Министерство образования Республики Беларусь

Учреждение образования

БелорусскиЙ государственный университет

информатики и радиоэлектроники

Факультет Компьютерных систем и сетей

Кафедра Информатики

|  |
| --- |
| *К защите допустить*: |
| Заведующий кафедрой Информатики |
| \_\_\_\_\_\_\_\_\_\_\_\_ Н. А. Волорова |

Пояснительная записка

к дипломному проекту

на тему

**Чат, учитывающий геолокационные данные**

**пользователей**

БГУИР ДП 1-40 01 03 00 060 ПЗ

|  |  |  |
| --- | --- | --- |
| Студент |  | Е. В. Тарненок |
| Руководитель |  | А. П. Шкор |
| Консультанты: |  |  |
| *от кафедры Информатики* |  | В. Я. Анисимов |
| *по экономической части* |  | В. И. Шкода |
|  |  |  |
| Нормоконтролер |  | В. В. Шиманский |
| Рецензент |  |  |

Минск 2016

**РЕФЕРАТ**

Дипломный проект предоставлен следующим образом. Чертежный материал: 6 листов формата А1. Пояснительная записка: 77 страниц, 12 рисунков, 2 таблицы, 25 литературных источников, 12 формул.

Ключевые слова: CQRS, EVENT SOURCING, MESSAGE BROKER, ЧАТ, ANDROID

Объектом исследования и разработки является построение легко масштабируемых программных систем

Целью проекта является разработка программного средства для организации общения между пользователями, учитывающего геолокационные данные участников.

При разработке программного средства использовалась среда разработки Visual studio 2015, Android Studio 2.0

Областью практического применения программного средства являются мобильные устройства. Для использования программы не необходимо никаких практических и теоретических навыков и знаний в информатике. Поэтому программным продуктом может пользоваться любой желающий, у которого есть мобильное устройство с установленной операционной системой Android 4.0 и выше

Разработанный программный продукт можно считать экономически эффективным, и он полностью оправдывает вложенные в него средства.

Дипломный проект является завершенным, поставленная задача решена в полной мере, присутствует возможность дальнейшего развития программы и увеличение ее функционала.

Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет: ФКСиС. Кафедра: Информатики.

Специальность: 1-400103

Специализация: нет.

УТВЕРЖДАЮ

Зав. кафедрой

\_\_\_\_\_\_\_\_\_\_\_ И. О. Фамилия

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2016 г.

ЗАДАНИЕ

по дипломному проекту студента

Тарненок Евгения Валерьевич

**1.** Тема проекта: «Чат, учитывающий геолокационные данные пользователей» – утверждена приказом по университету от 25.03.2016 г. № 525-с.

**2.** Срок сдачи студентом законченного проекта: 1 июня 2016 г.

**3.** Исходные данные к проекту:

**3.1** Операционная система: Android.

**3.2** Среда разработки: Android Studio 2.0, Visual Studio 2015

**3.3** База данных: Mongo DB.

**3.4** Брокер сообщений: RabbitMQ

**4.** Содержание пояснительной записки (перечень подлежащих разработке вопросов):

Введение. 1. Обзор предметной области. 2. Используемые технологии 3. Системное проектирование. 4. Функциональное проектирование. 5. Методика испытаний. 6. Руководство пользователя. 7. Технико-экономическое обоснование дипломного проекта. Заключение. Список используемых источников.

**5.** Перечень графического материала:

**5.1** Вводный плакат (ПЛ), формат А1, лист 1

**5.2** Схема работы CQRS (ПЛ), формат А1, лист 1

**5.3** Диаграмма вариантов использования (ПЛ), формат А1, лист 1

**5.4** Диаграмма классов (ПД), формат А1, лист 1

**5.5** Структурная схема (ПД), формат А1, лист 1

**5.6** Диаграмма вариантов использования (ПД), формат А1, лист 1

**6.** Содержание задания по экономической части: «Технико-экономическое обоснование эффективности разработки чата, учитывающего геолокационные данные пользователей».

ЗАДАНИЕ ВЫДАЛА В. И. Шкода

КАЛЕНДАРНЫЙ ПЛАН

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапов  дипломного проекта | Объем  этапа, % | Срок выполнения этапа | Примечания |
| Подбор и изучение литературы | 15 | 27.03 – 02.04 |  |
| Структурное проектирование | 15 | 03.04 – 09.04 |  |
| Функциональное проектирование | 25 | 10.04 – 23.04 |  |
| Программа и методика испытаний | 10 | 01.05 – 07.05 |  |
| Расчет экономической эффективности | 10 | 08.05 – 16.05 |  |
| Завершение оформления пояснительной записки | 25 | 17.05 – 28.05 |  |

Дата выдачи задания: 1 февраля 2016 г. \_\_\_\_\_\_\_\_\_\_Руководитель А. П. Шкор

ЗАДАНИЕ ПРИНЯЛ К ИСПОЛНЕНИЮ \_\_\_\_\_\_\_\_\_\_ Е. В. Тарненок

СОДЕРЖАНИЕ

[ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ 7](#_Toc452668544)

[ВВЕДЕНИЕ 8](#_Toc452668545)

[1 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ 10](#_Toc452668546)

[1.1 Архитектура REST-сервисов 10](#_Toc452668547)

[1.2 CQRS 12](#_Toc452668548)

[1.3 EventSourcing 13](#_Toc452668549)

[1.4 NoSQL 14](#_Toc452668550)

[2 ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ 20](#_Toc452668551)

[2.1 Kotlin 20](#_Toc452668552)

[2.2 Программная платформа Microsoft .NET 22](#_Toc452668553)

[2.3 Язык программирования C# 24](#_Toc452668554)

[2.4ОС Android 28](#_Toc452668555)

[3 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ 34](#_Toc452668556)

[4 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 37](#_Toc452668557)

[4.1 Блок работы с данными по протоколу HTTP 37](#_Toc452668558)

[4.2 Блок аутентификации и регистрации пользователей 39](#_Toc452668559)

[4.3 Блок реализации бизнес-логики приложения 40](#_Toc452668560)

[4.4 Блок работы с базой данных 44](#_Toc452668561)

[4.5 Блок уведомления пользователей о происходящих событиях 45](#_Toc452668562)

[4.6 Блок работы с api – сервером 50](#_Toc452668563)

[4.7 Блок определяющий местоположение пользователей 52](#_Toc452668564)

[5 МЕТОДИКА ИСПЫТАНИЙ 55](#_Toc452668565)

[5.1 Модульное тестирование 55](#_Toc452668566)

[6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 60](#_Toc452668567)

[6. 1 Установка программного средства 60](#_Toc452668568)

[6.2 Использование программного продукта 60](#_Toc452668569)

[7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ЭФФЕКТИВНОСТИ РАЗРАБОТКИ И РЕАЛИЗАЦИИ ЧАТА, УЧИТЫВАЮЩЕГО ГЕОЛАКАЦИОННЫЕ ДАННЫЕ ПОЛЬЗОВАТЕЛЕЙ 65](#_Toc452668570)

[7.1 Характеристика чата, учитывающего геолокационные данные пользователей 65](#_Toc452668571)

[7.2 Расчет затрат на разработку программного продукта 65](#_Toc452668572)

[7.3 Расчет экономического эффекта от продажи программного продукта 69](#_Toc452668573)

[7.4 Расчет показателей эффективности разработки программного продукта 71](#_Toc452668574)

[ЗАКЛЮЧЕНИЕ 74](#_Toc452668575)

[СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 75](#_Toc452668576)

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ

* XML (eXtensible Markup Language) - расширяемый [язык разметки](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%80%D0%B0%D0%B7%D0%BC%D0%B5%D1%82%D0%BA%D0%B8). Является базовым стандартом для языка описания ресурсов, RDF, позволяющего упростить многие проблемы в Web, связанные с поиском нужной информации, обеспечением контроля за содержимым сетевых ресурсов, создания электронных библиотек и т.д.
* JSON (JavaScript Object Notation) - [текстовый формат](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BA%D1%81%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82) [обмена данными](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%BC%D0%B8), основанный на [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) и обычно используемый именно с этим языком. Как и многие другие текстовые форматы, JSON легко читается людьми
* CQRS (command-query responsibility segregation) - это принцип [императивного программирования](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), изобретённый [Бертраном Мейером](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B9%D0%B5%D1%80,_%D0%91%D0%B5%D1%80%D1%82%D1%80%D0%B0%D0%BD) во время работы над языком программирования [Eiffel](https://ru.wikipedia.org/wiki/Eiffel). Принцип гласит, что метод должен быть либо командой, выполняющей какое-то действие, либо запросом, возвращающим данные, но не одновременно. Другими словами, задавание вопроса не должно менять ответ. Более формально, возвращать значение можно только [чистым](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), не имеющим [побочных эффектов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B1%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%8D%D1%84%D1%84%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) методам. Следует отметить, что неукоснительное соблюдение этого принципа делает невозможным отслеживание количества вызовов запросов.
* REST (Representational State Transfer) -  [архитектурный стиль](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) взаимодействия компонентов распределённого приложения в [сети](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C). REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой [гипермедиа](https://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%BF%D0%B5%D1%80%D0%BC%D0%B5%D0%B4%D0%B8%D0%B0)-системы.
* SQL (structured query language) - [формальный](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) непроцедурный язык программирования, применяемый для создания, модификации и управления данными в произвольной [реляционной базе данных](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), управляемой соответствующей системой управления базами данных ([СУБД](https://ru.wikipedia.org/wiki/%D0%A1%D0%A3%D0%91%D0%94)).
* NoSQL (not only SQL) -  термин, обозначающий ряд подходов, направленных на реализацию хранилищ [баз данных](https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), имеющих существенные отличия от моделей, используемых в традиционных [реляционных СУБД](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%A1%D0%A3%D0%91%D0%94) с доступом к данным средствами языка [SQL](https://ru.wikipedia.org/wiki/SQL).

ВВЕДЕНИЕ

В настоящее время человечество стало сильно зависеть от компьютерных сетей, в основном от глобальной сети Internet. Не составляет труда найти человека, отправить ему письмо или мгновенное сообщение, услышать и увидеть его. Сейчас тяжело представить жизнь человечества без электронных платежей, интернет–магазинов, виртуальных–библиотек социальных сетей и облачных хранилищ данных.

Вам не составит труда скачать приложение для обмена сообщениями и послать собеседнику на другом конце света таинственное и чарующие послание. Вы даже можете позвонить и увидеть своего собеседника перед собой, хотя он находится за тысячи километров от вас. И все это позволяет осуществить бесподобная компьютерная сеть.

Однако, что, если вам нужно отослать сообщение человеку, которого у вас нет в контактах, но вам что-что необходимо от него. Например, сколько раз вы бывали в такой ситуации, когда приходите в какое-нибудь кафе или шикарнейший ресторан и ждете немалое количество времени, когда к вам подойдет официант и примет заказ. А что вам нужно? Нужно чтобы официант всего лишь принял ваш заказ и отдал его на кухню. И чтобы сэкономить ваше время, я решил разработать чат, учитывающий геолокационные данные пользователей.

Примерный принцип работы данного чата выглядит следующим образом (на примере кафе): владелец или администратор кафе создает чат, местом действия которого будет является кафе, далее даются специальные права официантам в чате (чтобы обыкновенные пользователи могли с легкостью определить их). И теперь пользователи заходят в кафе, могут общаться в этом чате и назначать себе свободного официанта. Ведь так же превосходно, когда вам не надо искать и просить официанта, чтобы он просто вам принес счет.

Основное достоинство таких чатов, что вам не нужно их искать в глобальной паутине или знать его название. Вы сразу же его находите в зависимости от контекста, то есть местоположения где вы находите.

Вот еще один пример из жизни. Приходите вы уставший домой, собираетесь себе приготовить поужинать и тут как на зло закончилась всемиообажаемая соль. Решение есть, вы можете сходить и отдолжить соли у красивой соседки, но в друг у нее есть только сахар? Поэтому вы можете создать чат, действие которого будет распространяться всего лишь на жильцов вашего подъезда и задать необходимый вопрос.

На данный момент в интернете существует огромное количество приложений, позволяющих обмениваться мгновенными сообщениями между собой. Но основной недостаток их в том, что вам нужно знать название определенного чата, или быть приглашенным одним из участников.

Таким образом, с учетом популярности и доступности смартфонов и планшетов, актуальной является задача разработки приложения для мобильной платформы, позволяющего обмениваться сообщениями между пользователями в зависимости от их местоположения.

Для достижения поставленной цели необходимо решить следующие задачи дипломного проектирования:

* выполнить исследование предметной области;
* выполнить системное проектирование;
* выполнить функциональное проектирование;
* разработать программу и методику испытаний системы;
* разработать руководство пользователя;
* выполнить технико-экономическое обоснование проекта.

Объектом исследования является принцип построения программных продуктов с использованием CQRS. Предметом исследования является чат, учитывающий геолокационные данные пользователей, реализованный в виде мобильного приложения

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ
   1. Архитектура REST-сервисов

REST (Representational state transfer) – стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения веб-служб.

Термин REST был введен в 2000 году Роем Филдингом, одним из авторов HTTP-протокола. Системы, поддерживающие REST, называются RESTful-системами.

В общем случае REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

Отсутствие дополнительных внутренних прослоек означает передачу данных в том же виде, что и сами данные.

Каждая единица информации однозначно определяется URL – это значит, что URL по сути является первичным ключом для единицы данных. Отсюда и получается строго заданный формат. Причем совершенно не имеет значения, в каком формате находятся данные по указанному адресу – это может быть и HTML, и отсканированная копия в виде jpeg-файла, и документ Microsoft Word.

Архитектура REST предполагает следующие условия.

Клиент-серверная архитектура – единый интерфейс между клиентом и сервером. Такое разделение подразумевает отсутствие связи между клиентами и хранилищем данных. Это хранилище остается внутренним устройством сервера, таким образом переносимость клиентского кода увеличивается, что способствует упрощению сервера и его масштабируемости. Серверы и клиенты могут быть мгновенно заменены независимо друг от друга, так как интерфейс между ними не меняется.

Отсутствие состояния – серверы не связаны с интерфейсами клиентов и их состояниями. На стороне сервера не сохраняется пользовательский контекст между двумя разными запросами. Каждый запрос содержит всю информацию, необходимую обработчику, а состояние сессии хранится на клиенте. Состояние сессии может быть передано сервером на другой сервис благодаря поддержке постоянного состояния базой данных. Клиент отсылает запросы, когда готов совершить транзакцию на изменение состояния.

Кэширование – как и в WWW, каждый из клиентов, а также промежуточные узлы между сервером и клиентами могут кэшировать ответы сервера. В каждом запросе клиента должно явно содержаться указание о возможности кэширования ответа и получения ответа из существующего кэша. В свою очередь, ответы могут явно или неявно определяться как кэшируемые или некэшируемые для предотвращения повторного использования клиентами в последующих запросах сохраненной информации. Правильное использование кэширования в REST-архитектуре устраняет избыточные клиент-серверные взаимодействия, что улучшает скорость и расширяемость системы.

