updating Stack...
Serverless: Checking Stack update progress...
CloudFormation - UPDATE_IN_PROGRESS - AWS::CloudFormation::Stack - taphut-api-public-dev
CloudFormation - UPDATE_IN_PROGRESS - AWS::AppSync::GraphQLSchema - GraphQLSchema
CloudFormation - UPDATE_IN_PROGRESS - AWS::AppSync::ApiKey - GraphQLApiKeyDefault
CloudFormation - UPDATE_COMPLETE - AWS::AppSync::ApiKey - GraphQLApiKeyDefault
CloudFormation - UPDATE_COMPLETE - AWS::AppSync::GraphQLSchema - GraphQLSchema
CloudFormation - UPDATE_COMPLETE_CLEANUP_IN_PROGRESS - AWS::CloudFormation::Stack - taphut-api-public-dev
CloudFormation - UPDATE_COMPLETE - AWS::CloudFormation::Stack - taphut-api-public-dev
Serverless: Stack update finished...
Service Information
service: taphut-api-public
stage: dev
region: us-east-1
stack: taphut-api-public-dev
resources: 10
api keys:
  None
appsync api keys:
  da2-ifkbkodsrbhlvl5aksnd34i
endpoints:
  None
appsync endpoints:
  https://gnbbtz4fsnbvzen5fmu3azqw24.appsync-api.us-east-1.amazonaws.com/graphql
functions:
  None
layers:
  None

Stack Outputs
GraphQLApiUrl: https://gnbbtz4fsnbvzen5fmu3azqw24.appsync-api.us-east-1.amazonaws.com/graphql
GraphQLApiKeyDefault: da2-ifkbkodsrbhlvl5aksnd34i
GraphQLApiId: hz2zpsdnd6veejwbwdgdevb1
ServerlessDeploymentBucketName: taphut-api-public-dev-serverlessdeploymentbucket-1dv3e1hy1vfff

Serverless: Removing old service artifacts from S3...

*****
Serverless: Update available. Run "npm install -g serverless@2.16.0" to update
You may turn on automatic updates via "serverless config --autoupdate"
*****

+ Done in 71.76s.
```

Рисунок 3.4 – Деплоя с помощью Servless.

Terraform — это инструмент от компании Hashicorp, помогающий декларативно управлять инфраструктурой.

На рисунке 3.5 приведен модуль для создания удаленного бэкенда.

```
resource "aws_s3_bucket" "taphut-tf-backend" {
  bucket = "taphut-terraform-backend-${var.env}"
  acl    = "private"

  # Enable versioning so we can see the full revision history of our state files
  versioning {
    enabled = true
  }

  tags = {
    Where_Used = "For terraform stored states"
    Environment = "Environment: ${var.env}"
  }
}

resource "aws_dynamodb_table" "taphut-tf-state" {
  name           = "taphut-terraform-state"
  billing_mode   = "PAY_PER_REQUEST"
  hash_key       = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }

  tags = {
    Where_Used = "For terraform stored states"
    Environment = "Environment: ${var.env}"
  }
}
```

Рисунок 3.5 – Модуль для создания удаленного бэкенда.

Все остальные примеры будут указаны в приложении А.

На картинке 3.6 приведен пример того, что будет, если выполнить деплой инфраструктуры с нескольких клиентов.

```
alexander@af-Vostro-5568:~/files/university/taphut/terraform/dev(master)$ aws-vault exec dev -- terraform plan
Acquiring state lock. This may take a few moments...

Error: Error locking state: Error acquiring the state lock: ConditionalCheckFailedException: The conditional request failed
Lock Info:
ID:          26e4526e-ac03-9807-13d2-6496d03aec75
Path:        taphut-terraform-backend-dev/terraform.tfstate
Operation:   OperationTypePlan
Who:         alexander@af-Vostro-5568
Version:     0.14.0
Created:     2020-12-20 17:04:18.062335195 +0000 UTC
Info:

Terraform acquires a state lock to protect the state from being written
by multiple users at the same time. Please resolve the issue above and try
again. For most commands, you can disable locking with the "-lock=false"
flag, but this is not recommended.
```

Рисунок 3.6 – Деплой инфраструктуры с нескольких клиентов.

Так как мы работаем с облачным провайдером, нам нужно как-то хранить, и управлять ключами, с помощью которых мы проходим аутентификацию и авторизацию для деплоя.

AWS Vault[30] - это инструмент для безопасного хранения учетных данных AWS и доступа к ним в среде разработки.

AWS Vault[30] хранит учетные данные IAM в защищенном хранилище ключей вашей операционной системы, а затем генерирует из них временные учетные данные для предоставления вашей оболочке и приложениям. Он разработан как дополнение к инструментам AWS CLI и учитывает ваши профили и конфигурации.

```
alexander@af-Vostro-5568:~/files/university/taphut(master)$ aws-vault list
Profile          Credentials      Sessions
=====
test             -               -
taphut-dev       taphut-dev      -
dev              dev             1608487364
prod-direct      prod-direct     -
admin            -               -
alexander@af-Vostro-5568:~/files/university/taphut(master)$
```

Рисунок 3.7 – Пример работы AWS-VAULT.

На рисунке 3.7 приведен пример работы AWS-VAULT.

Для удобной работы/депоя с Serverless Framework, был разработан Shell script, который позволяет облегчает деплой. Полный пример данного скрипта представлен в приложении 3.

3.3 Описание разработки бэка

AWS Cognito обеспечивает аутентификацию, авторизацию и управление пользователями для ваших веб-приложений и мобильных приложений. Ваши пользователи могут входить в систему напрямую, используя имя пользователя и пароль, или через третьих лиц, таких как Facebook, Amazon, Google или Apple.

Настройка AWS Cognito осуществлялась полностью с помощью браузерного клиента AWS Console.

User Pools – это управляемый сервис, обеспечивающий регистрацию, аутентификацию (подтверждение подлинности) пользователей и хранение их учетных записей. Его пример отображен на рисунке 3.8.

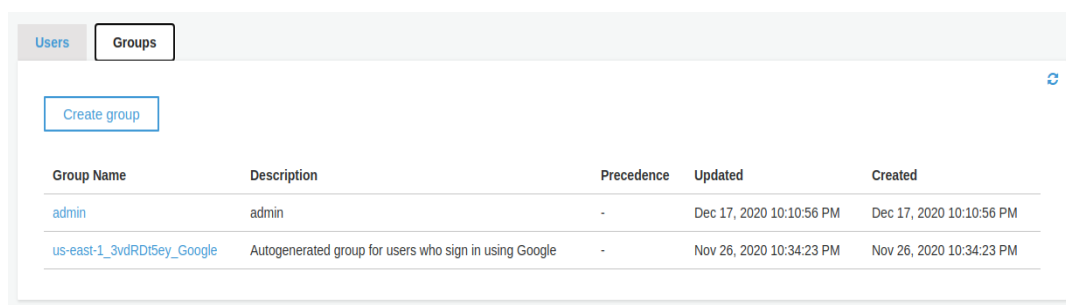
The screenshot displays the configuration page for an AWS Cognito User Pool. The settings are organized into several sections, each with a title and a list of parameters. The parameters are as follows:

- Pool Id:** us-east-1_3vdRDt5ey
- Pool ARN:** arn:aws:cognito-idp:us-east-1:337219789850:userpool/us-east-1_3vdRDt5ey
- Estimated number of users:** 13
- Required attributes:** email
- Alias attributes:** none
- Username attributes:** none
- Enable case insensitivity?:** Yes
- Custom attributes:** [Choose custom attributes...](#)
- Minimum password length:** 8
- Password policy:** uppercase letters, lowercase letters, special characters, numbers
- User sign ups allowed?:** Users can sign themselves up
- FROM email address:** Default
- Email Delivery through Amazon SES:** No
- Note:** You have chosen to have Cognito send emails on your behalf. Best practices suggest that customers send emails through Amazon SES for production User Pools due to a daily email limit. [Learn more about email best practices.](#)
- MFA:** [Enable MFA...](#)
- Verifications:** Email
- Advanced security:** [Enable advanced security...](#)
- Tags:** [Choose tags for your user pool](#)
- App clients:** test_client
- Triggers:** [Add triggers...](#)

Рисунок 3.8 – Пример информации в User Pools.

Кроме этого, следовало бы привести пример групп пользователей, которые присутствуют в нашем приложении.

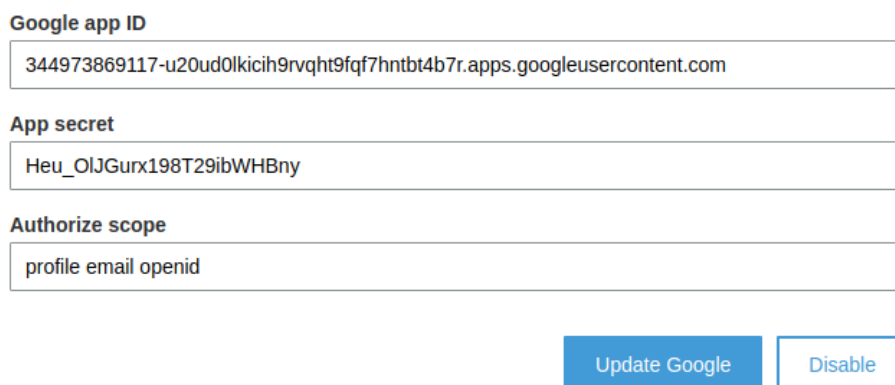
Группы пользователей в User Pools представлены на рисунке 3.9.



Group Name	Description	Precedence	Updated	Created
admin	admin	-	Dec 17, 2020 10:10:56 PM	Dec 17, 2020 10:10:56 PM
us-east-1_3vdRDt5ey_Google	Autogenerated group for users who sign in using Google	-	Nov 26, 2020 10:34:23 PM	Nov 26, 2020 10:34:23 PM

Рисунок 3.9 – Группы пользователей.

OAuth – открытый протокол авторизации, который позволяет предоставить третьей стороне ограниченный доступ к защищённым ресурсам пользователя без необходимости передавать ей логин и пароль.



Google app ID
344973869117-u20ud0lkic9rvqht9fqf7hntbt4b7r.apps.googleusercontent.com

App secret
Heu_OIJGurx198T29ibWHBny

Authorize scope
profile email openid

Update Google Disable

Рисунок 3.10 – Пример подключения OAuth.

Пример подключения OAuth представлен на рисунке 3.10.

Так же как было сказано в разработке базы данных мы использовали DynamoDB и описывали 2 коллекции.