Единообразие интерфейса – ограничения на унифицированный интерфейс являются фундаментальными в дизайне REST-сервисов. Каждый из сервисов функционирует и развивается независимо. Выделяются следующие ограничения для унификации интерфейса:

* идентификация ресурсов. Индивидуальные ресурсы идентифицированы в запросах, например, с использованием URI в интернет-системах. Ресурсы сами по себе отделены от представлений, которые возвращаются клиентам. Например, сервер может отсылать данные из базы данных в виде HTML, XML или JSON, ни один из которых не является типом хранения внутри хранилища сервера;
* манипуляция ресурсами через представление. В момент, когда клиенты хранят представление ресурса, включая метаданные, они имеют достаточно данных для модификации или удаления ресурса;
* самодостаточные сообщения. Каждое сообщение достаточно информативно для того, чтобы описать каким образом его обрабатывать. К примеру, какой парсер необходимо применить для извлечения данных из сообщения согласно Internet медиа-типу;
* гипермедиа, как средство изменения состояния сервера. Клиенты могут изменить состояние системы только через действия, которые динамически идентифицируются на сервере посредством гипермедиа (к примеру, гиперссылки в гипертексте, формы связи, флажки, радиокнопки и прочее). До того момента, пока сервер в явном виде не сообщит обратное, клиенты могут полагаться на то, что любое из предоставленных действий доступно для выполнения на сервере.

Слои – клиент может взаимодействовать не напрямую с сервером, а через промежуточные узлы (слои). При этом клиент может не знать об их существовании, за исключением случаев передачи конфиденциальной информации. Промежуточные серверы выполняют балансировку нагрузки и могут использовать дополнительное кэширование.

Архитектура REST имеет следующие преимущества:

* надежность (за счет отсутствия необходимости сохранять информацию о состоянии клиента, которая может быть утеряна);
* производительность (за счет использования кэша);
* масштабируемость;
* прозрачность системы взаимодействия (особенно необходимая для приложений обслуживания сети);
* простота интерфейсов;
* портативность компонентов;
* легкость внесения изменений.
  1. CQRS

В компании, где я проходил преддипломную практику, на большинстве проектов используются данные технологии CQRS, Event Sourcing, и MongoDB в качестве базы данных. Из этого всего я слышал только о MongoDB. Попытавшись вникнуть в CQRS, я не сразу понял все тонкости данного подхода, но почему-то мне понравилась идея разделения модели взаимодействия с данными на две — read и write. Возможно потому что она как-то перекликалась с парадигмой программирования “разделение обязанностей”, возможно потому - что была очень в духе DDD

Вообще многие говорят о CQRS как о паттерне проектирования. На мой взгляд он слишком сильно влияет на общую архитектуру приложения, чтобы называться просто “паттерном проектирования”, поэтому я предпочитаю называть его принципом или подходом. Использование CQRS проникает почти во все уголки приложения.

Сразу хочу уточнить что я работал только со связкой CQRS + Event Sourcing, и никогда не пробовал просто CQRS, так как мне кажется, что без Event Sourcing он теряет очень много преимуществ. Я здесь упомяну только несколько фреймворков для .NET. Наиболее популярные это [NCQRS](https://github.com/ncqrs/ncqrs), [Lokad CQRS](https://github.com/Lokad/lokad-cqrs),[SimpleCQRS](https://github.com/tyronegroves/SimpleCQRS). Так же можете обратить внимание на [Event Store](https://github.com/joliver/EventStore) Джонатана Оливера с поддержкой огромного количества различных баз данных.

CQRS расшифровывается как Command Query Responsibility Segregation (разделение ответственности на команды и запросы). Это паттерн проектирования, о котором я впервые услышал от Грега Янга (Greg Young). В его основе лежит простое понятие, что вы можете использовать разные модели для обновления и чтения информации. Однако это простое понятие ведет к серьёзным последствиям в проектировании информационных систем.  Не сказать, что исчерпывающее определение, но сейчас я попробую объяснить, что именно Фаулер имел в виду.

К настоящему времени сложилась такая ситуация что практические все работают с моделью данных как с CRUD хранилищем. CQRS предлагает альтернативный подход, но затрагивает не только модель данных. Если вы используете CQRS, то это сильно отражается на архитектуре вашего приложения.

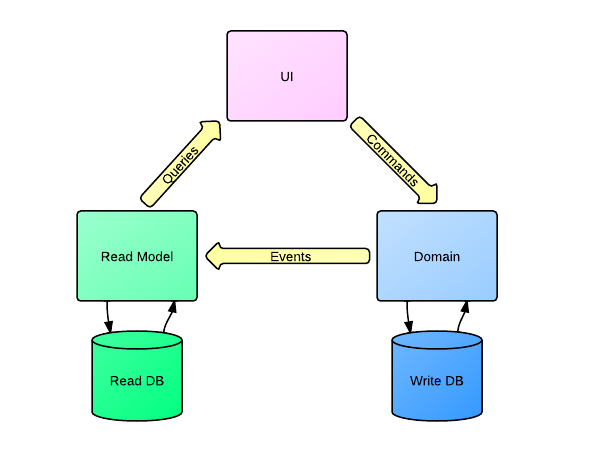
Вот как я изобразил схему работы CQRS на рисунке 1.1. Первое что бросается в глаза это то что у вас уже две модели данных, одна для чтения (Queries), одна для записи (Commands). И обычно это значит, что у вас еще и две базы данных. И так как мы используем CQRS + Event Sourcing, то write-база (write-модель) — это Event Store, что-то вроде лога всех действий пользователя (на самом деле не всех, а только тех, которые важны с точки зрения бизнес-модели и влияют на построение read-базы). А read-база — это в общем случае денормализировнное хранилище тех данных, которые вам нужны для отображения пользователю. Почему я сказал, что read-база денормализированная?  


Рисунок 1.1 – Схема работы CQRS

Вы конечно можете использовать любую структуру данных в качестве read-модели, но я считаю, что при использовании CQRS + Event Sourcing не стоит сильно заморачиваться над нормализвацией read-базы, так как она может быть полностью перестроена в любое время. И это большой плюс, особенно если вы не хотите использовать реляционные базы данных и смотрите в сторону NoSQL.

Write-база вообще представляет собой одну коллекцию ивентов. То есть тут тоже нету смысла использовать реляционную базу.

* 1. EventSourcing

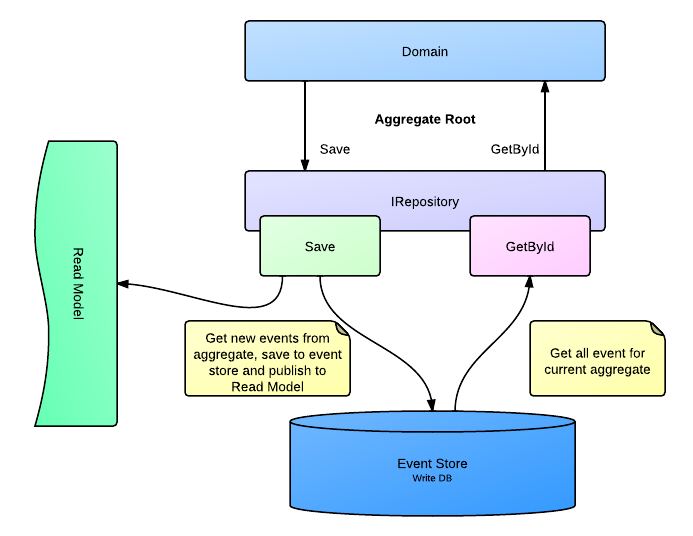
Идея Event Sourcing в том, чтобы записывать каждое событие, которое меняет состояние приложения в базу данных. Таким образом получается, что мы храним не состояние наших сущностей, а все события, которые к ним относятся. Однако мы привыкли к тому чтобы манипулировать именно состоянием, оно храниться у нас в базе, и мы всегда можем его посмотреть.  
В случае с Event Sourcing мы тоже оперируем с состоянием сущности. Но в отличии от обычной модели мы это состоянием не храним, а воспроизводим каждый раз при обращении (см. рисунок 1.2).

Рисунок 1.2 – Работа домена приложения

* 1. NoSQL

*Характеристики NoSQL баз данных*

Общих характеристик для всех NoSQL немного, так как под лэйблом NoSQL сейчас скрывается множество разнородных систем (самый полный, пожалуй, список можно найти на сайте <http://nosql-database.org/>). Многие характеристики свойственны только определенным NoSQL базам, это я обязательно упомяну при перечислении.

***Не используется SQL .***

Имеется в виду ANSI SQL DML, так как многие базы пытаются использовать query languages похожие на общеизвестный любимый синтаксис, но полностью его реализовать не удалось никому и вряд ли удастся. Хотя по слухам есть стартапы, которые пытаются реализовать SQL, например, в хадупе (<http://www.drawntoscalehq.com/> и <http://www.hadapt.com/> )

***Неструктурированные (schemaless)***

Смысл таков, что в NoSQL базах в отличие от реляционных структура данных не регламентирована (или слабо типизированна, если проводить аналогии с языками прогаммирования) — в отдельной строке или документе можно добавить произвольное поле без предварительного декларативного изменения структуры всей таблицы. Таким образом, если появляется необходимость поменять модель данных, то единственное достаточное действие — отразить изменение в коде приложения.

Если мы меняем логику приложения, значит мы ожидаем новое поле также и при чтении. Но в силу отсутствия схемы данных поле totalSum отсутствует у других уже существующих объектов Order. В этой ситуации есть два варианта дальнейших действий. Первый — обойти все документы и обновить это поле во всех существующих документах. В силу объемов данных этот процесс происходит без каких-либо блокировок (сравним с командой alter table rename column), поэтому во время обновления уже существующие данные могут считываться другими процессами.

Приятное следствие отсутствия схемы — эффективность работы с разреженными (sparse) данными. Если в одном документе есть поле date\_published, а во втором — нет, значит никакого пустого поля date\_published для второго создано не будет. Это, в принципе, логично, но менее очевидный пример — column-family NoSQL базы данных, в которых используются знакомые понятия таблиц/колонок. Однако в силу отсутствия схемы, колонки не объявляются декларативно и могут меняться/добавляться во время пользовательской сессии работы с базой. Это позволяет в частности использовать динамические колонки для реализации списков.

У неструктурированной схемы есть свои недостатки — помимо упомянутых выше накладных расходов в коде приложения при смене модели данных — отсутствие всевозможных ограничений со стороны базы (not null, unique, check constraint и т.д.), плюс возникают дополнительные сложности в понимании и контроле структуры данных при параллельной работе с базой разных проектов (отсутствуют какие-либо словари на стороне базы). Впрочем, в условиях быстро меняющегося современного мира такая гибкость является все-таки преимуществом. В качестве примера можно привести Твиттер, который лет пять назад вместе с твиттом хранил лишь немного дополнительной информации (время, Twitter handle и еще несколько байтов метаинформации), однако сейчас в дополнение к самому сообщению в базе сохраняется еще несколько килобайт метаданных(Здесь и далее речь идет в-основном о key-value, document и column-family базах данных, graph базы данных могут не обладать этими свойствами).

***Представление данных в виде агрегатов (aggregates).***

В отличие от реляционной модели, которая сохраняет логическую бизнес-сущность приложения в различные физические таблицы в целях нормализации, NoSQL хранилища оперируют с этими сущностями как с целостными объектами (см. рисунок 1.3):

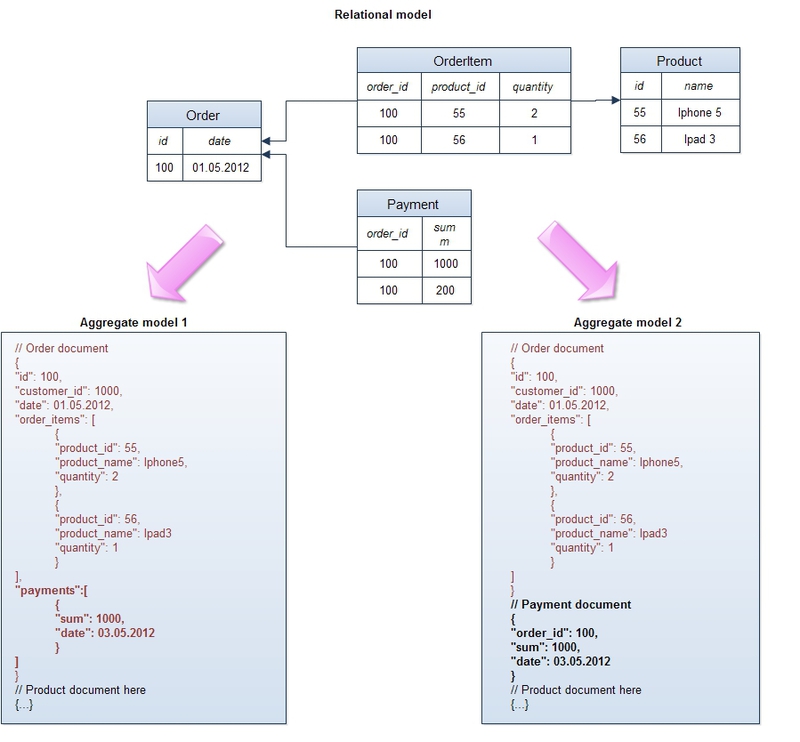


Рисунок 1.3 – Сравнение представление данных в базах данных

В этом примере продемонстрированы агрегаты для стандартной концептуальной реляционной модели e-commerce “заказ — позиции заказа — платежи — продукт”. В обоих случаях заказ объединяется с позициями в один логический объект, при этом каждая позиция хранит в себе ссылку на продукт и некоторые его атрибуты, например, название (такая денормализация необходима, чтобы не запрашивать объект продукта при извлечении заказа — главное правило распределенных систем — минимум “джоинов” между объектами). В одном агрегате платежи объединены с заказом и являются составной частью объекта, в другом — вынесены в отдельный объект. Этим демонстрируется главное правило проектирования структуры данных в NoSQL базах — она должна подчиняться требованиям приложения и быть максимально оптимизированной под наиболее частые запросы. Если платежи регулярно извлекаются вместе с заказом — имеет смысл их включать в общий объект, если же многие запросы работают только с платежами — значит, лучше их вынести в отдельную сущность.