```
export interface IOnlinerApartment {
  id: number;
  price: {
    amount: string;
    currency: OnlinerCurrencies;
    converted: {
      [key: string]: {
        amount: string;
        currency: OnlinerCurrencies;
      };
    };
  };
  rent_type: OnlinerRentType;
  location: IOnlinerApartmentLocation;
  photo: string;
  contact: {
    owner: boolean;
  };
  created_at: string;
  last_time_up: string;
  up_available_in: number;
  url: string;
}
```

Рисунок 3.11 – Описание коллекции IOnlinerApartment.

На рисунке 3.11 описание коллекции IOnlinerApartment из базы данных.

На рисунке 3.12 описание коллекции TelegramUserFilters из базы данных.

```
export interface IFilter {
  filterName: string;
  city?: string;
  currency?: Currency;
  minPrice?: number;
  maxPrice?: number;
  roomsNumber?: number;
}
```

Рисунок 3.12 – Описание коллекции TelegramUserFilters.

Onliner-Crawler сервис, который будет собирать данные с onliner, проверять их и отправлять новые данные в DynamoDB для следующей обработки. Он запускается каждые 15 минут. Его описание представлено на рисунке 3.13

```
functions:
  onliner-crawler:
    handler: onliner-crawler.handler
    description: Taphut telegram-bot webhook
    memorySize: 512
    timeout: 60
    events:
      - schedule:
          name: onliner-crawler-event
          description: 'Run the Onliner Crawler to fetch data from Onliner'
          rate: rate(15 minutes)
```

Рисунок 3.13 – Описание Onliner-Crawler сервиса.

Обобщенный алгоритм, по которому работает Onliner-Crawler сервис:

1. Совершает первоначальный запрос на Onliner API, получаем количество элементов и информацию для пагинации.
2. Формируется массив запросов к Onliner API с параметрами пагинации.
3. Формируется параллельное выполнение запросов к Onliner API batch-ами по 10 штук, и timeout = 2500ms, объявления из ответов запросов складываются в объект apartmentMap типа ключ значение, где ключ – это уникальный идентификатор объявления с Onliner API, а значение - само объявление.
4. Для оптимизации уменьшения количества запросов в БД, мы начинаем выполнять SCAN объявлений из нашей коллекции OnlinerApartment, и удаляем объявления из apartmentMap, которые пришли из нашей коллекции (так как приходилось бы для каждого элемента совершать операцию PutItem с условием выполнения запроса, что безусловно занимает большее количество обращений в БД, чем обычный scan и сделать обработку данных с помощью Lambda функции).
5. Конечный сформированный/обработанный объект apartmentMap, мы начинаем перебирать по ключу и заносить данные в нашу коллекцию OnlinerApartment со status: "NEW".

AWS AppSync является полностью управляемым сервисом, что упрощает разработку API GraphQL благодаря встроенной обработке сложных задач по подключению к AWS DynamoDB, Lambda и многим другим источникам данных.

GraphQL – это язык запросов и манипуляций с открытым исходным кодом для API, а также среда выполнения для выполнения запросов с существующими данными. Пример объявления GraphQL в AppSync представлен на рисунке 3.14.

```
# ...
mappingTemplatesLocation: .
mappingTemplates:
  - dataSource: OnlinerApartment
    type: Query
    field: onlinerApartments
    request: apartments/online/request.vtl
    response: apartments/online/response.vtl
# ...
```

Рисунок 3.14 – Объявление GraphQL в AppSync.

Так же следовало бы представить схему GraphQL (рисунок 3.15).

```
schema {
  query: Query
}

type Query {
  # get onliner apartments
  onlinerApartments(limit: Int = 10, nextToken: String = null): OnlinerApartmentRowWithPagination
}
```

Рисунок 3.15 – Схема GraphQL.

Следом была произведена разработка telegram-bot сервиса.

Важно отметить, что была выбрана реализация Telegram-bot именно через webhook, так long polling запросы требуют открытого соединения, что не целесообразно для AWS Lambda. Так как мы платим за время выполнения функции и максимально допустимое время выполнения функции составляет не более 15 минут. Его конфигурация представлена на рисунке 3.16.

```
functions:
  webhook:
    handler: webhook.handler
    description: Taphut telegram-bot webhook
    memorySize: 256
    timeout: 30
    events:
      - http:
          path: bot-api
          method: post
          cors: true
```

Рисунок 3.16 – Конфигурация telegram-bot сервиса.

Конфигурация telegram-bot сервиса представлена на рисунке 3.14.

Реализация entry point для данного сервиса представлена в приложении Б.

Для реализации интерфейса взаимодействия пользователя с приложением, я выбрал вариант разработки CLI интерфейса, и потом посредством его парсить входящие данные из бота, и возвращать результат.

Для реализации CLI интерфейса взаимодействия был выбран пакеты `yargs`.

`Yargs` - помогает создавать интерактивные инструменты командной строки, анализируя аргументы и создавая элегантный пользовательский интерфейс.

```
export type CustomExtend = { [key: string]: Options };
export type CustomArgv<O extends CustomExtend> = Arguments<InferredOptionTypes<O>> & Context;
export type CustomArgvHandler<O extends CustomExtend> = (argv: CustomArgv<O>) => void;

function buildResponder(token: string, chat_id: string): any {
  return function (msg: any): any {
    sendToUser({ token, chat_id, text: msg });
    console.log(msg);
  };
}

type buildParserParams = {
  token: string;
  chatId: string;
};

function addCommands<T>({ argv, chatId }: CommandBuilderType<T>): Argv<T> {
  argv = adddGetFiltersCommand({ argv, chatId }); // filter-list
  argv = adddGetFilterByIdCommand({ argv, chatId }); // filter-get
  argv = adddCreateFilterCommand({ argv, chatId }); // filter-create
  argv = adddUpdateFilterByIdCommand({ argv, chatId }); // filter-update
  argv = adddDeleteFilterByIdCommand({ argv, chatId }); // filter-delete
  return argv;
}

export function buildParser({ token, chatId }: buildParserParams): (stringCommand: string) => void {
  const argv = yargs().scriptName('').usage('[command]');
  const parser = addCommands({
    argv,
    chatId,
  })
  .demand(1)
  .strict()
  .version('v')
  .alias('v', 'version')
  .help('h')
  .alias('h', 'help')
  .epilog('Taphut');
}
```

Рисунок 3.17 – Builder CLI parser.

Файл, в котором я представлен builder CLI parser на рисунке 3.17.

Все остальные файлы с указанием правил различных команд будут изложены в приложении В.

`Yargs` пакет автоматически потом дополнит такие команды как `help`, основываясь на моих командах: `filter-list`, `filter-get`, `filter-create`, `filter-update`, `filter-delete`. Так же, был переопределил метод ответа CLI парсера, вместо логирования в консоль, конечный ответ вызовет функцию, которая отправит запрос на Telegram-API и пользователь получит ответ. Далее, мы обратились к `@BotFather` для создания телеграмм-бота, установили полученный токен в AWS System Manager[19] Parameter Store. И сконфигурировали webhook нашего бота на endpoint APIGateway, который передаст обработку запроса уже в нашу Lambda функцию.

Далее разработаем telegram-notifier сервис.

Обобщенный алгоритм:

1. Scan все объявления из коллекции OnlinerApartment со статусом NEW;
2. Scan все фильтры из DynamoDB коллекции TelegramUserFilters;
3. для каждого объявления проверяем фильтр:
 - 3.1. инициализируем пустой массив notifications;
 - 3.2. инициализируем и формируем объект filterMapObj типа { [key: chatId]: arrayOfChatFilters[] };
 - 3.3. проходим по массиву notifications:
 - 3.3.1. проходим по ключам объекта filterMapObj и достаём массивы фильтров в chatFilters;
 - 3.3.2. проходим по массиву фильтров chatFilters;
 - 3.3.3. если объявление подходит под фильтр, то заносим идентификатор чата и объявление в массив notifications и переходим к следующему ключу объектов filterMapObj;
 4. Запускаем параллельную обработку запросов массива notifications, отсылаем пользователю объявление с помощью sendToUser(filter.chatID, convertApartmentToMsg(apartment));
 5. Изменяем статус всех объявлений с NEW на OLD;

```
functions:
  telegram-notifier:
    handler: telegram-notifier.handler
    description: Taphut telegram-bot webhook
    memorySize: 256
    timeout: 300
    events:
      - schedule:
          name: telegram-notifier-event
          description: 'Scan all users filters and search for suitable new apartment'
          rate: rate(15 minutes)
```

Рисунок 3.18 – Serverless.yml config file.

На рисунке 3.18 приведена часть кода из serverless.yml config file, в котором настраивается schedule событие, которое будет запускать данную функцию каждые 15 минут.

Entry point для telegram-notifier сервиса представлен в приложении Г.

3.4 Разработка Angular client-а и его настройка его CI/DI

Для начала была реализована Аутентификация/Авторизация. Для этого нам понадобилась Amplify[22]. AWS Amplify – это фреймворк, который позволяет создавать и подключаться к облачным сервисам, такими как сервис авторизации, GraphQL API и Lambda функциям. Данная библиотека нужна нам для обращения к нашему AWS Cognito сервису. Полный вариант обертки с использованием Amplify указан в приложении Д.

Далее был реализован AuthenticationGuard, для защиты перехода не аутентифицированного пользователя на страницы приложения (рисунок 3.19).

```
@Injectable()
export class AuthenticationGuard implements CanActivate, CanActivateChild {

  constructor(private router: Router, private amplify: AmplifyService) { }

  async canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Promise<true | UrlTree> {
    return await this.checkAuth(route, state);
  }

  async canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Promise<true | UrlTree> {
    return await this.checkAuth(route, state);
  }

  private async checkAuth({ queryParams }: ActivatedRouteSnapshot, { url }: RouterStateSnapshot): Promise<true | UrlTree> {
    if (await this.amplify.isAuthenticated()) {
      return true;
    }

    if (url.split('?').length) {
      url = url.split('?')[0];
    }

    const redirectTo = ROUTES.signin.split('/');

    return this.router.createUrlTree(['/', ...redirectTo]);
  }
}
```

Рисунок 3.19 – AuthenticationGuard.

Его мы будем использовать в определении Routes (если не аутентифицированный пользователь захочет перейти на закрытую страницу, то он будет перенаправлен на страницу login). Файл Routes представлен на рисунке 3.20.

```
const routes: Routes = [
  {
    canActivate: [AuthenticationGuard],
    component: ShellComponent,
    path: '',
  },
  {
    component: SigninComponent,
    path: 'auth/signin',
  },
  {
    component: SignupComponent,
    path: 'auth/signup',
  },
  {
    component: ConfirmSignupComponent,
    path: 'auth/signup/confirm',
  },
  {
    component: ForgotPasswordComponent,
    path: 'auth/password/forget',
  },
  {
    component: ForgotPasswordSubmitComponent,
    path: 'auth/password/forget/confirm',
  },
  {
    component: ChangePasswordComponent,
    path: 'auth/password/change',
  },
  {
    path: '***',
    redirectTo: '',
  },
];
```

Рисунок 3.20 – Файл Routes.

В Angular-е очень удобно использовать реактивный подход с помощью Rxjs. RxJS – это библиотека для реактивного программирования, которая позволит

удобно организовать работу с событиями и асинхронным кодом, писать сложную логику декларативно. RxJS активно используется в фреймворке Angular, а также с Vue (Vue-rx) и лежит в основе реализации middleware для Redux (redux-observable) для React.

Следующим шагом было подключение UI библиотеки Angular Material плюсы которой:

- интернационализированные и доступные компоненты для всех;
- простые API с последовательным кроссплатформенным поведением;
- универсальность (предоставьте инструменты, которые помогут разработчикам создавать собственные настраиваемые компоненты с общими шаблонами взаимодействия);
- настраивается в рамках спецификации Material Design;
- создан командой Angular для интеграции с Angular, имея высокую производительность/эффективность.

На следующем шаге следует подключить Angular Material. Angular Material состоит из набора предустановленных компонентов Angular. В отличие от Bootstrap, предоставляющего компоненты, которые вы можете использовать любым способом, Angular Material стремится обеспечить расширенный и последовательный пользовательский интерфейс.

Далее в root компоненте следует реализовать header который представлен на рисунке 3.21.

```
<mat-toolbar color="primary" class="example-toolbar">
  <h1 class="header-title big">Taphut</h1>
  <span class="flex-1"></span>

  <button *ngIf="amplify.isAuthenticatedSubj | async" mat-button (click)="logout()">logout</button>
  <a
    *ngIf="amplify.isAuthenticatedSubj | async"
    mat-button
    [routerLink]="['/', 'auth', 'password', 'change']"
    >password change</a>
  >
  <a
    *ngIf="!(amplify.isAuthenticatedSubj | async)"
    mat-button
    [routerLink]="['/', 'auth', 'signin']"
    >signin</a>
  >
  <a
    *ngIf="!(amplify.isAuthenticatedSubj | async)"
    mat-button
    [routerLink]="['/', 'auth', 'signup']"
    >signup</a>
  >
</mat-toolbar>

<router-outlet></router-outlet>
```

Рисунок 3.21 – Header.

Далее надо было реализовать GraphQL сервис, который позволит нам взаимодействовать с нашим бэком посредством GraphQL API. Первым делом было решено реализовать на главной странице показ объявления по аренде недвижимости с infinite-scroll пагинацией.

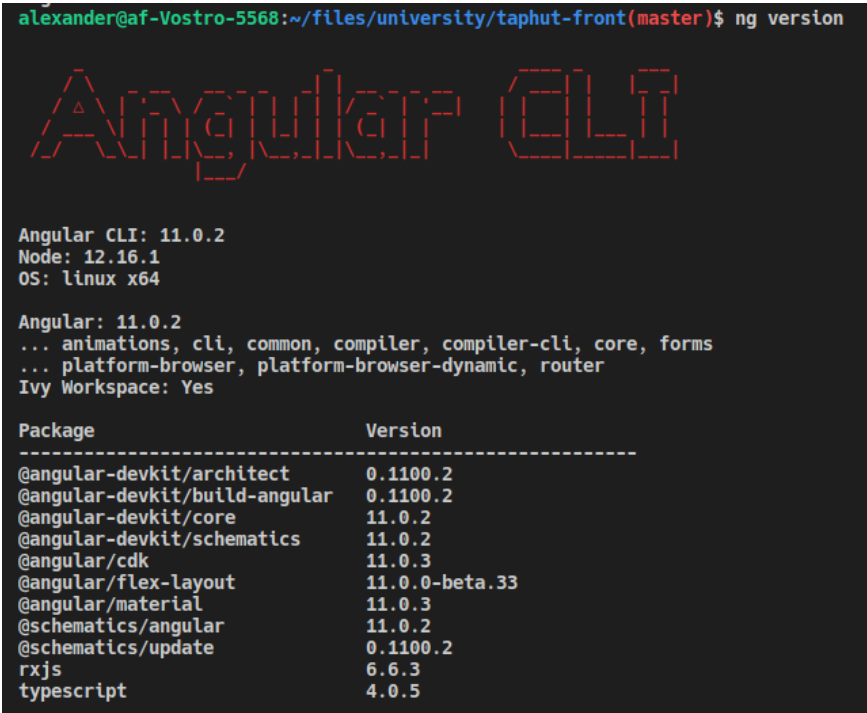
Код сервиса, который будет предоставлять API для получения объявлений указан в приложении Е.

Следующим шагом стало добавления Telegram Login Widget, чтобы в будущем позволить web-клиенту привязать к своей учетной записи чат с ботом из телеграмма. Это нужно, чтобы в будущем добавить функционал, для более удобного редактирования фильтров, просмотра объявлений и статистики с помощью браузерного клиента, при этом получая нотификации в Telegram клиенте.

Первая сложность в реализации Telegram Login Widget стало то, что в Angular нельзя просто так вставить в разметку тег `<script>`. Я создал компонент, который обходит, добавляя динамически в разметку необходимый `<script>`. Код с добавлением указан в приложении Ж.

Далее, для корректной работы Telegram Login Widget необходимо запустить приложение на https. На этом этапе я решил подключить CI/CD для более удобной работы/тестирования моего Angular приложения.

В качестве CI/CD инструмента я выбрал облачного провайдера Vercel[9]. Версия которого указана на рисунке 3.22.



```
alexander@af-Vostro-5568:~/files/university/taphut-front(master)$ ng version

Angular CLI
Angular CLI: 11.0.2
Node: 12.16.1
OS: linux x64

Angular: 11.0.2
... animations, cli, common, compiler, compiler-cli, core, forms
... platform-browser, platform-browser-dynamic, router
Ivy Workspace: Yes

Package                       Version
-----
@angular-devkit/architect     0.1100.2
@angular-devkit/build-angular 0.1100.2
@angular-devkit/core          11.0.2
@angular-devkit/schematics    11.0.2
@angular/cdk                  11.0.3
@angular/flex-layout           11.0.0-beta.33
@angular/material             11.0.3
@schematics/angular           11.0.2
@schematics/update             0.1100.2
rxjs                           6.6.3
typescript                     4.0.5
```

Рисунок 3.22 – Версия Angular CLI.

Vercel – это облачная платформа для статических сайтов и бессерверных функций, которая идеально подходит для вашего рабочего процесса. Он позволяет разработчикам размещать веб-сайты и веб-службы Jamstack, которые мгновенно развертываются, автоматически масштабируются и не требуют контроля, и все это без настройки.

Jamstack – это новая стандартная архитектура для Интернета. Используя рабочие процессы Git и современные инструменты сборки, предварительно обработанный контент передается в CDN и становится динамическим с помощью API и бессерверных функций. Технологии в стеке включают фреймворки JavaScript, генераторы статических сайтов, безголовые CMS и CDN.

4. Тестирование

Тестирование программного обеспечения — это процесс проверки соответствия заявленных к продукту требований и реально реализованной функциональности, осуществляемый путем наблюдения за его работой в искусственно созданных ситуациях и на ограниченном наборе тестов, выбранных определенным образом.

Ручное тестирование (мануальное тестирование, manual testing) — часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно производится тестировщиком без использования программных средств, для проверки программы или сайта путём моделирования действий пользователя. В роли тестировщиков могут выступать и обычные пользователи, сообщая разработчикам о найденных ошибках.

В моем приложении мы будем использовать как ручное, так и автоматизированное тестирование.