Многие возразят, заметив, что работа с большими, часто денормализованными, объектами чревата многочисленными проблемами при попытках произвольных запросов к данным, когда запросы не укладываются в структуру агрегатов. Что, если мы используем заказы вместе с позициями и платежами по заказу (так работает приложение), но бизнес просит нас посчитать, сколько единиц определенного продукта было проданно в прошлом месяце? В этом случае вместо сканирования таблицы OrderItem (в случае реляционной модели) нам придется извлекать заказы целиком в NoSQL хранилище, хотя большая часть этой информации нам будет не нужна. К сожалению, это компромисс, на который приходится идти в распределенной системе: мы не можем проводить нормализацию данных как в обычной односерверной системе, так как это создаст необходимость объединения данных с разных узлов и может привести к значительному замедлению работы базы   
Плюсы и минусы обоих подходов я попытался сгруппировать в табличке, представленной на рисунке 1.4:

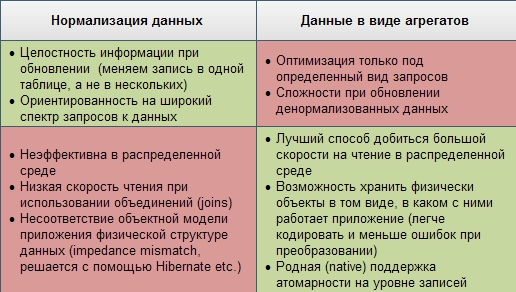


Рисунок 1.4 – Сравнение баз данных

***Слабые ACID свойства.***

Долгое время консистентность (consistency) данных была “священной коровой” для архитекторов и разработчиков. Все реляционные базы обеспечивали тот или иной уровень изоляции — либо за счет блокировок при изменении и блокирующего чтения, либо за счет undo-логов. С приходом огромных массивов информации и распределенных систем стало ясно, что обеспечить для них транзакционность набора операций с одной стороны и получить высокую доступность и быстрое время отклика с другой — невозможно. Более того, даже обновление одной записи не гарантирует, что любой другой пользователь моментально увидит изменения в системе, ведь изменение может произойти, например, в мастер-ноде, а реплика асинхронно скопируется на слейв-ноду, с которой и работает другой пользователь. В таком случае он увидит результат через какой-то промежуток времени. Это называется eventual consistency и это то, на что идут сейчас все крупнейшие интернет-компании мира, включая Facebook и Amazon. Последние с гордостью заявляют, что максимальный интервал, в течение которого пользователь может видеть неконсистентные данные составляют не более секунды.

Логичный вопрос, который появляется в такой ситуации — а что делать системам, которые классически предъявляют высокие требования к атомарности-консистентности операций и в то же время нуждаются в быстрых распределенных кластерах — финансовым, интернет-магазинам и т.д? Практика показывает, что эти требования уже давно неактуальны: вот что сказал один разработчик финансовой банковской системы: “Если бы мы действительно ждали завершения каждой транзакции в мировой сети ATM (банкоматов), транзакции занимали бы столько времени, что клиенты убегали бы прочь в ярости. Что происходит, если ты и твой партнер снимаете деньги одновременно и превышаете лимит? — Вы оба получите деньги, а мы поправим это позже.” Другой пример — бронирование гостиниц, показанный на картинке. Онлайн-магазины, чья политика работы с данными предполагает eventual consistency, обязаны предусмотреть меры на случай таких ситуаций (автоматическое решение конфликтов, откат операции, обновление с другими данными). На практике гостиницы всегда стараются держать “пул” свободных номеров на непредвиденный случай и это может стать решением спорной ситуации.

На самом деле слабые ACID свойства не означают, что их нет вообще. В большинстве случаев приложение, работающее с реляционной базой данных, использует транзакцию для изменения логически связанных объектов (заказ — позиции заказа), что необходимо, так как это разные таблицы. При правильном проектировании модели данных в NoSQL базе (агрегат представляет из себя заказ вместе с перечнем пунктов заказа) можно добиться такого же самого уровня изоляции при изменении одной записи, что и в реляционной базе данных.

***Распределенные системы, без совместно используемых ресурсов***

Опять же, это не касается граф баз данных, чья структура по определению плохо разносится по удаленным нодам.

Это, возможно, главный лейтмотив развития NoSQL баз. С лавинообразным ростом информации в мире и необходимости ее обрабатывать за разумное время встала проблема вертикальной масштабируемости — рост скорости процессора остановился на 3.5 Ггц, скорость чтения с диска также растет тихими темпами, плюс цена мощного сервера всегда больше суммарной цены нескольких простых серверов. В этой ситуации обычные реляционные базы, даже кластеризованные на массиве дисков, не способны решить проблему скорости, масштабируемости и пропускной способности. Единственный выход из ситуации — горизонтальное масштабирование, когда несколько независимых серверов соединяются быстрой сетью и каждый владеет/обрабатывает только часть данных и/или только часть запросов на чтение-обновление. В такой архитектуре для повышения мощности хранилища (емкости, времени отклика, пропускной способности) необходимо лишь добавить новый сервер в кластер — и все. Процедурами шардинга, репликации, обеспечением отказоустойчивости (результат будет получен даже если одна или несколько серверов перестали отвечать), перераспределения данных в случае добавления ноды занимается сама NoSQL база. Вкратце представлю основные свойства распределенных NoSQL баз:

Репликация — копирование данных на другие узлы при обновлении. Позволяет как добиться большей масштабируемости, так и повысить доступность и сохранность данных. Принято подразделять на два вида:   
master-slave и peer-to-peer.

Первый тип предполагает хорошую масштабируемость на чтение (может происходить с любого узла), но немасштабируемую запись (только в мастер узел). Также есть тонкости с обеспечением постоянной доступности (в случае падения мастера либо вручную, либо автоматически на его место назначается один из оставшихся узлов). Для второго типа репликации предполагается, что все узлы равны и могут обслуживать как запросы на чтение, так и на запись.

Шардинг часто использовался как “костыль” к реляционным базам данных в целях увеличения скорости и пропускной способности: пользовательское приложение партицировало данные по нескольким независимым базам данных и при запросе соответствующих данных пользователем обращалось к конкретной базе. В NoSQL базах данных шардинг, как и репликация, производятся автоматически самой базой и пользовательское приложение обособленно от этих сложных механизмов.

1. ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

2.1 Kotlin

Не так давно компания [JetBrains](http://jetbrains.com/) (версия 1.0 вышла в первом квартале 2016 года), занимающаяся созданием сред разработки, анонсировала свой новый продукт — язык программирования Kotlin. На компанию обрушилась волна критики: критикующие предлагали компании одуматься и доделать плагин для Scala, вместо того, чтобы разрабатывать свой язык. Разработчикам на Scala действительно очень не хватает хорошей среды разработки, но и проблемы разработчиков плагина можно понять: Scala, которая появилась на свет благодаря исследователям из Швейцарии, вобрала в себя многие инновационные научные концепции и подходы, что сделало создание хорошего инструмента для разработки крайне непростой задачей. На данный момент сегмент современных языков со статической типизацией для JVM невелик, поэтому решение о создании своего языка вместе со средой разработки к нему выглядит очень дальновидным. Даже если этот язык совсем не приживется в сообществе — JetBrains в первую очередь делает его для своих нужд. Эти нужды может понять любой java-программист: Java, как язык, развивается очень медленно, новые возможности в языке не появляются (функции первого порядка мы ждем уже не первый год), совместимость со старыми версиями языка делает невозможным появление многих полезных вещей и в ближайшем будущем (например, приличной параметризации типов). Для компании, разрабатывающей ПО язык программирования — основной рабочий инструмент, поэтому эффективность и простота языка — это показатели, от которых зависит не только простота разработки инструментов для него, но и затраты программиста на кодирование, т. е. насколько просто будет этот код сопровождать и разбираться в нем.

Рассмотрим несколько отличительных особенностей языка, которые выделяют его из многообразия существующих языков. Язык статически типизирован. Но по сравнению с java, компилятор Kotlin добавляет в тип информацию о возможности ссылки содержать null, что ужесточает проверку типов и делает выполнение более безопасным:

fun foo(text:String) {

println(text.toLowerCase()) // NPE? Нет!

}

val str:String? = null // String? -- тип допускающий null-ы

foo(str) // <- компилятор не пропустит такой вызов --

// тип str должен быть String, чтобы

// передать его в foo

Несмотря на то, что такой подход может избавить программиста от ряда проблем связанных с NPE, для java-программиста поначалу это кажется излишним — приходится делать лишние проверки или преобразования. Но через некоторое время программирования на kotlin, возвращаясь на java или .net, чувствуешь, что тебе не хватает этой информации о типе. С этим связаны и вопросы обратной совместимости с java — этой информации в байткоде java нет, но насколько мне известно, этот вопрос еще в процессе решения, а пока все приходящие из java типы — nullable.

В языке есть поддержка first-class functions. Это значит, что функция — это встроенный в язык тип для которого есть специальный синтаксис. Функции можно создавать по месту, передавать в параметры другим функциям, хранить на них ссылки:

fun doSomething(thing:()->Unit) { // объявляем параметр типа функция

// ()->Unit ничего не принимает и

// ничего важного не возвращает

thing() // вызываем

}

doSomething() { // а здесь на лету создаем функцию типа

// ()->Unit и передаем её в функцию doShomething

// если функция -- последний параметр, можно

// вынести её за скобки вызова

println("Hello world")

}

Если добавить к этому extension-функции, позволяющие расширить уже существующий класс методом не нарушающим инкапсуляцию класса, но к которым можно обращаться как к методам этого класса, то мы получим довольно мощный механизм расширения достаточно бедных в плане удобств стандартных библиотек java. По традиции, добавим уже существующую в стандартной библиотеке возможность фильтрации списка:

fun <T> List<T>.filter(condition:(T)->Boolean):List<T> {

val result = list<T>()

for(item in this) {

if(condition(item))

result.add(item)

}

return result

}

val someList = list(1, 2, 3, 4).filter { it > 2 } // someList==[3, 4]

Обратите внимание на то, что у переменных не указаны типы — компилятор Kotlin выводит их, если это возможно и не мешает понятности интерфейса. Вообще, язык сделан таким образом, чтобы максимально избавить человека за клавиатурой от набирания лишних знаков: короткий, но понятный синтаксис с минимум ключевых слов, отсутствие необходимости точек с запятой для разделения выражений, вывод типов, где это уместно, отсутствие ключевого слова new для создания класса — только необходимое.

Но на мой взгляд одно из самых важных преимуществ языка Kotlin это его полная совместимость с java. То есть вы с легкостью можете в одном проекте использовать как java, так и Kotlin код (хотя это не особо и приветствуется, так как возникает путаница).

2.2 Программная платформа Microsoft .NET

Программная платформа Microsoft .NET является одной из реализаций стандарта ECMA-335 и является современным инструментом создания клиентских и серверных приложений для операционной системы Windows. Первая общедоступная версия .NET Framework вышла в феврале 2002 года. С тех пор платформа активно эволюционировала и на данный момент было выпущено шесть версии данного продукта. На данный момент номер последней версии .NET Framework—4.5. Платформа Microsoft .NET была призвана решить некоторые наболевшие проблемы, скопившиеся на момент её выхода, в средствах разработки приложений под Windows. Ниже перечислены некоторые из них:

* + - сложность создания надежных приложений;
    - сложность развертывания и управления версиями приложений и библиотек;
    - сложность создания, переносимого ПО;
    - отсутствие единой целевой платформы для создателей компиляторов;
    - проблемы с безопасным исполнением непроверенного кода;
    - великое множество различных технологий и языков программирования, которые не совместимы между собой.

Многие из этих проблем были решены. Далее более подробно рассматривается внутреннее устройство Microsoft .NET. Основными составляющими компонентами Microsoft .NET являются общая языковая исполняющая среда (Common Language Runtime) и стандартная библиотека классов (Framework Class Library). CLR представляет из себя виртуальную машину и набор сервисов обслуживающих исполнение программ, написанных для Microsoft .NET. Ниже приводится перечень задач, возлагаемых на CLR:

* + - загрузка и исполнение управляемого кода;
    - управление памятью при размещении объектов;
    - изоляция памяти приложений;
    - проверка безопасности кода;
    - преобразование промежуточного языка в машинный код;
    - доступ к расширенной информации от типах—метаданным;
    - обработка исключений, включая межъязыковые исключения;
    - взаимодействие между управляемым и неуправляемым кодом (в том числе и COM-объектами);
    - поддержка сервисов для разработки (профилирование, отладка и т.д.).

Программы, написанные для Microsoft .NET представляют из себя набор типов взаимодействующих между собой. Microsoft .NET имеет общую систему типов (Common Type System, CTS). Данная спецификация описывает определения и поведение типов, создаваемых для Microsoft .NET [16]. В частности, в данной спецификации описаны возможные члены типов, механизмы сокрытия реализации, правила наследования, типы-значения и ссылочные типы, особенности параметрического полиморфизма и другие возможности, предоставляемые CLI. Общая языковая спецификация (Common Language Specification, CLS)—подмножество общей системы типов. Это набор конструкций и ограничений, которые являются руководством для создателей библиотек и компиляторов в среде .NET Framework. Библиотеки, построенные в соответствии с CLS, могут быть использованы из любого языка программирования, поддерживающего CLS. Языки, соответствующие CLS (к их числу относятся языки C#, Visual Basic .NET, Visual C++/CLI), могут интегрироваться друг с другом. CLS—это основа межъязыкового взаимодействия в рамках платформы Microsoft .NET.

Некоторые из возможностей, предоставляемых Microsoft .NET: верификация кода, расширенная информация о типах во время исполнения, сборка мусора, безопасность типов, —невозможны без наличия подробных метаданных о типах из которых состоит исполняемая программа. Подробные метаданные о типах генерируются компиляторами и сохраняются в результирующих сборках. Сборка—это логическая группировка одного или нескольких управляемых модулей или файлов ресурсов, является минимальной единицей с точки зрения повторного использования, безопасности и управлениями версиями.

Одной из особенностей Microsoft .NET, обеспечивающей переносимость программ без необходимости повторной компиляции, является представление исполняемого кода приложений на общем промежуточном языке (Common Intermediate Language, CIL). Промежуточный язык является бестиповым, стековым, объекто-ориентированным ассемблером. Данный язык очень удобен в качестве целевого языка для создателей компиляторов и средств автоматической проверки кода для платформы Microsoft .NET, также язык довольно удобен для чтения людьми. Наличие промежуточного языка и необходимость создания производительных программ подразумевают наличие преобразования промежуточного кода в машинный код во время исполнения программы. Одним из компонентов общей языковой исполняющей среды, выполняющим данное преобразование, является компилятор времени исполнения (Just-in-time compiler) транслирующий промежуточный язык в машинные инструкции, специфические для архитектуры компьютера на котором исполняется программа.

Ручное управление памятью всегда являлось очень кропотливой и подверженной ошибкам работой. Ошибки в управлении памятью являются одними из наиболее сложных в устранении типами программных ошибок, также эти ошибки обычно приводят к непредсказуемому поведению программы, поэтому в Microsoft .NET управление памятью происходит автоматически. Автоматическое управление памятью является механизмом поддержания иллюзии бесконечности памяти. Когда объект данных перестает быть нужным, занятая под него память автоматически освобождается и используется для построения новых объектов данных. Имеются различные методы реализации такого автоматического распределения памяти. В Microsoft .NET для автоматического управления памятью используется механизм сборки мусора (garbage collection). Существуют различные алгоритмы сборки мусора со своими достоинствами и недостатками. В Microsoft .NET используется алгоритм пометок (mark and sweep) в сочетании с различными оптимизациями, такими как, например, разбиение всех объектов по поколениям и использование различных куч для больших и малых объектов.