```
import * as _ from 'lodash';

import * as ApartmentsActions from './apartments.actions';
import { MOCK_ONLINER_APARTMENTS } from 'src/app/common/mocks';

const LIMIT = 10;

describe('Apartments Actions', () => {
  it('should create a LoadApartments action', () => {
    const expectedPayload: IApartmentsParams = { limit: LIMIT };
    const action: ApartmentsActions.LoadApartments = new ApartmentsActions.LoadApartments(
      expectedPayload
    );
    expect(action.type).toEqual(ApartmentsActions.ApartmentsTypes.GET_APARTMENTS);
    expect(_.isEqual(action.payload, expectedPayload)).toBeTruthy();
  });
  it('should create a LoadApartmentsFail action', () => {
    const expectedPayload: Error = new Error('some error');
    const action: ApartmentsActions.LoadApartmentsFail = new ApartmentsActions.LoadApartmentsFail(
      expectedPayload
    );
    expect(action.type).toEqual(ApartmentsActions.ApartmentsTypes.GET_APARTMENTS_FAIL);
    expect(action.payload).toEqual(expectedPayload);
  });
  it('should create a LoadApartmentsSuccess action', () => {
    const expectedPayload: OnlinerResData = _.cloneDeep(MOCK_ONLINER_APARTMENTS);
    const action: ApartmentsActions.LoadApartmentsSuccess = new ApartmentsActions.LoadApartmentsSuccess(
      expectedPayload
    );
    expect(action.type).toEqual(ApartmentsActions.ApartmentsTypes.GET_APARTMENTS_SUCCESS);
    expect(action.payload).toEqual(expectedPayload);
  });
});
```

Рисунок 4.1 – Тестирование Action-ов.

Для каждой части приложения было написано определенное количество тестов. На рисунке 4.1 скрин кода теста, который проверяет корректность выполнения Action-ов на frontend, где используется NgRX.

Для тестирования использовались такие инструменты как Jasmine и Karma

Jasmine — это, по факту, фреймворк для написания тестов Angular. Это фреймворк для тестирования, использующий нотации behavior-driven.

Тест Jasmin состоит из минимум двух элементов: функция describe (сюет тестов) и функция it (сам тест). Обычно с помощью describe показывают функцию, на которой сосредоточены — например, createCustomer(). Внутри сюета создается несколько тестов it. Каждый тест проверяет целевую функцию на ожидаемое поведение с разным условием.

Karma — инструмент для выполнения исходного кода с тестовым кодом в браузере. Поддерживается запуск тестов во всех браузерах, для которых настроен инструмент. Результаты отображаются в командной строке и в браузере. Так разработчик может следить за тем, какие тесты прошли, а какие упали. Karma следит за файлами и умеет перезапускать тест при их изменении.

```
import * as _ from 'lodash';
import { MOCK_ONLINER_APARTMENTS } from 'src/app/common/mocks';

import * as apartments from '../actions/apartments.actions';
import { apartmentsInitialState, apartmentsReducer } from './apartments.reducer';

const LIMIT = 10;

describe('Apartments Reducer', () => {
  it('should execute GET_APARTMENTS case', () => {
    const payload: IApartmentsParams = {
      limit: LIMIT,
    };
    const expectedState: IApartmentsState = {
      ...apartmentsInitialState,
      loading: true,
    };
    const action: apartments.LoadApartments = new apartments.LoadApartments(payload);

    const actualState: IApartmentsState = apartmentsReducer(
      _.cloneDeep(apartmentsInitialState),
      action
    );

    expect(action.type).toEqual(apartments.ApartmentsTypes.GET_APARTMENTS);
    expect(action.type).toEqual(apartments.ApartmentsTypes.GET_APARTMENTS);
    expect(_.isEqual(actualState, expectedState)).toBeTruthy();
  });

  it('should execute GET_APARTMENTS_SUCCESS case', () => {
    const payload: OnlinerResData = MOCK_ONLINER_APARTMENTS;
    const expectedState: IApartmentsState = {
      ..._.cloneDeep(apartmentsInitialState),
      loading: false,
      data: MOCK_ONLINER_APARTMENTS,
    };
    const action: apartments.LoadApartmentsSuccess = new apartments.LoadApartmentsSuccess(payload);

    const actualState: IApartmentsState = apartmentsReducer(
      _.cloneDeep(apartmentsInitialState),
      action
    );
```

Рисунок 4.2 — Тестирование Reducer-ов.

На рисунке 4.2 скрин кода теста, который проверяет корректность выполнения Reducer-ов на frontend, где используется NgRX.

На рисунке 4.3 скрин кода теста, который проверяет корректность выполнения Selector-ов на frontend, где используется NgRX.

```
import * as _ from 'lodash';

import { apartmentsInitialState } from '../reducers';
import {
  getApartmentsData,
  getApartmentsLoading,
  getApartmentsError,
} from './apartments.selectors';

describe('Apartments Selectors', () => {
  const appState: IAppState = {
    apartmentsState: _.cloneDeep(apartmentsInitialState),
  };

  it('should return apartments data from AppState', () => {
    expect(_isEqual(getApartmentsData(appState), { token: undefined, data: [] })).toBeTruthy();
  });

  it('should return apartments loading from AppState', () => {
    expect(
      _isEqual(getApartmentsLoading(appState), appState.apartmentsState.loading)
    ).toBeTruthy();
  });

  it('should return apartments error from AppState', () => {
    expect(_isEqual(getApartmentsError(appState), appState.apartmentsState.error)).toBeTruthy();
  });
});
```

Рисунок 4.3 – Тестирование Selector-ов.

Подводя итог при вызове команды “ng test” происходит проверка всех элементов, которые были описаны выше, а также проверка создания всех важных элементов что мы можем увидеть на рисунке 4.4.

```
alexander@aaf-Vostro-5568:~/files/university/taphut-front(feature/store)$ ng test
* Generating browser application bundles (phase: building)...27 12 2020 16:02:23.777:WARN [karma]: No captured browser, open http://localhost:9876/
27 12 2020 16:02:23.782:INFO [karma-server]: Karma v5.1.1 server started at http://localhost:9876/
27 12 2020 16:02:23.783:INFO [launcher]: Launching browsers Chrome with concurrency unlimited
27 12 2020 16:02:23.786:INFO [launcher]: Starting browser Chrome
✓ Browser application bundle generation complete.
27 12 2020 16:02:49.660:WARN [karma]: No captured browser, open http://localhost:9876/
27 12 2020 16:02:50.318:INFO [Chrome 87.0.4280.88 (Linux x86_64)]: Connected on socket 0kn-A1LVEjCaIL5YAAAA with id 23381997
ERROR: 'NG0303: Can't bind to 'formGroup' since it isn't a known property of 'form'.'
Chrome 87.0.4280.88 (Linux x86_64): Executed 0 of 19 SUCCESS (0 secs / 0 secs)
ERROR: 'NG0303: Can't bind to 'formGroup' since it isn't a known property of 'form'.'
Chrome 87.0.4280.88 (Linux x86_64): Executed 8 of 19 SUCCESS (0 secs / 0.372 secs)
ERROR: 'NG0304: 'app-telegram' is not a known element:
1. If 'app-telegram' is an Angular component, then verify that it is part of this module.
2. If 'app-telegram' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.'
Chrome 87.0.4280.88 (Linux x86_64): Executed 9 of 19 SUCCESS (0 secs / 0.414 secs)
ERROR: 'NG0304: 'app-telegram' is not a known element:
1. If 'app-telegram' is an Angular component, then verify that it is part of this module.
ERROR: 'NG0303: Can't bind to 'infiniteScrollDistance' since it isn't a known property of 'div'.'
Chrome 87.0.4280.88 (Linux x86_64): Executed 9 of 19 SUCCESS (0 secs / 0.414 secs)
ERROR: 'NG0303: Can't bind to 'infiniteScrollThrottle' since it isn't a known property of 'div'.'
Chrome 87.0.4280.88 (Linux x86_64): Executed 9 of 19 SUCCESS (0 secs / 0.414 secs)
Chrome 87.0.4280.88 (Linux x86_64): Executed 19 of 19 SUCCESS (0.641 secs / 0.512 secs)
TOTAL: 19 SUCCESS
```

Рисунок 4.4 – Общее тестирование.

Для тестирования backend части использовался Jest. Jest – это фреймворк для тестирования JavaScript с акцентом на простоту.

Он не требует дополнительных настроек, легкий в понимании и применении, а также имеет довольно хорошую документацию. Отлично подходит для проектов, использующих Node, React, Angular, Vue, Babel, TypeScript и не только.

Также он имеет открытый исходный код. Он позволяет вам писать тесты с приемлемым, знакомым и функциональным API, и быстро достигать желаемых результатов.