Ниже перечислены, без приведения подробностей, некоторые важные функции исполняемые общей языковой исполняющей средой:

* + - обеспечение многопоточного исполнения программы;
    - поддержание модели памяти, принятой в CLR;
    - поддержка двоичной сериализации;
    - управление вводом и выводом;
    - структурная обработка исключений;
    - возможность размещения исполняющей среды внутри других процессов.

Как уже упоминалось выше, большую ценностью для Microsoft .NET представляет библиотека стандартных классов—соответствующая CLS-спецификации объектно-ориентированная библиотека классов, интерфейсов и системы типов (типов-значений), которые включаются в состав платформы Microsoft .NET. Эта библиотека обеспечивает доступ к функциональным возможностям системы и предназначена служить основой при разработке .NET-приложений, компонент, элементов управления.

2.3 Язык программирования C#

C#—объектно-ориентированный, типо-безопасный язык программирования общего назначения. Язык создавался с целью повысить продуктивность программистов. Для достижения этой цели в языке гармонично сочетаются простота, выразительность и производительность промежуточного кода, получаемого после компиляции. Главным архитектором и идеологом языка с первой версии является Андрес Хейлсберг (создатель Turbo Pascal и архитектор Delphi). Язык C# является платформенно нейтральным, но создавался для хорошей работы с Microsoft .NET. Этот язык сочетает простой синтаксис, похожий на синтаксис языков C++ и Java, и полную поддержку всех современных объектно-ориентированных концепций и подходов. В качестве ориентира при разработке языка было выбрано безопасное программирование, нацеленное на создание надежного и простого в сопровождении кода.

Язык имеет богатую поддержку парадигмы объекто-ориентированного программирования, включающую поддержку инкапсуляции, наследования и полиморфизма. Отличительными чертами C# с точки зрения ОО парадигмы являются:

* Унифицированная система типов. В C# сущность, содержащая данные и методы их обработки, называется типом. В C# все типы, являются ли они пользовательскими типами, или примитивами, такими как число, производны от одного базового класса.
* Классы и интерфейсы. В классической объекто-ориентированной парадигме существуют только классы. В C# дополнительно существуют и другие типы, например, интерфейсы. Интерфейс—это сущность, напоминающая классы, но содержащая только определения членов. Конкретная реализация указанных членов интерфейса происходит в типах, реализующих данный интерфейс. В частности, интерфейсы могут быть использованы при необходимости проведения множественного наследования (в отличие от языков C++ и Eiffel, C# не поддерживает множественное наследование классов).
* Свойства, методы и события. В чистой объекто-ориентированной парадигме все функции являются методами. В C# методы являются лишь одной из возможных разновидностей членов типа, в C# типы также могут содержать свойства, события и другие члены. Свойство—это такая разновидность функций, которая инкапсулирует часть состояния объекта. Событие—это разновидность функций, которые реагируют на изменение состояния объекта.

В большинстве случаев C# обеспечивает безопасность типов в том смысле, что компилятор контролирует чтобы взаимодействие с экземпляром типа происходило согласно контракту, который он определяют. Например, компилятор C# не скомпилирует код, который обращается со строками, как если бы они были целыми числами. Говоря более точно, C# поддерживает статическую типизацию, в том смысле что большинство ошибок типов обнаруживаются на стадии компиляции. За соблюдение более строгих правил безопасности типов следит исполняющая среда. Статическая типизация позволяет избавиться от широкого круга ошибок, возникающих из-за ошибок типов. Она делает написание и изменение программ более предсказуемыми и надежными, кроме того, статическая типизация позволяет существовать таким средствам как автоматическое дополнение кода и его предсказуемый статический анализ. Еще одним аспектом типизации в C# является её строгость. Строгая типизация означает, что правила типизации в языке очень «сильные». Например, язык не позволяет совершать вызов метода, принимающего целые числа, передавая в него вещественное число. Такие требования спасают от некоторых ошибок.

C# полагается на автоматическое управление памятью со стороны исполняющей среды, предоставляя совсем немного средств для управления жизненным циклом объектов. Несмотря на это, в языке все же присутствует поддержка работы с указателями. Данная возможность предусмотрена для случаев, когда критически важна производительность приложения или необходимо обеспечить взаимодействие с неуправляемым кодом.

Как уже упоминалось C# не является платформенно зависимым языком. Благодаря усилиям компании Xamarin возможно писать программы на языке C# не только для операционных систем Microsoft, но и ряда других ОС. Существуют инструменты создания приложений на C# для серверных и мобильных платформ, например, iOS, Android, Linux и других.

Создатели языка C# не являются противниками привнесения в язык новых идей и возможностей, в отличии от создателей одного из конкурирующих языков. Каждая новая версия компилятора языка привносит различные полезные возможности, которые отчаются требованиям индустрии. Далее приводится краткий обзор развития языка.

Первая версия C# была похожа по своим возможностям на Java 1.4, несколько их расширяя: так, в C# имелись свойства (выглядящие в коде как поля объекта, но на деле вызывающие при обращении к ним методы класса), индексаторы (подобные свойствам, но принимающие параметр как индекс массива), события, делегаты, циклы foreach, структуры, передаваемые по значению, автоматическое преобразование встроенных типов в объекты при необходимости (boxing), атрибуты, встроенные средства взаимодействия с неуправляемым кодом (DLL, COM) и прочее.

Версия Microsoft .NET 2.0 привнесла много новых возможностей в сравнении с предыдущей версией, что отразилось и на языках под эту платформу. Проект спецификации C# 2.0 впервые был опубликован Microsoft в октябре 2003 года; в 2004 году выходили бета-версии (проект с кодовым названием Whidbey), C# 2.0 окончательно вышел 7 ноября 2005 года вместе с Visual Studio 2005 и Microsoft .NET 2.0. Ниже перечислены новые возможности в версии 2.0

– Частичные типы (разделение реализации класса более чем на один файл).

– Обобщённые, или параметризованные типы (generics). В отличие от шаблонов C++, они поддерживают некоторые дополнительные возможности и работают на уровне виртуальной машины. Вместе с тем, параметрами обобщённого типа не могут быть выражения, они не могут быть полностью или частично специализированы, не поддерживают шаблонных параметров по умолчанию, от шаблонного параметра нельзя наследоваться.

– Новая форма итератора, позволяющая создавать сопрограммы с помощью ключевого слова yield, подобно Python и Ruby.

– Анонимные методы, обеспечивающие функциональность замыканий.

– Оператор ??: return obj1 ?? obj2; означает (в нотации C# 1.0) return obj1!=null ? obj1 : obj2;.

– Обнуляемые (nullable) типы-значения (обозначаемые вопросительным знаком, например, int? i = null;), представляющие собой те же самые типы-значения, способные принимать также значение null. Такие типы позволяют улучшить взаимодействие с базами данных через язык SQL.

– Поддержка 64-разрядных вычислений позволяет увеличить адресное пространство и использовать 64-разрядные примитивные типы данных [20]. Третья версия языка имела одно большое нововведение

—Language Integrated Query (LINQ), для реализации которого в языке дополнительно появилось множество дополнительных возможностей. Ниже приведены некоторые из них:

– Ключевые слова select, from, where, позволяющие делать запросы из SQL, XML, коллекций и т.п.

– Инициализацию объекта вместе с его свойствами:

Customer c = new Customer(); c.Name = "James"; c.Age=30;

можно записать как

Customer c = new Customer { Name = "James", Age = 30 };

– Лямбда-выражения:

listOfFoo.Where(delegate(Foo x) { return x.size > 10; });

теперь можно записать как

listOfFoo.Where(x => x.size > 10);

– Деревья выражений—лямбда-выражения теперь могут быть представлены в виде структуры данных, доступной для обхода во время выполнения, тем самым позволяя транслировать строго типизированные C# выражения в другие домены (например, выражения SQL).

– Вывод типов локальной переменной:

var x = "hello";

вместо string

x = "hello";

– Безымянные типы:

var x = new { Name = "James"};

– Методы-расширения—добавление метода в существующий класс с помощью ключевого слова this при первом параметре статической функции.

– Автоматические свойства: компилятор сгенерирует закрытое поле и соответствующие аксессор и мутатор для кода вида

public string Name { get; private set; }

C# 3.0 совместим с C# 2.0 по генерируемому MSIL-коду; улучшения в языке —чисто синтаксические и реализуются на этапе компиляции. Visual Basic .NET 10.0 и C# 4.0 были выпущены в апреле 2010 года, одновременно с выпуском Visual Studio 2010. Новые возможности в версии 4.0:

– Возможность использования позднего связывания.

– Именованные и опциональные параметры.

– Новые возможности COM interop.

– Ковариантность и контрвариантность интерфейсов и делегатов.

– Контракты в коде (Code Contracts).

В C# 5.0 было немного нововведений, но они носят большую практическую ценность. В новой версии появилась упрощенная поддержка выполнения асинхронных функций с помощью двух новых слов— async и await. Ключевым словом async помечаются методы и лямбда-выражения, которые внутри содержат ожидание выполнения асинхронных операций с помощью оператора await, который отвечает за преобразования кода метода во время компиляции.

* 1. ОС Android

Платформа Android основана на идее вычислений общего назначения с помощью карманных устройств. Эта комплексная платформа предлагает стек Linux-подобной операционной системы для управления устройствами, памятью и процессами. Библиотеки Java-кода Android предназначены для работы с телефонией, видео, голосом, графикой, подключениями, программированием пользовательского интерфейса и рядом других аспектов таких устройств.

Хотя платформа Android создана для работы мобильных и планшетных устройств, она обладает всеми характеристиками полнофункциональной среды настольных устройств. Компания Google сделала эту среду доступной для программирования на Java с помощью пакета для разработки ПО ( Software Development Kit — SDK), который называется Android SDK. При работе с Android SDK вы редко будете чувствовать, что пишете программу для мобильного устройства, т.к. в вашем распоряжении находятся почти все библиотеки классов, используемые на настольном компьютере или сервере — в том числе и реляционная СУБД.

Android SDK поддерживает платформу Java Platform, Standard Edition (Java SE) почти в полном объеме, за исключением средств Abstract Window Toolkit (AWT) и Swing. Вместо них в Android SDK имеется собственная обширная инфраструктура для построения современных пользовательских интерфейсов. Поскольку программирование приложений осуществляется на языке Java, можно было ожидать, что понадобится виртуальная машина Java Virtual Machine (JVM), которая отвечает за интерпретацию байт-кода Java во время выполнения. JVM обычно обеспечивает необходимую оптимизацию, что помогает Java достичь уровней производительности, сравнимых с компилируемыми языками вроде C и C++. ОС Android содержит собственную оптимизированную машину JVM для выполнения скомпилированных файлов Java-классов, которая учитывает ограничения карманных устройств на память, скорость работы процессора и электропитание. Эта виртуальная машина называется Dalvik VM, и мы подробно ознакомимся с ней в разделе “Знакомство с Dalvik VM”.

В мобильных телефонах используются различные операционные системы: Symbian OS, Microsoft Windows Phone OS, Mobile Linux, iPhone OS (на основе Mac OS X), Moblin (от компании Intel) и многие другие патентованные ОС. К настоящему времени ни одна из них не стала общепринятым стандартом. Доступные API-интерфейсы и среды для разработки мобильных приложений обладают слишком многими ограничениями и проигрывают в сравнении с инфраструктурами настольных компьютеров. В отличие от них, платформа Android предлагает открытость, экономичность, доступный исходный код и, что более важно, компактную высокоуровневую и согласованную среду разработки. Компания Google приобрела на тот момент только образовавшуюся компанию Android Inc. в 2005 г. и тем самым начала разработку платформы Android (рис. 1.2). Основные действующие лица в Android Inc. — Энди Рубин (Andy Rubin), Рич Майнер (Rich Miner), Ник Сирс (Nick Sears) и Крис Уайт (Chris White). Android SDK был выпущен в ноябре 2007 г. сначала как “пробный вариант”. В сентябре 2008 г. компания T-Mobile объявила о выпуске T-Mobile G1 — первого смартфона на платформе Android. С тех пор мы могли наблюдать выпуск SDK версий 2.0, 3.0, а теперь и 4.0 — грубо говоря, ежегодно по одному SDK. Появление устройств, работающих под управлением Android, поначалу было похоже на слабый ручеек, но теперь оно сродни бурному потоку. Одной из ключевых архитектурных целей при разработке Android является возможность взаимодействия приложений друг с другом и совместное использование их компонентов. Такое многократное использование применимо не только к службам, но также к данным и пользовательскому интерфейсу.

ОС Android дала импульс движению разработчиков и впоследствии поддерживала его с помощью созданных самостоятельно компонентов для использования модели облачных вычислений, предлагаемых веб-ресурсами, и локального хранения данных на самом карманном устройстве. Быстрому признанию Android способствовала и поддержка реляционной СУБД в устройстве (см рисунок 2.1).

В выпусках 1.0 и 1.1 (2008 г.) Android не поддерживала программную клавиатуру и требовала наличия на устройствах физических клавиш. С выходом SDK 1.5 в апреле 2009 г. эта проблема была устранена, и был добавлен ряд других компонентов, таких как расширенные возможности записи медиафайлов, виджеты и Live Folders (“живые папки”).

В сентябре 2009 г. вышел выпуск ОС Android 1.6, а через месяц — Android 2.0, что привело к появлению целого потока Android-устройств к рождеству 2009 г. В этом выпуске появились расширенные возможности поиска и речевого воспроизведения текста

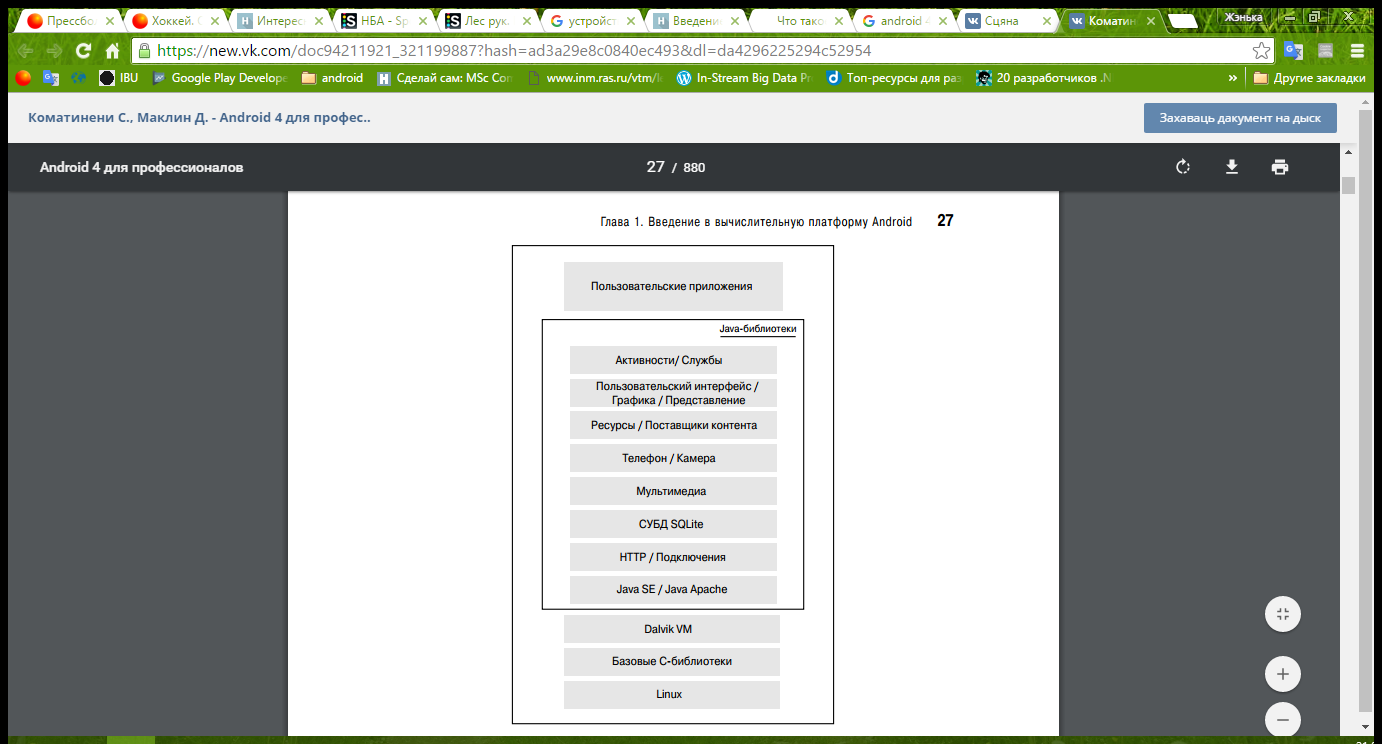
Самыми заметными нововведениями в Android 2.3 были дистанционное удаление администраторами конфиденциальных данных, возможность проведения фото- и видеосъемки при низкой освещенности, работа с точками доступа Wi-Fi, существенное повышение производительности, расширенная функциональность Bluetooth, установка приложений по желанию на SD-карту, поддержка OpenGL ES 2.0, улучшенное резервное копирование, повышенное удобство поиска, поддержка Near Field Communications для обработки кредитных карт, существенно улучшенная поддержка движения и датчиков (подобно Wii), видеочат и усовершенствованный Android Market. Версия Android 3.0 ориентирована на планшетные устройства и гораздо более мощные двухъядерные процессоры, такие как NVIDIA Tegra 2. Основным новшеством данного выпуска стала поддержка экранов больших размеров. Введена существенно более новая концепция под названием фрагменты (fragments). Фрагменты присутствуют в Android 3.0 повсеместно.

Рисунок 2.1 – Компоненты ОС Android

Были добавлены возможности, больше относящиеся к настольным компьютерам, вроде панели действий (action bar) и перетаскивания (drag and drop).

В Android 3.0 существенно улучшены виджеты домашнего экрана, и стали доступными дополнительные элементы управления пользовательского интерфейса. В области трехмерной графики была улучшена поддержка OpenGL с Renderscript для дальнейшего добавления ES 2.0. Словом, Android 3.0 — великолепный инструмент для работы с планшетами. Тем не менее, функциональность версии Android 3.0 была ограничена для планшетов. На время выхода выпуска 3.0 ветвь 2.x ОС Android продолжала обслуживать телефоны, тогда как ветви 3.x обслуживали планшеты.

Начиная с выпуска 4.0, эти ветви в ОС Android были слиты в единый SDK. Для пользователей телефонов главное отличие в интерфейсе связано с тем, что принципы работы с планшетами также стали доступными в телефонах

Ниже перечислены ключевые аспекты работы пользователей в Android 4.0.

* Улучшенный способ организации приложений в папки на домашних страницах.
* Возможность перетаскивания приложений и папок в лоток избранного, который всегда присутствует в нижней части экрана устройства.
* Оптимизация уведомлений на основе типа устройства. В случае небольших устройств они отображаются в верхней части экрана, а в случае устройств больших размеров — в системной области внизу экрана.
* Виджеты с возможностью прокрутки и изменения размеров.
* Разнообразные способы разблокирования экранов.
* Проверка орфографии.
* Модернизированная анимация на основе изменения свойств объектов, включая представления.
* Фиксированное количество поведений списковых виджетов из версии 3.0.
* Значительно более зрелая панель действий со встроенным поиском.
* Поддержка множества мобильных стандартов: Advanced Audio Distribution Profile (A2DP: возможность использования внешних колонок), Realtime Transport Protocol (RTP: для передачи потокового аудио/видео по протоколу IP), Media Transfer Protocol (MTP), Picture Transfer Protocol (PTP: для привязки к компьютерам с целью загрузки фотографий и медиафайлов) и Bluetooth Headset Profile (HSP).
* Полное шифрование устройства.
* Технические средства защиты авторских прав (Digital Rights Management — DRM).
* Зашифрованное хранилище и пароли.
* Интерфейс Social API, включающий персональные профили. • Расширенный интерфейс Ca

При работе с Android в Google много усилий было потрачено на оптимизацию проектирования для маломощных карманных устройств. Карманные устройства отстают от своих настольных собратьев по объему памяти и скорости лет на восемь-десять. Их вычислительные возможности ограничены. Как результат, при работе на карманных устройствах требования к производительности оказываются очень жесткими, и проектировщикам приходится оптимизировать каждую мелочь. Если посмотреть список пакетов в Android, можно увидеть, что являются полнофункциональными и обширными

Эти проблемы заставили Google пересмотреть стандартную реализацию JVM во многих отношениях. Ключевой фигурой в реализации Google собственной JVM является Дэн Борнштейн (Dan Bornstein), который написал Dalvik VM (по названию города в Исландии). Dalvik VM объединяет сгенерированные файлы Java-классов с одним или несколькими исполняемыми файлами Dalvik Executable (.dex). Целью Dalvik VM является нахождение всевозможных способов оптимизации JVM в плане занимаемого пространства, производительности и срока службы батареи.

Окончательный исполняемый код в Android, как результат наличия Dalvik VM, основан не на байт-коде Java, а на .dex-файлах. Это означает невозможность непосредственного выполнения байт-кода Java; необходимо взять файлы Java-классов и преобразовать их в готовые для связывания .dex-файлы.

Мы уже познакомились с историей развития ОС Android и с ее возможностями оптимизации, включая Dalvik VM, и при этом был упомянут программный стек Java. В настоящем разделе мы рассмотрим аспект разработки для Android. Основой платформы Android является ядро Linux, которое отвечает за драйверы устройств, доступ к ресурсам, управление электропитанием и другие функции ОС. К поставляемым драйверам относятся драйверы для экрана, камеры, клавиатуры, Wi-Fi, флеш- памяти, звука и IPC (inter-process communication — взаимодействие процессов). И хотя ядром является Linux, большинство приложений — если не все — на Android-устройствах, такие как Motorola Droid, разработаны на Java и выполняются с помощью Dalvik VM.

На следующем уровне, непосредственно над ядром, находится ряд библиотек C/C++: OpenGL, WebKit, FreeType, Secure Sockets Layer (SSL), библиотека времени выполнения C (libc), SQLite и Media. Системная библиотека C, за основу которой взят дистрибутив BSD (Berkeley Software Distribution), оптимизирована (и примерно вполовину ужата) для устройств со встроенной ОС Linux. Библиотека Surface Manager управляет доступом к системе дисплея и поддерживает двухмерную и трехмерную графику. Библиотека WebKit отвечает за поддержку браузера; именно она используется в Google Chrome и Apple Safari. Библиотека FreeType отвечает за поддержку шрифтов. SQLite представляет собой реляционную СУБД, доступную непосредственно на устройстве. SQLite также является независимой разработкой с открытым кодом для реляционных СУБД и не привязана конкретно к Android. С базами данных Android можно работать и с помощью других средств, предназначенных для SQLite. В основном инфраструктура для разработки приложений обращается к этим базовым библиотекам через Dalvik VM — шлюз для платформы Android. Как уже было сказано в предыдущих разделах, виртуальная машина Dalvik оптимизирована для работы нескольких экземпляров виртуальных машин.

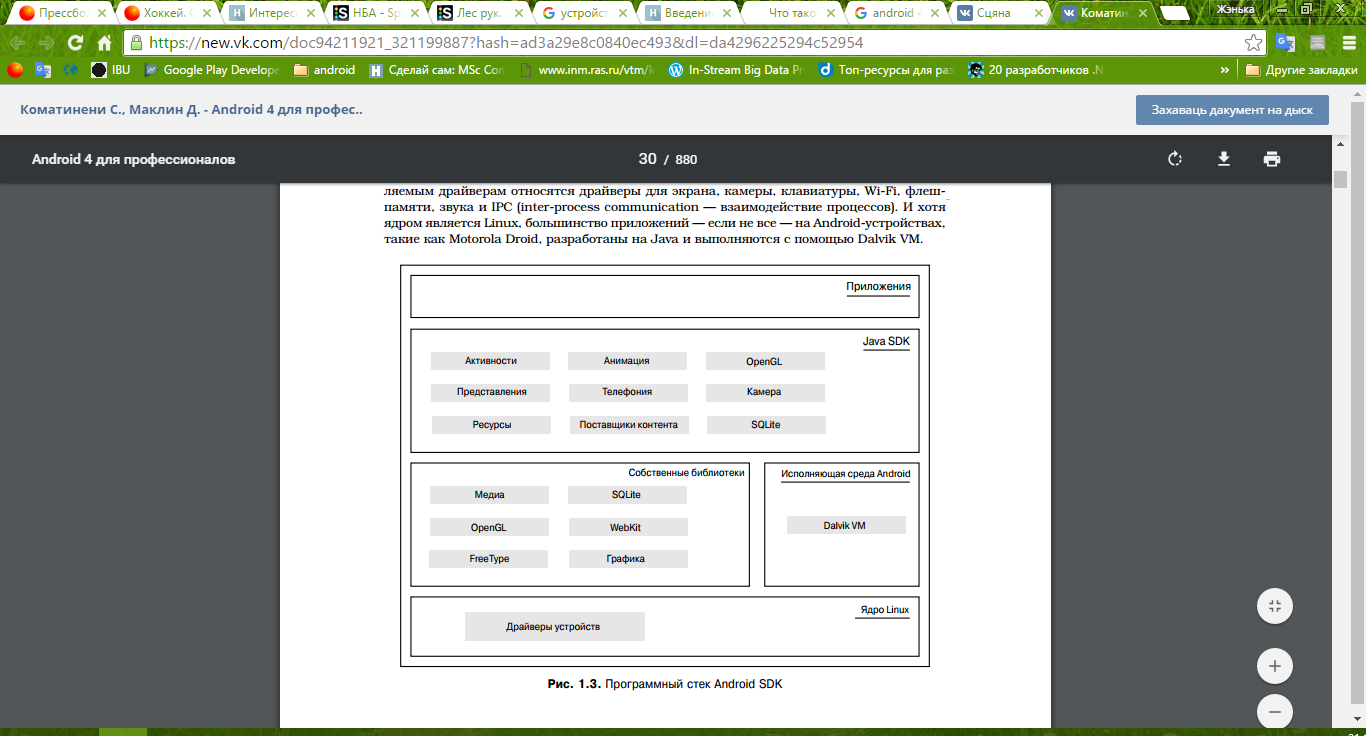


Рисунок 2.2 – Устройстов ОС Android

При обращении Java-приложений к этим базовым библиотекам каждое приложение получает собственный экземпляр виртуальной машины. Основные библиотеки из API-интерфейса Android Java предназначены для работы с телефонией, ресурсами, местоположениями, пользовательским интерфейсом, поставщиками контента (данные) и диспетчерами пакетов (установка, защита и т.д.). Программисты разрабатывают приложения для конечных пользователей на базе этого Java API. Примеры приложений для конечных пользователей на устройстве — Home (Домашний экран), Contacts (Контакты), Phone (Телефон) и Browser (Браузер).

ОС Android также поддерживает библиотеку двумерной графики собственной разработки Google под названием Skia, которая написана на C и C++. Кроме того, Skia образует ядро браузера Google Chrome. Однако API-интерфейсы для работы с трехмерной графикой основаны на реализации OpenGL. OpenGL ES содержит подмножества OpenGL ES, ориентированные на встроенные системы. С точки зрения медиа платформа Android поддерживает наиболее распространенные форматы аудиофайлов, видеофайлов и изображений. С точки зрения беспроводной связи Android содержит API-интерфейсы для поддержки Bluetooth, EDGE, 3G, Wi-Fi и телефонии Global System for Mobile Communication (GSM), в зависимости от оборудования (см. рисунок 2.2)

1. СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Все программное средство разбито на отдельные модули. Это необходимо для обеспечения гибкой структуры ПО. При данном подходе допускается модернизация любого из выбранных модулей без изменения остальных, позволяется возможность адаптировать данное программное средство для различных платформ. Для разрабатываемой программы были выделены модули, которые представлены в структурной схеме.

На каждый логически выделенный блок программы возлагаются определенные задачи. Кроме того, каждый блок программы так или иначе связан с остальными блоками, чтобы обеспечить работоспособность всего программного средства в целом. Связь, как правило, реализуется посредством обмена сообщений.

Разрабатываемый программный продукт предназначена для обмена сообщения между пользователями в определенном месте на основе геолакационных данных. Поэтому на основании задания на разработку и исследования литературных источников можно выделить следующие функции, которая должна обеспечивать система

* Регистрация и аутентификация пользователей
* Определение местоположения пользователей
* Обмен сообщениями между пользователями
* Уведомление о новых сообщениях
* Настройка созданных чатов

Программный продукт состоит из двух основных систем: Api – сервер, обрабатывающий данные пользователей и клиентское приложение, размещаемое на устройствах пользователей.

На основе указанных требований продукта и разрабатываемых подсистем, можно выделить следующие блоки:

Блоки Api-сервера

* + - * Блок уведомления пользователей о происходящих событиях
      * Блок аутентификации и регистрации пользователей
      * Блок работы с данными через протокол HTTP
      * Блок реализации бизнес-логики приложения
      * Блок работы обеспечивающий хранение данных

Блоки мобильного приложения

* + - * Блок работы с api – сервером
      * Блок вывода уведомлений
      * Блок настройки
      * Блок определяющий местоположение пользователей
      * Блок работы с сообщениями

Структурная схема, иллюстрирующая перечисленные блоки и связи между ними приведена на чертеже.

Кроме функциональных блоков на схеме представлены два блока данных:

* + - База данных
    - Брокер сообщений

Каждый из перечисленных блоков системы решает определенные задачи. Для обеспечения работоспособности системы в целом, блоки связаны между собой. Связь между блоками реализуется путем обмена сообщения.

Опишем более подробно каждый из блоков.

***Блок работы с данными по протоколу HTTP***

Отвечает за предоставление данных в определенном формате для клиентских приложений, в частности для мобильного клиента. Реализован в виде REST сервиса, который отдаёт данные в формате Json. Также данный блок создан с использованием паттерна проектирования Facade, который позволяет скрыть внутренние детали реализации и предоставляет определенный Api.