```
import { eachLimit, sleep, rejectAfter } from './async';

async function tester(x: number) {
  await sleep(Math.random() * 10);
  return 2 * x;
}

async function testerFast(x: number) {
  // sleep for just 1ms
  await sleep(1);
  return 2 * x;
}

async function testerBad() {
  await sleep(10);
  throw new Error('I was naughty!');
}

test('eachLimit', async () => {
  const input = [...Array.from(Array(100).keys())];
  const res = await eachLimit(input, tester);
  expect(res).toEqual(input.map((x) => 2 * x));

  const res2 = await eachLimit(input, testerFast);
  expect(res2).toEqual(input.map((x) => 2 * x));

  await expect(eachLimit([2000], sleep, 10, 10)).rejects.toThrow;
  const res3 = await eachLimit([100], sleep, 10, 2000);
  expect(res3).toEqual([undefined]);
  await expect(eachLimit(input, testerBad)).rejects.toThrow;
});

test('rejectAfter', async () => {
  try {
    await rejectAfter(1);
  } catch (e) {
    expect(e.name).toEqual('RejectAfter');
  }
  expect.assertions(1);
});
```

Рисунок 4.5 – Общее тестирование.

Тест на рисунке 4.5 проверяет нашу программу на прерывания и ожидания части запросов с целью продолжения, так как программа рассчитана на большое количество пользователей, которые могут отправить неограниченное количество запросов, которые не могут обрабатывать все сразу. Следовательно часть запросов должна ждать пока не выполняются предыдущие, а затем отработать корректно.

Так как мы используем AWS то на нас так же ложится часть ограничений, которые были обозначены системой и следующие тесты, созданы как раз проверять корректность всех данных, которые мы вводим или передаем.

```
import * as date from './date';
import { getISODate } from './date';

test('getLastWeekRange', () => {
  const d = new Date('2018-05-01');
  const range = date.getLastWeekRange(d);
  expect(range).toEqual({
    start: '2018-04-22T00:00:00.000Z',
    end: '2018-04-29T00:00:00.000Z',
  });
  const range2 = date.getLastWeekRange(d, 3);
  expect(range2).toEqual({
    start: '2018-04-18T00:00:00.000Z',
    end: '2018-04-25T00:00:00.000Z',
  });
  const range3 = date.getLastWeekRange(d, 4, -4);
  expect(range3).toEqual({
    start: '2018-04-18T20:00:00.000Z',
    end: '2018-04-25T20:00:00.000Z',
  });
});

test('getISODate()', () => {
  expect(getISODate(new Date('2018-06-25T00:00:00.000Z'))).toEqual('2018-06-25');
});

test('stripMillisISO', () => {
  expect(date.stripMillisISO('2018-04-25T20:00:00.000Z')).toEqual('2018-04-25T20:00:00Z');
});
```

Рисунок 4.6 – Обработка дат.

Тест на рисунке 4.6 проверяет приложение на обработку дат разного типа и формата.

```
import { DynamoDB_cleanRows } from './dynamodb';

test('DynamoDB_cleanRows', async () => {
  const o = {
    a: 'asdf',
    b: [] as any[],
    c: {},
    d: '',
    e: null as null,
    f: { a: '', b: 'asdf' },
    g: { a: '' },
  };
  const clean = DynamoDB_cleanRows([o]);
  expect(clean).toEqual([
    {
      a: 'asdf',
      b: [],
      c: {},
      f: { b: 'asdf' },
      g: {},
    },
  ]);
});
```

Рисунок 4.7 – Общее тестирование.

Тест на рисунке 4.7 проверяет приложение на очистку нулевых объектов, которые не поддерживает используемая в приложении база данных.

В итоге мы можем сказать, что все тесты, которые мы проводили были нужны для проверки нашего приложения на корректность работы всех частей системы и при запуске всех тестов мы можем сказать, что приложение работает как надо и по всем заданным требованиям.

```
alexander@af-Vostro-5568:~/files/university/taphut(master)$ yarn test
yarn run v1.22.10
$ jest
PASS src/utils/async.2.test.ts
PASS src/utils/date.test.ts
PASS src/utils/async.test.ts
PASS src/utils/dynamodb.test.ts

Test Suites: 4 passed, 4 total
Tests:       9 passed, 9 total
Snapshots:   0 total
Time:        4.424 s, estimated 6 s
Ran all test suites.
Done in 5.12s.
```

Рисунок 4.8 – Общий тест backend.

На рисунке 4.8 изображен вызов всех тестов и их корректное выполнение.

<p>Email</p> <hr/> <p>Email is required</p>	<p>Email</p> <hr/> <p>Email is required</p>
<p>Password</p> <hr/> <p>Password is required</p>	<p>Password</p> <hr/> <p>Password is required</p>
<p>LOGIN</p>	<p>REGISTER</p>

Рисунок 4.9 – Общий тест backend.

При использовании формы логина или регистрации (рисунок 4.9) мы можем увидеть, что работает валидация, которая указывает нам что поле должно быть заполнено.



Рисунок 4.10 – Обработка ошибки на телеграмм боте.

Также при взаимодействии с ботом мы можем что обрабатывается ошибка с вводом неверного имени фильтра.

Также при введении не веной команды или аргумента телеграмм бот будет предупреждать об этом и предлагать варианты по исправлению (рисунок 4.11).

```
fc -n filter1 --cr 1123

Taphut
[command]

Commands:
  filter-list  List all filters           [aliases: fl]
  filter-get   Get filter by name         [aliases: fg]
  filter-create Create filter             [aliases: fc]
  filter-update Update filter             [aliases: fu]
  filter-delete Delete filter            [aliases: fd]

Options:
  -h, --help    Show help                [boolean]
  -v, --version Show version number      [boolean]

Taphut

Unknown arguments: n, cr, fc
```

Рисунок 4.11 – Обработка ошибки неверного аргумента.

Так же при помощи AWS была создана форма логирования и регистрации с расширенной валидацией и регистрацией через Google (рисунок 4.12).

The image shows a web form for user registration. On the left, under 'Sign In with your social account', there is a 'Continue with Google' button. Below it, a note states: 'We won't post to any of your accounts without asking first'. On the right, under 'Sign up with a new account', there are input fields for 'Username' (containing 'admin'), 'Email' (containing 'fejejif479'), and 'Password' (masked with dots). Below the password field, a list of validation rules is shown: a green checkmark for 'Password must contain a lower case letter', and four red 'X' marks for 'Password must contain an upper case letter', 'Password must contain a special character', 'Password must contain a number', and 'Password must contain at least 8 characters'. At the bottom right is a 'Sign up' button, and below it is a link: 'Already have an account? Sign in'.

Рисунок 4.12 – Расширенная форма авторизации.

На эту форму поступают все ошибки обрабатываемые на backend. Здесь мы можем увидеть все ограничения, наложенные на форму.

5. Руководство программиста

Так как мы используем AWS то сперва следовало бы ознакомиться со всеми элементами, которые были использованы в курсовом проекте, а именно AWS-Vault, Serverless, Terraform, Vercel, Anuglar, Telegram API.

Далее следует создать аккаунт на AWS и локально задать набор конфигурация для AWS-Vault для корректного деплоя приложения.

Следом выкачать все файлы для деплоя по ссылке с GitHub[26].

В том случае если вы разворачиваете инфраструктуру не с нуля, то следующий шаг следует пропустить.

Потом создать terraform backend, применив команды на рисунке 5.1.

```
cd ./terraform/init-stored-state/on-dev
aws-vault exec ${your_account} -- terraform init
aws-vault exec ${your_account} -- terraform apply
```

Рисунок 5.1 – Команды terraform backend.

На следующем шаге надо развернуть ресурсы, которые управляются с помощью terraform. Для этого применяем команды на рисунке 5.2.

```
cd ./terraform/dev
aws-vault exec ${your_account} -- terraform init
aws-vault exec ${your_account} -- terraform plan
aws-vault exec ${your_account} -- terraform apply
```

Рисунок 5.2 – Команды для развертывания ресурсов.

Потом создать вручную Telegram-bot и добавить его токен в AWS KMS и начать разворачивать Serverless ресурсы в определенном порядке. Для этого вводим команды на рисунке 5.3.

```
yarn sls ${your_account} src/telegram-bot deploy -v
yarn sls ${your_account} src/telegram-notifier deploy -v
yarn sls ${your_account} src/online-crawler deploy -v
yarn sls ${your_account} src/appsync/public deploy -v
```

Рисунок 5.3 – Команды для развертывания ресурсов в определенном порядке.

Под конец надо установить webhook для бота, указав в set-telegram-webhook.ts newEndpoint и после вызвав скрипт на рисунке 5.4