***Блок аутентификации и регистрации пользователей***

Данный блок отвечает за регистрацию пользователей и сохранения их в базе данных. Так же для аутентификации используются Jwt token. Поэтому этот блок также отвечает за создание и хранение пары открытого и закрытого ключа. Задачей этого блока является подписание созданного токена пользователя закрытым ключом, и проверка подлинности подписи при аутентификации открытым ключом.

***Блок реализации бизнес-логики приложения***

Отвечает за выполнения бизнес правил приложения. Принимает данные от *блока работы с данными по протоколу HTTP* и посылает их в *блок работы обеспечивающий хранение данных.* Также восстанавливает все события, происходящие с определённым агрегатом, для корректного выполнения бизнес-логики. Может быть размещен на отдельном сервере для увеличения производительности приложения и снятия нагрузки с основных серверов

***Блок работы с базой данных***

Основная задача данного блока состоит в принятии событий от *блока реализации бизнес-логики приложения* и комбинирования принятых событий для построения нужных моделей данных. Может быть размещен на отдельном сервере для увеличения производительности приложения и снятия нагрузки с основных серверов

***Блок уведомления пользователей о происходящих событиях***

Позволяет производить двухсторонний обмен сообщениями между клиентским приложением и api сервером. Данный обмен реализован по средствам протокола Websoсket. С помощью этого блока происходит уменьшение нагрузки на основные сервера, так как сервер оповещает клиента о необходимых ему событиях, а не клиент постоянно производит запросы к серверу с необходимым интервалом. Так же посредством данного блока удается реализовать постоянное обновление данных на клиенте

***Блок работы с api – сервером***

Данный блок ответственен за получение данных с api-сервера. Обеспечивает загрузку данных в формате json, преобразование к определённым объектам клиентского приложения. Данный блок является составной частью мобильного приложения.

***Блок вывода уведомлений***

Сфера деятельности данного блока – производить оповещения пользователя о различных событиях в приложении в виде всплывающих окон. Примеры: потеря связи с сервером, пользователь находится не в зоне действия чата и т. д. Данный блок является составной частью мобильного приложения.

***Блок определяющий местоположение пользователей***

Отвечает за постоянный мониторинг местоположения пользователя. Данный блок реализуется при помощи специальных сервисов Google Play. Поэтому для корректной работы разрабатываемого программного продукта на мобильном устройстве должно быть установлено приложение Google Play Services. Данные от этого сервиса приходит в виде текущей координаты пользователя, которая включает долготу, широту и высоту над уровнем моря. Данный блок является составной частью мобильного приложения.

***Блок настройки***

Позволяет производить изменения пользовательских данных пользователей. А также изменения настроек чат, если пользователь является в нем админом. Блок связан с блоком работы с api – сервером. Данный блок является составной частью мобильного приложения.

***Блок работы с сообщениями***

Отвечает за отображение сообщений пользователей в чате, их изменение и удаление. Производит подгрузку новых сообщений. Блок связан с блоком работы с api – сервером. Данный блок является составной частью мобильного приложения.

1. ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Рассмотрим подробно функционирование программного средства. Для этого проведем анализ блоков, которые входят в состав программы, и рассмотрим назначение всех функций и структур данных.

4.1 Блок работы с данными по протоколу HTTP

Отвечает за предоставление данных в определенном формате для клиентских приложений, в частности для мобильного клиента. Реализован в виде REST сервиса, который отдаёт данные в формате Json. Также данный блок создан с использованием паттерна проектирования Facade, который позволяет скрыть внутренние детали реализации и предоставляет определенный Api.

Концепция MVC позволяет разделить данные, представление и обработку действий пользователя на три отдельных компонента:

* Модель ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Model). Модель предоставляет знания: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти знания можно визуализировать.
* Представление, вид ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) View). Отвечает за отображение информации (визуализацию). Часто в качестве представления выступает [форма (окно)](https://ru.wikipedia.org/wiki/%D0%9E%D0%BA%D0%BD%D0%BE_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) с графическими элементами.
* Контроллер ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Controller). Обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции.

Важно отметить, что как представление, так и контроллер зависят от модели. Однако модель не зависит ни от представления, ни от контроллера. Тем самым достигается назначение такого разделения: оно позволяет строить модель независимо от визуального представления, а также создавать несколько различных представлений для одной модели.

Наиболее типичная реализация отделяет вид от модели путём установления между ними протокола взаимодействия, используя аппарат событий (подписка/оповещение). При каждом изменении внутренних данных в модели она оповещает все зависящие от неё представления, и представление обновляется. Для этого используется шаблон «[наблюдатель](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B1%D0%BB%D1%8E%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))». При обработке реакции пользователя представление выбирает, в зависимости от нужной реакции, нужный контроллер, который обеспечит ту или иную связь с моделью. Для этого используется шаблон «[стратегия](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D0%B0%D1%82%D0%B5%D0%B3%D0%B8%D1%8F_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))», или вместо этого может быть модификация с использованием шаблона «[команда](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))». А для возможности однотипного обращения с подобъектами сложно-составного иерархического вида может использоваться шаблон «[компоновщик](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%BE%D0%B2%D1%89%D0%B8%D0%BA_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))». Кроме того, могут использоваться и другие шаблоны проектирования, например, «[фабричный метод](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F))», который позволит задать по умолчанию тип контроллера для соответствующего вида.

В данном проекте можно сказать использован усеченный паттерн MVC, не использующий View, так за отображение данных отвечают блоки клиентского приложения. Рассмотрим некоторые функции данного блока

* [HttpPost("create")]

public async Task<IActionResult> Create([FromBody]CreateChartViewModel model)

Создает новый чат на основе переданных данных от клиента

public class CreateChartViewModel

{

[Required]

public string Title { get; set; }

[Required]

public double Latitude { get; set; }

[Required]

public double Longitude { get; set; }

[Required]

public double Radius { get; set; }

public string Description { get; set; }

}

Данный класс содержит следующую базовую информацию

* + - Заглавие
    - Широта
    - Долгота
    - Радиус действия
    - Описание

Как видно из сигнатуры метода, возвращается объект Task, который позволяет выполнять компилятору определенные манипуляции и выполнять этот метод асинхронно

* [HttpGet("inlocation")]

public async Task<List<ChartViewModelShort>> GetCharts(Location location)

Возвращает список все доступных чатов, в которых пользователь может участвовать на основе его местоположения

public class Location

{

public double Longitude { get; set; }

public double Latitude { get; set; }

}

Данный класс содержит следующую информацию

* + - Широта
    - Долгота
* [HttpPost("create")]

public async Task<IActionResult> Create([FromBody]CreateMessageViewModel model)

Создает новое сообщение на основе переданных данных от клиента

public class CreateMessageViewModel

{

[Required]

public string ChartId { get; set; }

[Required]

public string Message { get; set; }

}

Данный класс содержит следующую информацию

* + - Идентификатор чата
    - Сообщение
* [HttpPost("register")]

public async Task<IActionResult> Register([FromBody]UserRegisterViewModel model)

Создание нового пользователя на основе представленных данных

public class UserRegisterViewModel

{

[Required(ErrorMessage = MessageConstants.Errors.EmailRequired)]

[EmailAddress(ErrorMessage = MessageConstants.Errors.EmailInvalid)]

public string Email { get; set; }

[Required(ErrorMessage = MessageConstants.Errors.PasswordRequired)]

public string Password { get; set; }

[Required(ErrorMessage = MessageConstants.Errors.ConfirmPasswordRequired)]

[Compare(nameof(Password), ErrorMessage = MessageConstants.Errors.ConfirmPasswordRequired)]

public string ConfirmPassword { get; set; }

[Required(ErrorMessage = MessageConstants.Errors.UserNameRequired)]

public string UserName { get; set; }

}

Данный класс содержит следующую информацию

* + - Email пользователя
    - Логин – предназначен для идентификации пользователя в чате, для каждого чата может быть разный
    - Пароль
    - Подтверждающий пароль, должен быть такой же, как и основной

4.2 Блок аутентификации и регистрации пользователей

Данный блок отвечает за регистрацию пользователей и сохранения их в базе данных. Так же для аутентификации используются Jwt token. Поэтому этот блок также отвечает за создание и хранение пары открытого и закрытого ключа. Задачей этого блока является подписание созданного токена пользователя закрытым ключом, и проверка подлинности подписи при аутентификации открытым ключом.

Так же этот блок реализован в виде oAuth провайдера. Что позволяет в следующем с легкостью, производить авторизацию с помощью различных соцсетей и быть провайдером для других приложений.

Сейчас более подробно о том, что такое oAuth.

OAuth — открытый протокол [авторизации](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F), который позволяет предоставить третьей стороне ограниченный доступ к защищённым ресурсам пользователя без необходимости передавать ей (третьей стороне) логин и пароль. Например, пользователь, который хочет предоставить онлайн-сервису печати фотографий доступ к фотографиям своего [Facebook-аккаунта](https://ru.wikipedia.org/wiki/Facebook), не должен сообщать сервису пароль от этого аккаунта. Вместо этого он проходит авторизацию непосредственно в Facebook, который (с разрешения пользователя или администратора сервиса) предоставляет сервису онлайн-печати полномочия доступа к фотографиям.

При использовании OAuth-[авторизации](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F) пользователь не передает свой логин и пароль к защищенным ресурсам напрямую в приложение. Поэтому:

* У пользователя больше оснований доверять приложению, поскольку пользователь может быть уверен, что несанкционированный доступ к его личным данным невозможен. Не владея логином и паролем пользователя, приложение сможет выполнять только те действия с данными, которые разрешил пользователь, и никакие другие.
* При разработке приложения не нужно заботиться об обеспечении конфиденциальности логина и пароля пользователя. Логин и пароль не передаются приложению, а, следовательно, не могут попасть в руки злоумышленников.

Рассмотрим более подробно некоторые функции, реализуемы в нашем приложении

* public override Task ValidateTokenRequest(ValidateTokenRequestContext context)

Данная функция проверяет, является ли токен, полученный пользователем при авторизации целостным и не поврежденным

* public override async Task GrantResourceOwnerCredentials(GrantResourceOwner CredentialsContext context)

Здесь происходит проверка, введенных данный пользователем, например, пароль и логин. А именно: существует ли данный пользователь в приложение с таким паролем.

Для того, чтобы обезопасить пользователей от взлома их аккаунтов, если вдруг злоумышленники получат доступ к базе данных, пароли не хранятся в явном виде. А используются стандартные алгоритмы хеширования паролей, как MD5

4.3 Блок реализации бизнес-логики приложения

Отвечает за выполнения бизнес правил приложения. Принимает данные от *блока работы с данными по протоколу HTTP* и посылает их в *блок работы обеспечивающий хранение данных.* Также восстанавливает все события, происходящие с определённым агрегатом, для корректного выполнения бизнес-логики. Может быть размещен на отдельном сервере для увеличения производительности приложения и снятия нагрузки с основных серверов

В этом блоке в основном и происходит реализация принципов CQRS и EventSourcing, а также принципов DDD.

Для начала нам нужно послать сообщение из *блока работы с данными по протоколу HTTP*. Для этого реализуем паттерн проектирования ServiceBus, который будет высокоуровневым интерфейсом над определенным ПО по пересылки сообщений.

Основной принцип сервисной шины — концентрация обмена сообщениями между различными системами через единую точку, в которой, при необходимости, обеспечивается [транзакционный контроль](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), [преобразование данных](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B2%D0%B5%D1%80%D1%82%D0%B0%D1%86%D0%B8%D1%8F_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), сохранность сообщений. Все настройки обработки и передачи сообщений предполагаются также сконцентрированными в единой точке, и формируются в терминах служб, таким образом, при замене какой-либо информационной системы, подключённой к шине, нет необходимости в перенастройке остальных систем.

Наименование подобрано по аналогии с [системной шиной компьютера](https://ru.wikipedia.org/wiki/%D0%A8%D0%B8%D0%BD%D0%B0_(%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80)), позволяющей подключать несколько устройств и передавать данные между ними по одному набору проводников.

И так вот наш интерфейс для команд

Task SendAsync(params ICommand[] commands);

Данный метод производит асинхронную посылку команд

public interface ICommand

{

ICommandMetadata Metadata { get; set; }

}

public interface ICommandMetadata

{

string CommandId { get; set; }

string UserId { get; set; }

string TypeName { get; set; }

DateTime CreatedDate { get; set; }

}

Данный интерфейс содержит следующие поля:

* + - идентификатор команд
    - идентификатор пользователя, пославший данную команду
    - тип передаваемой команды в терминах CLR
    - дата создания команды

Теперь эту команду нужно неким образом обработать. Для этого реализуем еще один шаблон проектирования Dispatcher, который будет вызывать обработчики сообщений, в зависимости от типа нашего сообщения в контексте CLR.

Определим интерфейс

Task DispatchAsync(Object message)

Мы можем видеть, что здесь передается любой объект и в последствии происходит его асинхронная обработка. Но для начала мы должны определить необходимые обработчики событий. Сделать это мы может посредством реализации определённого интерфейса обработчиком.

public interface IMessageHandlerAsync<in TMessage>

{

Task HandleAsync(TMessage message);

}

Но для начала, нам нужно зарегистрировать данные обработчики, делается это посредством класса конфигурации диспатчера, который передается при создании

public static Dispatcher Create(Func<DispatcherConfiguration, DispatcherConfiguration> configurationAction, ILogFactory loggingFactory)

public class DispatcherConfiguration

{

public DispatcherHandlerRegistry DispatcherHandlerRegistry { get; set; }

public int NumberOfRetries { get; set; }

public IServiceProvider ServiceProvider { get; set; }

public Type MessageHandlerMarkerInterface { get; set; }

}

Данный класс содержит следующую информацию

* + - Количество попыток, при которых должен вызываться обработчик сообщения, если происходит какая-нибудь ошибка
    - Тип интерфейса, от которого должны наследовать все обработчики
    - И класс, который занимается регистрацией обработчиков событий

И так, в обработчике событий мы должны сформировать агрегат на основе восстановленных событий в базе данных, выполнить какое-нибудь действие в агрегате, сохранить новое событие в базе данных и по шине сообщений послать это событие на обработчик для создания или обновления моделей базы данных.