```
yarn script-dev src/script/set-telegram-webhook.ts
```

Рисунок 5.4 – Вызов скрипта.

В самом конце следует вручную на Vercel настроить CI/CD tapfut-front[27].

Если вы хотите посмотреть на уже существующий проект, то следует перейти по ссылке [29] а в поля указать соответствующие данные:

- user: test-account;
- password: a_5^8v1B2O5@Uav1.

Заключение

В данной курсовой работе было создано веб-приложение «Телеграмм-бот помощник в сфере недвижимости».

Перед началом разработки был произведен аналитический обзор прототипов приложений подобной тематики и определение функциональных возможностей разрабатываемого приложения.

В процессе выполнения курсовой работы была спроектирована база данных для хранения в ней информации о шаблонах и клиентах. База данных была разработана с помощью системы управления базами данных «DynamoDB». Также был создан сервер и клиент с пользовательским интерфейсом. Тестирование программного продукта было реализовано в том числе.

В результате написания курсового проекта, было разработано приложение с использованием AWS сервисов:

- Cognito (User Pool, Identity Pool);
- DynamoDB;
- Lambda;
- AppSync;
- APIGateway;
- CloudFormation;
- System Manager (Parameter Store);
- CloudWatch (Logs, Metric, Alerts).

Был выполнен колоссальный объем работы, связанный с подключением и объединением таких систем как AWS, Telegram API и Angular API.

В соответствии с полученным результатом, можно сказать, что разработанное приложение функционирует верно, требования технического задания реализованы в полном объеме, поэтому цель курсового проекта можно считать достигнутой.

Список использованной литературы

1. «Onliner.by» [Электронный ресурс] – Режим доступа: <https://www.onliner.by/> – Дата доступа: 30.09.2020.
2. «TypeScript» [Электронный ресурс] – Режим доступа: <https://www.typescriptlang.org/docs/handbook/tsconfig-json.html> – Дата доступа: 30.09.2020.
3. «ESLint» [Электронный ресурс] – Режим доступа: <https://eslint.org/> – Дата доступа: 30.09.2020.
4. «TSLint» [Электронный ресурс] – Режим доступа: <https://palantir.github.io/tslint/> – Дата доступа: 30.09.2020.
5. «Webpack» [Электронный ресурс] – Режим доступа: <https://webpack.js.org/> – Дата доступа: 30.09.2020.
6. «Yarn» [Электронный ресурс] – Режим доступа: <https://yarnpkg.com/> – Дата доступа: 30.09.2020.
7. «Serverless» [Электронный ресурс] – Режим доступа: <https://www.serverless.com/> – Дата доступа: 30.09.2020.
8. «Terraform» [Электронный ресурс] – Режим доступа: <https://www.terraform.io/> – Дата доступа: 30.09.2020.
9. «Vercel» [Электронный ресурс] – Режим доступа: <https://vercel.com/about> – Дата доступа: 30.09.2020.
10. «Github» [Электронный ресурс] – Режим доступа: <https://github.com/> – Дата доступа: 30.09.2020.
11. «AWS Cognito» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/cognito/> – Дата доступа: 30.09.2020.
12. «AWS S3» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/s3/index.html> – Дата доступа: 30.09.2020.
13. «AWS DynamoDB» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/dynamodb/index.html> – Дата доступа: 30.09.2020.
14. «AWS Lambda» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/lambda/index.html> – Дата доступа: 30.09.2020.
15. «AWS AppSync» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/appsync/index.html> – Дата доступа: 30.09.2020.
16. «AWS VTL Resolvers» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/appsync/latest/devguide/resolver-mapping-template-reference-programming-guide.html> – Дата доступа: 30.09.2020.
17. «AWS APIGateway» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html> – Дата доступа: 30.09.2020.
18. «AWS CloudFormation» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/cloudformation/> – Дата доступа: 30.09.2020.
19. «AWS System Manager» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/systems-manager/index.html> – Дата доступа: 30.09.2020.
20. «AWS KMS» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/kms/index.html> – Дата доступа: 30.09.2020.

21. «AWS CloudWatch» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/cloudwatch/index.html> – Дата доступа: 30.09.2020.
22. «AWS Amplify» [Электронный ресурс] – Режим доступа: <https://docs.amplify.aws/> – Дата доступа: 30.09.2020.
23. «AWS SNS» [Электронный ресурс] – Режим доступа: <https://docs.aws.amazon.com/sns/index.html> – Дата доступа: 30.09.2020.
24. «Angular» [Электронный ресурс] – Режим доступа: <https://angular.io/docs> – Дата доступа: 30.09.2020.
25. «Telegam API» [Электронный ресурс] – Режим доступа: <https://core.telegram.org/> – Дата доступа: 30.09.2020.
26. «Fomin2402/taphut-back» [Электронный ресурс] – Режим доступа: <https://github.com/Fomin2402/taphut-back/> – Дата доступа: 30.09.2020.
27. «Fomin2402/taphut-front» [Электронный ресурс] – Режим доступа: <https://github.com/Fomin2402/taphut-front/> – Дата доступа: 30.09.2020.
28. «Fomin2402/taphut-back» [Электронный ресурс] – Режим доступа: <https://337219789850.signin.aws.amazon.com/console> – Дата доступа: 30.09.2020.

Приложение А

```

resource "aws_s3_bucket" "taphut-tf-backend" {
  bucket = "taphut-terraform-backend-${var.env}"
  acl    = "private"

  # Enable versioning so we can see the full revision history of our state files
  versioning {
    enabled = true
  }

  tags = {
    Where_Used = "For terraform stored states"
    Environment = "Environment: ${var.env}"
  }
}

resource "aws_dynamodb_table" "taphut-tf-state" {
  name         = "taphut-terraform-state"
  billing_mode = "PAY_PER_REQUEST"
  hash_key     = "LockID"
  attribute {
    name = "LockID"
    type = "S"
  }

  tags = {
    Where_Used = "For terraform stored states"
    Environment = "Environment: ${var.env}"
  }
}

provider "aws" {
  region = "us-east-1"
}

terraform {
  backend "s3" {
    bucket = "taphut-terraform-backend-dev"
    key    = "terraform.tfstate"
    region = "us-east-1"
    dynamodb_table = "taphut-terraform-state"
  }
}

module "tables" {
  source = "../modules/tables"
}

module "alarms" {
  source = "../modules/alarms"
}

```

```

resource "aws_dynamodb_table" "TelegramUserFilters" {
  name      = "TelegramUserFilters"
  hash_key  = "chatId"
  range_key = "filterName"

  attribute {
    name = "chatId"
    type = "S"
  }

  attribute {
    name = "filterName"
    type = "S"
  }

  billing_mode = "PAY_PER_REQUEST"

  lifecycle {
    prevent_destroy = true
  }

  point_in_time_recovery {
    enabled = true
  }
}

resource "aws_dynamodb_table" "OnlinerApartment" {
  name      = "OnlinerApartment"
  hash_key  = "id"

  attribute {
    name = "id"
    type = "N"
  }

  ttl {
    attribute_name = "expirationTime"
    enabled        = true
  }

  billing_mode = "PAY_PER_REQUEST"

  lifecycle {
    prevent_destroy = true
  }

  point_in_time_recovery {
    enabled = true
  }
}

```

Приложение Б

```

setLogLevel(process.env.DEBUG_LEVEL || 'info');
const TG_BOT_TOKEN = 'TG_BOT_TOKEN';
async function processMessage({ body }: APIGatewayEvent): Promise<IHttpResponse> {
  const token: string = mustEnv(TG_BOT_TOKEN);
  if (!body) {
    logger.error(`APIGatewayEvent body is undefined: ${JSON.stringify(body)}`);
    return httpError(400, 'chat_id is required');
  }

  const { message, edited_message } = JSON.parse(body);
  try {
    if (!message) {
      if (!edited_message) {
        return httpError(500, 'Smth went wrong');
      }
      sendToUser({
        token,
        chat_id: edited_message.chat.id as string,
        text: 'Only new messages are expected',
      });
      return httpSuccess();
    }
  } catch (error) {
    logger.error(error.message);
    return httpError(500, 'Smth went wrong');
  }
  const chat_id = message?.chat?.id?.toString();
  const { text: stringCommand } = message;

  if (!chat_id) {
    logger.error(`chat_id is required: ${JSON.stringify(chat_id)}`);
    return httpError(400, 'chat_id is required');
  }
  logger.info(`Build Parser`);

  const parser = buildParser({ chatId: chat_id, token });
  parser(stringCommand);

  return httpSuccess();
}

export const handler: APIGatewayProxyHandler = async (
  event: APIGatewayEvent,
  context: Context,
  callback: Callback
): Promise<any> => {
  logger.info(`APIGatewayEvent event: ${JSON.stringify(event)}`);
  logger.info(`APIGatewayEvent context: ${JSON.stringify(context)}`);

  const res: IHttpResponse = await processMessage(event);

  logger.info(`processMessage result: ${JSON.stringify(res)}`);

  callback(null, toLambdaHttpResponse(res));
}

```



```
};
```

Приложение В

```
const COMMAND = ['filter-list', 'fl'];
const DESCRIPTION = 'List all filters';
```

```
const ERROR_MESSAGE = 'Some error occurred during getting filters.';
```

```
function buildGetFiltersByChatId<O extends CustomExtend>(chatId: string): CustomArgvHandler<O> {
  return async (argv: CustomArgv<O>) => {
    try {
      const { Items } = await getFilters(chatId);
      const filters = Items.map((f: IFilterRow) => f.filter);
      const msg = filtersToString(filters);
      argv.respond(msg);
    } catch (error) {
      argv.respond(error?.message || ERROR_MESSAGE);
    }
  };
}
```

```
export function adddGetFiltersCommand<T>({ chatId, argv }: CommandBuilderType<T>): Argv<T> {
  return argv.command(COMMAND, DESCRIPTION, {}, buildGetFiltersByChatId(chatId));
}
```

```
const COMMAND = ['filter-get', 'fg'];
const DESCRIPTION = 'Get filter by name';
const EXAMPLE = 'filter-get -n filterName';
```

```
const ERROR_MESSAGE = 'Some error occurred during getting the filter.';
```

```
function buildGetFilterById<O extends CustomExtend>(chatId: string): CustomArgvHandler<O> {
  return async (argv: CustomArgv<O>) => {
    const filterName = argv.name as string;
    try {
      const { filter } = await getFilterById(chatId, filterName);
      argv.respond(filterToString(filter));
    } catch (error) {
      if (error instanceof ItemNotFoundError) {
        argv.respond(`${filterName} filter doesn't exist!`);
      } else {
        argv.respond(error?.message || ERROR_MESSAGE);
      }
    }
  };
}
```

```
const defineCommandParameter: any = <T>(argv: Argv<T>) => {
  return argv
    .options('n', {
      alias: 'name',
      demandOption: true,
      describe: 'Filter name',
      type: 'string',
    })
    .example(EXAMPLE, DESCRIPTION);
};
```

```

export function adddGetFilterByIdCommand<T>({ chatId, argv }: CommandBuilderType<T>): Argv<T>
{
  return      argv.command(COMMAND,      DESCRIPTION,      defineCommandParameter,
buildGetFilterById(chatId));
}
const COMMAND = ['filter-create', 'fc'];
const DESCRIPTION = 'Create filter';
const EXAMPLE = 'fc -n filterName -c City --cr USD --min 100 --max 350 -r 2';

const ERROR_MESSAGE = 'Some error occurred during creating the filter.';

function buildCreateFilter<O extends CustomExtend>(chatId: string): CustomArgvHandler<O> {
  return async (argv: CustomArgv<O>) => {
    try {
      const { name, city, currency, min, max, rooms } = argv;

      const filter: IFilter = {
        filterName: name as string,
        city: city as string | undefined,
        currency: currency as Currency,
        minPrice: min as number | undefined,
        maxPrice: max as number | undefined,
        roomsNumber: rooms as number | undefined,
      };

      const createdFilter = await createFilter(chatId, filter);

      const msg = filterToString(
        createdFilter,
        `Filter ${filter.filterName} is created successfully.\n`
      );

      argv.respond(msg);
    } catch (error) {
      const errMsg = error?.message || ERROR_MESSAGE;
      argv.respond(errMsg);
    }
  };
}

const defineCommandParameter: any = <T>(argv: Argv<T>) => {
  return argv
    .option('n', {
      alias: 'name',
      demandOption: true,
      describe: 'filter name',
      type: 'string',
    })
    .option('c', {
      alias: 'city',
      demandOption: false,
      describe: 'city',
      type: 'string',
    })
    .option('cr', {
      alias: 'currency',
      demandOption: false,
      describe: 'currency',
    })

```

```

    type: 'string',
    default: 'USD',
    choices: ['USD', 'BYN'],
  })
  .option('min', {
    demandOption: false,
    describe: 'min price',
    type: 'number',
  })
  .option('max', {
    demandOption: false,
    describe: 'max price',
    type: 'number',
  })
  .option('r', {
    alias: 'rooms',
    demandOption: false,
    describe: 'number of rooms',
    type: 'number',
  })
  .example(EXAMPLE, "");
};

export function adddCreateFilterCommand<T>({ chatId, argv }: CommandBuilderType<T>): Argv<T> {
  return argv.command(COMMAND, DESCRIPTION, defineCommandParameter, buildCreateFil
const COMMAND = ['filter-update', 'fu'];
const DESCRIPTION = 'Update filter';
const EXAMPLE = 'fu -n filterName -c City --cr USD --min 100 --max 350 -r 2';

const ERROR_MESSAGE = 'Some error occurred during deleting the filter.';

function buildUpdateFilterById<O extends CustomExtend>(chatId: string): CustomArgvHandler<O> {
  return async (argv: CustomArgv<O>) => {
    const { name, city, currency, min, max, rooms } = argv;
    try {
      const updatedFilter: IFilter = {
        filterName: name as string,
        city: city as string | undefined,
        currency: currency as Currency,
        minPrice: min as number | undefined,
        maxPrice: max as number | undefined,
        roomsNumber: rooms as number | undefined,
      };

      const { filter } = await updateFilterById(chatId, updatedFilter);

      const msg = filterToString(filter, `Filter "${name as string}" is updated successfully.\n`);

      argv.respond(msg);
    } catch (error) {
      if (error instanceof ItemNotFoundError) {
        argv.respond(`"${name}" filter doesn't exist!`);
      } else {
        argv.respond(error?.message || ERROR_MESSAGE);
      }
    }
  };
};
}

```

```

const defineCommandParameter: any = <T>(argv: Argv<T>) => {
  return argv
    .options('n', {
      alias: 'name',
      demandOption: true,
      describe: 'filter name',
      type: 'string',
    })
    .options('c', {
      alias: 'city',
      demandOption: false,
      describe: 'city',
      type: 'string',
    })
    .options('cr', {
      alias: 'currency',
      demandOption: false,
      describe: 'currency',
      type: 'string',
      default: 'USD',
      choices: ['USD', 'BYN'],
    })
    .options('min', {
      demandOption: false,
      describe: 'min price',
      type: 'number',
    })
    .options('max', {
      demandOption: false,
      describe: 'max price',
      type: 'number',
    })
    .options('r', {
      alias: 'rooms',
      demandOption: false,
      describe: 'number of rooms',
      type: 'number',
    })
    .example(EXAMPLE, "");
};

export function addUpdateFilterByIdCommand<T>({ chatId, argv }: CommandBuilderType<T>):
Argv<T> {
  return argv.command(COMMAND, DESCRIPTION, defineCommandParameter,
buildUpdateFilterById(chatId));
}
const COMMAND = ['filter-delete', 'fd'];
const DESCRIPTION = 'Delete filter';
const EXAMPLE = 'filter-delete -n filterName';

const ERROR_MESSAGE = 'Some error occurred during deleting the filter.';

function buildDeleteFilterById<O extends CustomExtend>(chatId: string): CustomArgvHandler<O> {
  return async (argv: CustomArgv<O>) => {
    const filterName = argv.name as string;

    try {

```

```

const { filter } = await deleteFilterById(chatId, filterName);

const msg = filterToString(filter, `Filter "${filterName}" is deleted successfully.\n`);

argv.respond(msg);
} catch (error) {
  if (error instanceof ItemNotFoundError) {
    argv.respond(`"${filterName}" filter doesn't exist!`);
  } else {
    argv.respond(error?.message || ERROR_MESSAGE);
  }
}
};
}

const defineCommandParameter: any = <T>(argv: Argv<T>) => {
  return argv
    .options('n', {
      alias: 'name',
      demandOption: true,
      describe: 'Filter name',
      type: 'string',
    })
    .example(EXAMPLE, DESCRIPTION);
};

export function addDeleteFilterByIdCommand<T>({ chatId, argv }: CommandBuilderType<T>):
Argv<T> {
  return argv.command(COMMAND, DESCRIPTION, defineCommandParameter,
    buildDeleteFilterById(chatId));
}

```

Приложение Г

```

import { APIGatewayProxyHandler, APIGatewayEvent, Callback, Context } from 'aws-lambda';

import { IOnlinerApartment, IOnlinerApartmentRow } from 'onliner-crawler/model';
import { onlinerApartmentToString } from 'utils/converters/apartment';
import logger, { setDebugLevel } from 'utils/logger';
import { eachLimit, retryNTimes } from 'utils/async';
import { IFilter, IFilterRow } from 'utils/filter';
import { sendToUser } from 'utils/telegram';
import { mustEnv } from 'utils/env';
import {
  apartmentIsSuitedForFilter,
  getAllFilters,
  getAllNewApartments,
  updateApartmentsToOLD,
} from './telegram-notifier-helper';

setDebugLevel(process.env.DEBUG_LEVEL || 'info');

const TG_BOT_TOKEN = 'TG_BOT_TOKEN';

type Notification = { chat_id: string; apartment: IOnlinerApartment };

/**
 * 1. Scan All of NEW Apartments
 * 2. Scan All Filters
 * 3. For Apartments check filter
 * 3.1 Algorithm for form notification:
 *   - form filterMapObj { [key: chatId]: arrayOfChatFilters[] }
 *   - iterate through all points
 *   - iterate through chatIds in filterMapObj
 *     - iterate through all filters in chatIds
 *     - if apartment suite for filter, push { chatId, apartment } to notification and go to next chatId
 * 4. Parallel process notifications and execute sendToUser(filter.chatID,
convertApartmentToMsg(apartment))
 * 5. Update NEW Apartment to OLD Apartments
 */
export const handler: APIGatewayProxyHandler = async (
  _event: APIGatewayEvent,
  _context: Context,
  callback: Callback
): Promise<any> => {
  logger.info(`Telegram-notifier start`);
  const token: string = mustEnv(TG_BOT_TOKEN);
  const newApartments = await getAllNewApartments();
  logger.info(`newApartments count: ${newApartments.length}`);

  const filters = await getAllFilters();
  logger.info(`filters count: ${filters.length}`);

  const notifications: Notification[] = [];
  const filtersByChatId: { [chatId: string]: IFilter[] } = {};

  filters.forEach(({ chatId, filter }: IFilterRow) => {
    if (Array.isArray(filtersByChatId[chatId])) {
      filtersByChatId[chatId].push(filter);
    }
  });

```

```

    } else {
      filtersByChatId[chatId] = [filter];
    }
  });

newApartments.forEach(({ apartment }: IOnlinerApartmentRow) => {
  Object.keys(filtersByChatId).forEach((chat_id: string) => {
    const chatFilters: IFilter[] = filtersByChatId[chat_id];

    chatFilters.some((filter: IFilter) => {
      if (apartmentIsSuitedForFilter(apartment, filter)) {
        notifications.push({ chat_id, apartment });

        logger.info(
          `Apartment ${apartment.id} is suited for filter: (${chat_id} ${filter.filterName})`
        );
        logger.info(`Skip checking current chatId filters, go to next chatId filters`);
        return true;
      }

      return false;
    });
  });
});

logger.info(`Total notifications count: ${notifications.length}`);

if (notifications.length) {
  await eachLimit(
    notifications,
    async ({ chat_id, apartment }: Notification) =>
      await retryNTimes({
        func: async () =>
          await sendToUser({ token, chat_id, text: onlinerApartmentToString(apartment) }),
        timeout: 10000,
        maxTries: 1,
      }),
    10,
    10000
  );

  logger.info(`Notifications are send`);
}

await updateApartmentsToOLD(newApartments);

logger.info(`Apartments status is updated to OLD`);
logger.info(`Telegram-notifier complete`);
callback(null, 'success');
};

```

Приложение Д