И так, определим интерфейсы для работы с событиями и агрегатами

* Task Save(String aggregateId, TAggregate aggregate, ICommandMetadata commandMetadata);

Данный метод производит сохранение произведенных событий с базой данных и посылку события по шине

* Task<TAggregate> GetById(String id);

Производит восстоновление агрегата из базы данных на основе событий по идентификатору агрегата

* Task Perform(String id, Action<TAggregate> action, ICommandMetadata commandMetadata);

Произовит несколько действий с агрегатом, такие как

* + Восстановление агрегата из базы данных,
  + Выполнение определенных действий
  + Сохранение событий
  + Посылка событий по шине сообщений

Рассмотрим интерфейс, выполняющий операции чтения и записи событий в базу данных

public interface ITransitionRepository

{

Task AppendTransition(Transition transition);

Task AppendTransitions(IEnumerable<Transition> transitions);

Task<List<Transition>> GetTransitions(String streamId, Int32 fromVersion, Int32 toVersion);

Task<IEnumerable<Transition>> GetTransitions();

Task<IEnumerable<Transition>> GetTransitions(Int32 startIndex, Int32 count);

Task RemoveTransition(String streamId, Int32 version);

Task RemoveStream(String streamId);

Task<long> CountTransitions();

Task CreateIndexes();

}

Рассмотрим каждый из методов более подробно (как видим каждая из операций производится асинхронно):

* Task AppendTransition(Transition transition);

Дабляет событие в базу данных

* Task AppendTransitions(IEnumerable<Transition> transitions);

Добавляет несколько событий в базу данных

* Task<IEnumerable<Transition>> GetTransitions();

Читает все события из базы данных

* Task<List<Transition>> GetTransitions(String streamId, Int32 fromVersion, Int32 toVersion);

Читает события по определенным параметрам

* Task<IEnumerable<Transition>> GetTransitions(Int32 startIndex, Int32 count);
* Task RemoveTransition(String streamId, Int32 version);

Удаляет события по определенным параметрам

* Task RemoveStream(String streamId);

Удаляет все события агрегата

* Task<long> CountTransitions();

Возвращает кол-во событий

* Task CreateIndexes();

Создает индекс в базе данных

Основной интерфейс определяющий событие

public class Transition

{

public TransitionId Id { get; private set; }

public DateTime Timestamp { get; set; }

public string AggregateTypeId { get; set; }

public List<TransitionEvent> Events { get; private set; }

}

Содержит следующие данные:

* + - Идентификатор создания события
    - Время создания событий
    - Идентификатор агрегата события
    - Список событий за один поток

Для хранения данных наиболее удобно и выгодно в плане производительности использовать нереляционные базы данных. Так как наши события не имеют четкой схемы, каждое из них содержит разные данные схему данных(разную информацию). Также с базой данных событий, называемой write база, производятся только операции чтения и записи, с чем очень хорошо справляется не реляционные базы. В моем программном продукте я выбрал MongoDB.

* 1. Блок работы с базой данных

Основная задача данного блока состоит в принятии событий от *блока реализации бизнес-логики приложения* и комбинирования принятых событий для построения нужных моделей данных. Может быть размещен на отдельном сервере для увеличения производительности приложения и снятия нагрузки с основных серверов

Этот блок в некотором роде схож с *блоком реализации бизнес-логики приложения*. Он также обрабатывает события пришедшие из шины сообщений. Но при обработке не происходит проверка какой-нибудь бизнес логики из агрегатов, а формируются необходимые модели, которые необходимы нашему приложению для чтения. Поэтому формируемые модели называются *Ридмодели*. Данный подход позволяет сократить время чтения данных из базы данных приложения, так как необходимые для чтения модели уже сформированы.

Может возникнуть вполне очевидный вопрос, а что же делать, когда у нас изменяется схема ридмоделей? Допустим нам нужно добавить пару новых полей заполнить их информацией на основе прошедших событий. Благо у нас все события сохранены в write базе, и нам всего лишь достаточно по-новому обработать все наши события. Удобно, не правда ли? Этот процесс называется регенирацией базы данных. Правда он имеет и некоторые недостатки, самый главный из них на мой взгляд – это проблемы производительности и времени. Ведь представьте, что у вас в базе миллионы, что уж там, также сотни миллионов событий, и каждое из них вам нужно обработать по-новому и записать в базу данных. Многих этот может отпугивать, однако на мой взгляд достоинства данного подхода перевешивают недостатки. Тем более с каждым годом проблемы с производительностью становятся все меньше и меньше из-за развития аппаратной части компьютеров.

А теперь рассмотрим базовые интерфейсы для чтения данных

* + - public virtual Task<T> GetByIdAsync(string id, TFilter filter = null)

Производит считывание ридмодели определенного типа из базы данных по идентификатору и специальному фильтру, если он определен

* + - public Task<List<T>> GetAllAsync()

Производит считывание всех ридмоделей определенного типа

* + - public Task<long> CountAsync(TFilter filter)

Находит количество всех ридмоделей по определенному фильтру

* + - public Task<bool> IsExists(TFilter filter)

Проверяет существует ли ридмодель в базе данных по определённому фильтру

* + - public async Task<IEnumerable<T>> GetByFilter(TFilter filter)

Производит считывание всех ридмоделей определенного типа по фильтру из базы данных

* + - protected IFindFluent<T, T> GetCursorByFilter(TFilter filter)

Формирует курсор к данным в базе данных по определенному фильтру

* + - private List<FilterDefinition<T>> GetFilterQueries(TFilter filter)

Преобразовывает фильтр в запрос приемлемуму виду для драйвера базы данных

* + - protected virtual List<SortDefinition<T>> BuildSortExpression(TFilter filter)

Преобразовывает объеет, содержащий информацию о сортировке, к приемлемумы виду для дрйвера базы данных

* + - public abstract IEnumerable<FilterDefinition<T>> BuildFilterQuery(TFilter filter);

Преобразовывает фильтр в запрос приемлемуму виду для драйвера базы данных

* 1. Блок уведомления пользователей о происходящих событиях

Позволяет производить двухсторонний обмен сообщениями между клиентским приложением и api сервером. Данный обмен реализован по средствам протокола Websoсket. С помощью этого блока происходит уменьшение нагрузки на основные сервера, так как сервер оповещает клиента о необходимых ему событиях, а не клиент постоянно производит запросы к серверу с необходимым интервалом. Так же посредством данного блока удается реализовать постоянное обновление данных на клиенте

Для двухстороннего взаимодействия клиента и сервера существуют несколько различных технологий взаимодействия, рассмотрим их более подробно

* Polling  
  Самый простой, но самый не эффективный, метод: клиент раз в несколько секунд опрашивает сервер на наличие событий. Даже если ничего нет, то клиент все равно делает запрос — а мало ли что придет.  
  Плюсы:
  + Просто
  + Данные могут быть пожаты  
    Минусы:
  + Очень много лишних запросов
  + События всегда приходят с опозданием
  + Серверу приходится хранить события пока клиент не заберет их или пока они не устареют
* Long polling

Улучшенный вариант предыдущего метода. Клиент отправляет запрос на сервер, сервер держит открытым соединение пока не придут какие-нибудь данные или клиент не отключится самостоятельно. Как только данные пришли — отправляется ответ и соединение закрывается и открывается следующее и так далее.

Плюсы

* + Минимальное количество запросов
  + Высокая временная точность событий
  + Сервер хранит события только на время реконнекта
* Web sockets

Это бинарный дуплексный протокол, позволяющий клиенту и серверу общаться на равных. Этот протокол можно применять для игр, чатов и всех тех приложений где вам нужны предельно точные события близкие к реальному времени.  
Плюсы по сравнению с Long Polling:

* + Поднимается одно соединение
  + Предельно высокая временная точность событий
  + Управление сетевыми сбоями контролирует браузер  
    Минусы по сравнению с Long Polling:
  + HTTP не совместимый протокол, нужен свой сервер, усложняется отладка   
    Так почему же стоит применять SSE, раз у нас есть такой прекрасный протокол WebSockets?! Во-первых, не каждому веб-приложению необходима двусторонняя связь — подойдет и SSE. Во-вторых, SSE — HTTP совместимый протокол и вы можете реализовать рассылку событий на любом веб-сервере.
* Протокол Server-Sent Event

Клиент отправляет запрос на сервер, сервер в ответ отправляет следующий заголовок:

Content-Type: text/event-stream

И не закрывает соединение. Вот и все — SSE работает! Чтобы отправить клиенту какие-то данные сервер просто пишет в сокет строку следующего формата:

data: My message\n\n

Если необходимо отправить несколько строк данных, то формат будет следующим:

data: {\n

data: "msg": "hello world",\n

data: "id": 12345\n

data: }\n\n

Вот, впринципе, и вся база протокола. Кроме этого сервер может отправлять id сообщения это нужно на случай если соединение было разорвано. Если соединение было сброшено, то клиент при попытке подключения отправит специальный заголовок (Last-Event-ID), чтобы восстановить утраченные события:

В данном программном продукте для двухстороннего общения используется протокол Websockets, однако он не используется в явном виде. Существует множество различных библиотек, позволяющих организовать взаимодействие с клиентом в удобном для программиста виде. Одна из них и самая популярная это SignalR.

SignalR – это библиотека для создания многопользовательских real-time ASP.NET (и не только) приложений. Она состоит из набора серверных и клиентских библиотек, и представляет собой абстракцию над целым набором транспортов.

SignalR был придуман и реализован двумя разработчиками Microsoft: [Damian Edwards](https://twitter.com/DamianEdwards) и [David Fowler](https://twitter.com/davidfowl). В своих твиттерах они часто пишут полезную информацию о SignalR и сообщают о новостях.

**Транспорты**

Как я уже писал, SignalR – это абстракция над набором транспортов. Когда пользователь открывает страницу, работающую с SignalR, браузер опрашивает сервер на предмет поддержки разных транспортов и затем пытается подсоединиться по самому оптимальному для данного случая транспорту. Приоритет выглядит следующим образом(см. рисунок 4.1):

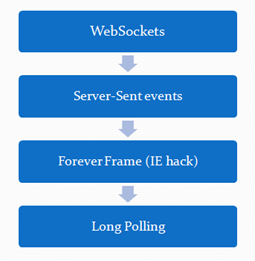
[](http://lh4.ggpht.com/-syj3ju_xXGQ/UKkYWdYFRkI/AAAAAAAAc8g/fNeljOtnFWA/s1600-h/image%5b12%5d.png)

Рисунок 4.1 – Приоритет соединений

То есть если сервер и клиент поддерживают WebSockets, то будет установлено WebSockets-соединение и все будут счастливы. Если же нет – то далее будет проверка, поддерживает ли клиент SSE, и если да – будет установлено это соединение. В случае IE сразу же будет испробован подход Forever Frame (невидимо висящий iframe устанавливает соединение и получает JS-инструкции с сервера) – это IE-хак, т.к. даже 10-я версия IE не поддерживает SSE. Молодцы, нечего сказать.

И в конце-концов, если клиент не поддерживает ни один из этих способов, или произошла ошибка, то SignalR откатится до самого надежного способа – Long Polling, который работает практически везде.

**Архитектура**

Архитектура SignalR очень проста. На сервере реализованы 2 вида API: низкоуровневый (PersistentConnection API) и высокоуровневый (Hub API), причем Hub опирается на PersistentConnection. Вы можете использовать любой из них, но в большинстве случаев вам будет достаточно возможностей, предоставляемых Hub API (см. рисунок 4.2).

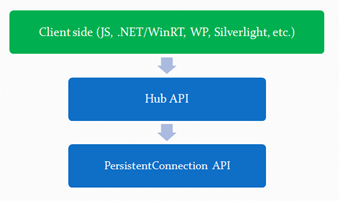
[](http://lh4.ggpht.com/-KgnnlLVi6RU/UKkYYosQ8jI/AAAAAAAAc8w/9wfLNhxUCyY/s1600-h/image%5b13%5d.png)

Рисунок 4.2 – Api SignalrR

И так в разрабатываемом продукте оповещение клиента в основном происходит из *блока работы обеспечивающего хранение данных* о каких-либо изменениях в базе данных. Данное оповещение происходит путем послания сообщения на шину сообщений с помощью данного метода

Task SendRealTimeMessageAsync(params IMessage[] messages);

Который принимает в виде аргументов массив объектов, каждый из которых реализует интерфейс IMessage

public interface IMessage

{

MessageMetadata Metadata { get; set; }

}

public class MessageMetadata

{

public string MessageId { get; set; }

public string UserId { get; set; }

}

Класс MessageMetadata содержит следующую информацию

* Идентификатор пользователя, пославшего сообщения
* Идентификатор сообщения

Далее сообщение принимается обработчиком и вызывается Hub Signalr. Отобразим список реализуемых методов:

* public async Task connectToChat(string connectionId, IEnumerable<string> chartIds)

Производит подсоединение клиента к определенным чатам, для получения обновлений от них

* public async Task disconnectFromChat(string connectionId, IEnumerable<string> chartIds)

Производит отсоединения клиента от определенных чатов, для того чтобы не нагружать сервер, если клиенту уже не нужны обновления

Опишем более подробно классы, которые содержат информацию об изменениях в чате, предназначаемые определенным клиентам

* public class ChartCreatedSignal : Message

{

public string ChartId { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public Location Location { get; set; }

public double Radius { get; set; }

public string OwnerId { get; set; }

public IList<string> AdminIds { get; set; } = new List<string>();

public IList<Participant> Participants { get; set; } = new List<Participant>();

}

Содержит следующие данные

* + - идентификатор чата
    - название чата
    - описание чата
    - геолокация чата
    - радиус действия
    - идентификатор создателя чата
    - список идентификаторов администраторов чата
    - список подробной информации об участниках чата, используемый для отображения кратких имен участников, и рассылки им сообщений
* public class Participant

{

public string UserId { get; set; }

public string UserName { get; set; }

}

Содержит информацию об участнике чата

* Идентификатор участника чата
* Краткое имя участника чата
* public class MessagePostedSignal : Message

{

public string MessageId { get; set; }

public string UserId { get; set; }

public string ChartId { get; set; }

public DateTime Created { get; set; }

public string Message { get; set; }

}

Содержит информацию о только что написанном сообщении, пересылаемую, участникам чата

* 1. Блок работы с api – сервером

Разрабатываемый продукт реализован в виде клиент-серверного приложения. В котором все манипуляции с пользовательскими данными происходят на стороне сервера, а отображение происходит на стороне клиента. Для эффективного взаимодействия клиента и сервера, была выбрана библиотека Retrofit 2.0, разработанная сотрудниками компании Google

У Retrofit, по сравнению с другими средствами, можно выделить несколько основных преимуществ:

* Крайне удобный и простой интерфейс, который предоставляет полный функционал для выполнения любых запросов;
* Гибкая настройка — можно использовать любой клиент для выполнения запроса, любую библиотеку для разбора json и т.д.;
* Отсутствие необходимости самостоятельно выполнять парсинг json-а — эту работу выполняет библиотека Gson (и уже [не только Gson](https://github.com/square/retrofit/blob/master/CHANGELOG.md#version-200-beta1-2015-08-27));
* Удобная обработка результата и ошибок;
* Поддержка Rx, что тоже является немаловажным фактором сегодня.