```

import { Injectable } from '@angular/core';
import { Auth } from 'aws-amplify';
import { BehaviorSubject, Observable, ReplaySubject } from 'rxjs';
import { fromPromise } from 'rxjs/internal-compatibility';
import { ISignUpResult, CognitoUser } from 'amazon-cognito-identity-js';
import { ICredentials } from 'aws-amplify/lib/Common/types/types';
import { Router } from '@angular/router';
import { tap } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class AmplifyService {

  public isAuthenticatedSubj: ReplaySubject<boolean>;

  get isAuthenticated(): Promise<boolean> {
    return Auth.currentAuthenticatedUser().then(() => {
      this.isAuthenticatedSubj.next(true);
      return true;
    })
    .catch(() => {
      this.isAuthenticatedSubj.next(false);
      return false;
    });
  }

  constructor(private router: Router) {
    this.isAuthenticatedSubj = new ReplaySubject<boolean>(1)
  }

  signIn(email: string, password: string): Observable<CognitoUser> {
    return fromPromise(
      Auth.signIn({
        username: email.toLocaleLowerCase(),
        password,
      }));
  }

  signOut(): Observable<any> {
    return fromPromise(Auth.signOut());
  }

  signUp(email: string, password: string): Observable<ISignUpResult> {
    return fromPromise(
      Auth.signUp({
        username: email.toLocaleLowerCase(),
        password,
        attributes: {
          email: email.toLocaleLowerCase(),

```



```

    }
  }));
}

confirmSignUp(email: string, code: string): Observable<any> {
  return fromPromise(Auth.confirmSignUp(email.toLocaleLowerCase(), code));
}

changePassword(user: CognitoUser, oldPassword: string, newPassword: string):
Observable<'SUCCESS'> {
  return fromPromise(Auth.changePassword(user, oldPassword, newPassword));
}

forgotPassword(email: string): Observable<any> {
  return fromPromise(Auth.forgotPassword(email.toLocaleLowerCase()));
}

forgotPasswordSubmit(email: string, code: string, password: string): Observable<any> {
  return fromPromise(Auth.forgotPasswordSubmit(email.toLocaleLowerCase(), code, password));
}

currentAuthenticatedUser(): Observable<CognitoUser> {
  return fromPromise(Auth.currentAuthenticatedUser());
}

currentUserCredentials(): Observable<ICredentials> {
  return fromPromise(Auth.currentUserCredentials());
}

logout(): Observable<ICredentials> {
  return fromPromise(Auth.signOut()).pipe(
    tap(() => localStorage.clear()),
    tap(() => this.isAuthenticatedSubj.next(false)),
    tap(() => this.router.navigate(['/']));
  );
}
}

```

Приложение Е

```

import { Injectable } from '@angular/core';
import { ApolloQueryResult } from '@apollo/client';
import { Apollo, gql } from 'apollo-angular';
import { Observable, of } from 'rxjs';
import { map, tap } from 'rxjs/operators';
import { IPagination, IProduct } from '../utils/models';

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  constructor(private apollo: Apollo) {

  }

  // TODO add
  getProductPage(limit: number, token?: string): Observable<IPagination<IProduct[]>> {
    return this.query(limit, token)
      .pipe(
        map((data: any) => {
          console.log(data.data.onlinerApartments);
          return data.data.onlinerApartments;
        }),
        map(({ nextToken, scannedCount, items }: ResData) => ({
          token: nextToken,
          limit: scannedCount,
          data: items.map((item: OnlinerApartmentRow) => this.convertToProduct(item))
        }))
      )
  }

  convertToProduct(item: OnlinerApartmentRow): IProduct {
    return {
      id: item.id.toString(),
      title: item.apartment.rent_type.split('_').concat(' ').toString(),
      description: item.apartment.location.address,
      price: item.apartment.price.amount,
      location: item.apartment.location.address,
      sourceLink: item.apartment.url,
      imageLink: item.apartment.photo
    };
  }

  query(limit: number = 10, token?: string): Observable<ApolloQueryResult<IOnlinerPaginationRes>>
  {
    if (token) {
      return this.queryWithParams(limit, token);
    } else {
      return this.queryWithoutParams()
    }
  }

```

```

    }

    private queryWithParams(limit: number = 10, token: string):
Observable<ApolloQueryResult<IOnlnerPagationRes>> {
    return this.apollo.query({
        query: gql`
            query($limit: Int!, $token: String){
                onlinerApartments(limit: $limit, nextToken: $token) {
                    items {
                        id
                        status
                        apartment {
                            id
                            price {
                                amount
                                currency
                                converted {
                                    key {
                                        amount
                                        currency
                                    }
                                }
                            }
                        }
                        rent_type
                        location {
                            address
                            user_address
                        }
                        photo
                        created_at
                        last_time_up
                        up_available_in
                        url
                    }
                    createdAt
                    expirationTime
                    updatedAt
                }
                nextToken
                scannedCount
            }
        `,
        variables: {
            limit,
            token
        },
    });
}

private queryWithoutParams(): Observable<ApolloQueryResult<IOnlnerPagationRes>> {
    return this.apollo.query({
        query: gql`

```

```

query onlinerApartments {
  onlinerApartments {
    items {
      id
      status
      apartment {
        id
        price {
          amount
          currency
          converted {
            key {
              amount
              currency
            }
          }
        }
      }
      rent_type
      location {
        address
        user_address
      }
      photo
      created_at
      last_time_up
      up_available_in
      url
    }
    createdAt
    expirationTime
    updatedAt
  }
  nextToken
  scannedCount
}

});
}
}

export type IOnlinerPaginationRes = {
  onlinerApartments: ResData;
}

export type ResData = {
  nextToken: string | undefined; // if undefined, then it is the last page
  scannedCount: number;          // same as limit, but in the answer
  items: OnlinerApartmentRow[];
};

export type OnlinerApartmentRow = {
  id: number;

```

```

status: ApartmentStatus;
apartment: OnlinerApartment;
createdAt: string;
updatedAt?: string;
expirationTime: number;
};

export type ApartmentStatus = 'NEW' | 'IN_FLIGHT' | 'ERROR' | 'OLD';

export type OnlinerApartment = {
  id: number;
  price: {
    amount: string;
    currency: OnlinerCurreneces;
    converted: {
      [key: string]: { // key ca be: USD and BYN
        amount: string;
        currency: OnlinerCurreneces;
      };
    };
  };
  rent_type: OnlinerRentType;
  location: IOnlinerApartmentLocation;
  photo: string;
  contact: {
    owner: boolean;
  };
  created_at: string;
  last_time_up: string;
  up_available_in: number;
  url: string;
};

export type OnlinerCurreneces = 'USD' | 'BYN';

export type OnlinerRentType = '1_rooms' | '2_rooms' | '3_rooms' | '4_rooms' | '5_rooms';

export interface IOnlinerApartmentLocation {
  address: string;
  user_address: string;
};

```

Приложение Ж

```

@Component({
  selector: 'app-telegram',
  template: `
    <div #script style.display="none">
      <ng-content></ng-content>
    </div>`,
  styleUrls: ['./telegram.component.scss']
})
export class TelegramComponent implements AfterViewInit {

  @ViewChild('script',{ static: true })
  script!: ElementRef;

  public constructor(private ngZone: NgZone) {
    (window as any)['loginViaTelegram'] = (loginData: unknown) => this.loginViaTelegram(loginData);
  }

  convertToScript() {
    const element = this.script.nativeElement;
    const script = document.createElement('script');
    script.src = 'https://telegram.org/js/telegram-widget.js?5';
    script.setAttribute('data-telegram-login', environment.telegramBotName);
    script.setAttribute('data-size', 'large');
    // Callback function in global scope
    script.setAttribute('data-onauth', 'loginViaTelegram(user)');
    script.setAttribute('data-request-access', 'write');
    element.parentElement.replaceChild(script, element);
  }

  ngAfterViewInit() {
    this.convertToScript();
  }

  private loginViaTelegram(loginData: unknown) {
    console.log(loginData);
    // If the login should trigger view changes, run it within the NgZone.
    // this.ngZone.run(() => console.log(loginRequest));
  }
}

```

Приложение 3

```

display_serverless_directories() {
    echo ""
    echo "Available serverless directories:"
    find . -path ./node_modules -prune -o -name serverless.yml -print | sed 's/\./serverless.yml//;s/.\\/'
}
SYNTAX_STRING='yarn sls [dev|prod] serverless_dir ...serverless_args'
EXAMPLE_STRING='yarn sls dev src/ticker logs -f refreshTicker'
if [ "$1" = "--help" ]; then
    echo "Help from the serverless command (pass this in place of ...serverless_args):"
    serverless --help
    display_serverless_directories
    echo ""
    echo "Command syntax: $SYNTAX_STRING"
    echo "Example: $EXAMPLE_STRING"
    exit
fi
if [ "$#" -lt "3" ]; then
    echo "Bad args. Syntax: $SYNTAX_STRING"
    echo "Example: $EXAMPLE_STRING"
    display_serverless_directories
    echo ""
    echo "yarn sls --help for more details on serverless commands"
    exit
fi
export stage=$1
export directory=$2
shift 2
branch=`git symbolic-ref -q --short HEAD`
if [ "$stage" = "prod" ]; then
    if echo $* | grep -q deploy; then
        if [ "$branch" != "master" ]; then
            echo "current branch $branch != master. Prod deploy disallowed"
            exit
        fi
        git remote update
        local=$(git rev-parse master)
        remote=$(git rev-parse origin/master)
        if [ $local != $remote ]; then
            echo "your local checkout must be up to date before prod deployment!"
            exit
        fi
        echo "Good to go for prod deploy. On master, fully up to date with remote..."
    fi
fi
if [ "$stage" = "forceprod" ]; then
    echo "!!!!Forcing prod deployment!!!!"
    stage="prod"
fi
export $(cat config/$stage.env | xargs)

cd $directory
profile=$stage

export $(aws-vault exec $profile --no-session -- env | grep AWS)
serverless --profile $profile --stage $stage "$@" --color | grep -v "Load command"

```