И так, рассмотрим более подробно интерфейсы, реализуемые в приложении

* @POST(**"api/chart/create"**)  
  **fun** createChart(@Body model: CreateChartModel?) : Call<Void>

Создает новый чат в системе, и принимает один параметр типа CreateChartNodel

* @GET(**"api/chart/inlocation"**)  
  **fun** getInLocation(@Query(**"longitude"**)longitude: Double, @Query(**"latitude"**)latitude: Double) : Call<MutableList<ChartReadModelShort>>

Производит запрос к *блоку работы с данными по протоколу HTTP* для получения всех доступных чатов в зависимости от текущего местоположения

Принимаемые параметры

* + Долгота
  + Широта
* @GET(**"api/chart/{id}"**)  
  **fun** getChart(@Path(**"id"**) id: String) : Call<ChartReadModel>

Запрашивает информацию о чате по идентификатору чата у *блока работы с данными по протоколу HTTP*

* @POST(**"api/auth/register"**)  
  **fun** registerUser(@Body model: UserRegisterModel?) : Call<Void>

Делает запрос к *блоку работы с данными по протоколу HTTP* для регистрации нового пользователя

* @FormUrlEncoded  
  @POST(**"token"**)  
  **fun** login(@Field(**"username"**) username:String, @Field(**"password"**)password: String,  
   @Field(**"grant\_type"**)grant\_type: String = **"password"**) : Call<TokenModel>

Производит запрос к *блоку работы с данными по протоколу HTTP* для прохождения авторизациию и получения токена, для последуюей работы с нашим приложением

Сейчас рассмотрим типы данных, которые используются в данном блоке

* **data class** UserRegisterModel(**val email**: String, **val userName**: String, **val password**: String, **var confirmPassword**: String)

Содержит следующие данные

* + Емаил пользователя
  + Краткое имя пользователя
  + Пароль
  + Пароль для подтверждения
* **data class** CreateChartModel(**val title**: String, **val latitude**: Double, **val longitude**: Double, **val radius**: Double, **val description**: String?)

Содержит следующие данные

* + Имя чата
  + Широта
  + Долгота
  + Описание
  + Радиус действия
* **data class** TokenModel(**val token\_type**: String, **val access\_token**: String, **val expires\_in**: String)

Содержит следующие данные

* + тип токена, может иметь разные значение согласно спейификации oAuth
  + значение токена
  + время истечение срока действия токена
  1. Блок определяющий местоположение пользователей

Отвечает за постоянный мониторинг местоположения пользователя. Данный блок реализуется при помощи специальных сервисов Google Play. Поэтому для корректной работы разрабатываемого программного продукта на мобильном устройстве должно быть установлено приложение Google Play Services. Данные от этого сервиса приходит в виде текущей координаты пользователя, которая включает долготу, широту и высоту над уровнем моря. Данный блок является составной частью мобильного приложения.

Вкратце рассмотрим, как происходит определение местоположение с использованием сотовой связи

Включив ваш мобильный телефон или любое другое устройство, работающее в GSM-сети (Mobile station), вы запускаете огромный технологический процесс.

Первое, что делает ваше мобильное устройство (MS), это осуществляет поиск ближайших базовых станций (Base station, BS). Если быть точнее, то прослушивает эфир в поисках доступных сот. Телефон может прослушивать до 16 широковещательных каналов. Из их числа определяет 6, наиболее удовлетворяющих с точки зрения затрат энергии и качества сигнала. Но в один момент времени работать будет только с одной. У каждой из сот есть свой уникальный номер(CellID).

При этом все базовые станции (BS) объединены в группы. Как правило, принадлежность к группе определяется их местоположением (Location area), и чтобы базовые станции можно было идентифицировать, каждой группе присваивается уникальный номер — Location area code (LAC).

Вместе параметры LAC и CellID работают как уникальные идентификаторы базовой станции, на которой зарегистрировано и работает мобильное устройство.

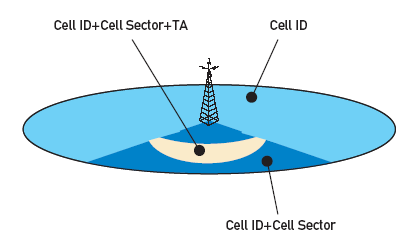


Рисунок 4.3 – Определение местопложения абонента

Именно используя эти параметры, Центр коммутации выбирает верное направление для отправки вызова в вашу сторону, иначе приходилось бы искать вас среди тысяч базовых станций.

Кроме того, определяется сектор базовой станции (Cell Sector) и фиксируется время, за которое сигнал от мобильного устройства достигает базовой станции – это параметр [Timing Advance](http://en.wikipedia.org/wiki/Timing_advance). Благодаря этому известна не только принадлежность к базовой станции, но и удаленность от нее.

Определение положения абонента в сотовой сети (см. рисунок 4.3). Данные о местоположении мобильного телефона обновляются с определенной периодичностью или в случае, если он перемещается (конечно же, вместе с вами), то при каждом переключении между базовыми станциями.

Взаимодействие систем для определения позиции мобильного устройства. Номер телефона, который хранится в огромной базе номеров сотового оператора.

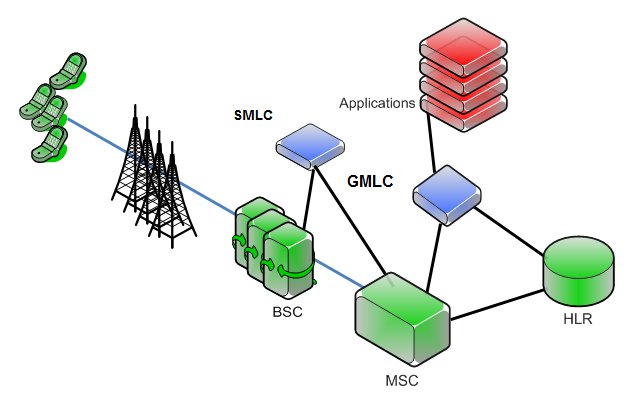
Как направить звонок нужному абоненту (см. рисунок 4.4)?   
  


Рисунок 4.4 – Направление звонка абоненту

Домашний регистр местоположения (Home location registry, HLR) содержит информацию обо всех абонентах, приписанных к нему. Найдя абонента в одной изHLR, мы перенаправляем вызов на связанный с ним Центр коммутации (Mobile switching centre, MSC), тот в свою очередь запрашивает информацию у Гостевого регистра местоположения (VLR), который содержит данные о последнем зарегистрированном CallID, LAC абонента. После чего контроллер базовых станций (Base station controller, BSC) связывается с базовыми станциями в рамках переданного LAC, и звонок направляется на нужную соту (CellID).

Как было указано ранее в нашем приложении используется специальный API предоставленной компанией Google для определения местоположения

**mApiClient** = GoogleApiClient.Builder(**this**)  
 .addConnectionCallbacks(**object** : GoogleApiClient.ConnectionCallbacks{  
 **override fun** onConnectionSuspended(p0: Int) { }  
  
 **override fun** onConnected(bundle: Bundle?) {  
 LocationServices.*FusedLocationApi*.requestLocationUpdates(  
 **mApiClient**, Config.**locationRequest**) **{** location **->** updateCharts(location)  
 **}**;  
 **try**{  
 **val** location = LocationServices.*FusedLocationApi*.getLastLocation(  
 **mApiClient**)  
 updateCharts(location)  
 }**catch**(ex: Exception){}  
 }  
 })  
 .addOnConnectionFailedListener **{ }** .addApi(LocationServices.*API*)  
 .build()

В данном методе происходит запрос каждые 3 секунды о текущем положении

LocationServices.*FusedLocationApi*.requestLocationUpdates

И далее вызывается метод

**fun** updateCharts(location: Location)  
Который принимает текущее местополжение пользователя и производит следующие действия:производит сравнение с предыдущим местоположением с погрешностью 1 метр, если местоположение изменилось, производится запрос на сервер к *блоку работы с данными по протоколу HTTP*

1. МЕТОДИКА ИСПЫТАНИЙ

Тестирование программного обеспечения – процесс исследования программного обеспечения с целью получения информации о качестве продукта. Тестирование является одним из важных этапов разработки, поскольку, при написании программного кода невозможно предусмотреть все случаи, и ошибки в работе программы неизбежны. Кроме того, возможно появление ошибок в случае внесения дополнительного функционала в программу даже в уже отлаженном рабочем коде.

Тестирование программы направлено на доказательство ее соответствия заявленным требованиям. Таким образом, основной целью тестирования является выделение несоответствия функционирования программы заявленным требованиям и устранение их, для повышения качества программного продукта.

Тестирование программы проводилось в два этапа:

* + - поэтапное тестирование модуля в процессе написания программного кода;
    - с использованием модульных тестов
    - полное тестирование программы после окончания процесса написания программного кода на нескольких мобильных устройствах

Эти этапы являются достаточно важными, ни один из них невозможно исключить. Например, без модульного тестирования, при анализе работы программы в целом, будет происходить достаточное количество сбоев, выявить и локализовать которые может оказаться достаточно сложным заданием, в то время, как при анализе работы одного модуля, неисправность оказывается достаточно очевидной. И обратный случай, работоспособность каждого компонента в отдельности не гарантирует корректное поведение всей программы в целом.

5.1 Модульное тестирование

Разберемся с некоторыми основными понятиями модульного тестировании.

Unit testing (юнит тестирование или модульное тестирование) — заключается в изолированной проверке каждого отдельного элемента путем запуска тестов в искусственной среде. Для этого необходимо использовать драйверы и заглушки. Поэлементное тестирование — первейшая возможность реализовать исходный код. Оценивая каждый элемент изолированно и подтверждая корректность его работы, точно установить проблему значительно проще чем, если бы элемент был частью системы.

Unit (Элемент) — наименьший компонент, который можно скомпилировать.

Драйверы — модули тестов, которые запускают тестируемый элемент.

Заглушки — заменяют недостающие компоненты, которые вызываются элементом и выполняют следующие действия:

* + - возвращаются к элементу, не выполняя никаких других действий;
    - отображают трассировочное сообщение и иногда предлагают тестеру продолжить тестирование;
    - возвращают постоянное значение или предлагают тестеру самому ввести возвращаемое значение;
    - осуществляют упрощенную реализацию недостающей компоненты;
    - Имитируют исключительные или аварийные условия.

White-box testing. Для конструирования тестов используются внутренняя структура кода и управляющая логика. При этом существует вероятность, что код будет проверяться так, как он был написан, а это не гарантирует корректность логики.

Black-box testing. Для конструирования тестов используются требования и спецификации ПО. Недостатки:

* + - таким способом невозможно найти взаимоуничтожающихся ошибок,
    - некоторые ошибки возникают достаточно редко (ошибки работы с памятью) и потому их трудно найти и воспроизвести

Для модульного тестирования на платформе .Net я использовал две основные библиотеки: NUnit и Moq. Так как тестирование приложений, написанный с помощью подхода CQRS не очень тривиальная задача. Рассмотрим более подробно реализацию тестирования данного проекта.

Так как вся логика приложения сосредоточена в агрегатах и обработчиков событий, то первым делом мы должны протестировать их.

И так, для начала создадим базовый класс для всех тестов агрегата

**namespace** mPower.Tests.Dnx.Environment

{

**public** **abstract** **class** **AggregateTest**<TAggregate>

where TAggregate : AggregateRoot

{

**public** **abstract** IEnumerable<IEvent> **Given**();

**public** **abstract** IEnumerable<ICommand> **When**();

**public** **abstract** IEnumerable<IEvent> **Expected**();

[SetUp]

**public** **void** **Prepare**()

{

\_id = Guid.**NewGuid**().**ToString**();

\_actualEvents = **new** List<IEvent>();

**PrepareEvents**();

var bus = GetInstance<IServiceBus>();

**foreach** (var command **in** **When**())

{

bus.**Send**(command);

}

\_actualEvents = ((InMemoryEventBus)\_eventBus).Events;

\_lastExceptions = bus.**GetLastException**();

}

**protected** **virtual** **void** **PrepareEvents**()

{

var store = GetInstance<ITransitionStorage>();

**using** (var stream = store.**OpenStream**(\_id))

{

var transitionEvents = **Given**().**Select**(e => **new** **TransitionEvent**("", e, **null**)).**ToList**();

stream.**Write**(**new** **Transition**(**new** **TransitionId**(\_id, 1), DateTime.Now, transitionEvents, **null**));

}

}

**public** **void** **Validate**(**params** **string**[] exclude)

{

var expectedEvents = **Expected**().**ToList**();

Assert.**AreEqual**(expectedEvents.Count, \_actualEvents.Count);

**for** (**int** i = 0; i < \_actualEvents.Count; i++)

{

var actual = \_actualEvents[i];

var expected = expectedEvents[i];

var excludeList = **new** List<**string**>(exclude);

excludeList.**Add**("Metadata");

excludeList.**Add**("NewBalanceOfAggregatedAccount");

var equal = ObjectComparer.**AreObjectsEqual**(expected, actual, IgnoreList.**Create**(excludeList.**ToArray**())); *// ignore property with Metadata name*

Assert.**IsTrue**(equal);

}

}

**public** **void** **DispatchEvents**(Action testReadModel, **bool** createNewDatabase = **true**)

{

**ApplyCreateNewDatabaseSetting**(createNewDatabase);

**try**

{

var currentEvents = **new** IEvent[\_actualEvents.Count];

\_actualEvents.**CopyTo**(currentEvents);

var events = **Given**().**Concat**(currentEvents);

**foreach** (var @event **in** events)

{

\_eventDispatcher.**Dispatch**(@event);

}

**if** (testReadModel != **null**)

**testReadModel**();

}

**finally**

{}}}}

И так, для того чтобы протестировать логику некого агрегата, нам во первых необходимо определить команды, которые послыются на агрегат. Делается это путем переопределения представленного метода в дочерних класса

**public** **abstract** IEnumerable<ICommand> **When**();

Далее нам неообходимо обозначить те события, которые бутут созданы при успешном выполнении логики аргреата в ответ на определеныые команды. Данной действие реализуется путем переопределения метода представленного метода в дочерних класса

**public** **abstract** IEnumerable<IEvent> **Expected**();

Также помимо этого, нам может потребовать некоторое состояние агрегата, для этого вам нужно определить список тех событий, которые должны быть выполнены до начала теста в методе

**public** **abstract** IEnumerable<IEvent> **Expected**();

Теперь, перед тем как тест агрегата начнет выполнятся, вызывается метод представленный ниже метод

**public** **void** **Prepare**()

Сущность его заключается в том, чтобы восстановить все события для агрегата, которые необходимы для прохождения тестов и вызов команд, для того чтобы сработала логика агрегатов и создались необходимые события.

Следующий шаг заключается в вызове метода

**public** **void** **Validate**(**params** **string**[] exclude)

Данный метод проверяет, все ли события создались в ответ на наши команды, которые мы хотели получить.

И на заключительном этапе вызывается метод

**public** **void** **DispatchEvents**(Action testReadModel, **bool** createNewDatabase = **true**)

Основная суть данного метода в проверки действий, которые происходят в обработчике команд, которые вызываются, при выполнении заданных правил в тестируемом агрегате.

Сейчас я приведу пример теста, который тестриует правильность действий по изменению пароля пользователя

**public** **class** **user\_change\_password\_test** : UserTest

{

**private** **const** **string** password = "asd123!";

**public** **override** IEnumerable<IEvent> **Given**()

{

**yield** **return** **User\_Created**();

}

**public** **override** IEnumerable<ICommand> **When**()

{

**yield** **return** **new** **User\_ChangePasswordCommand**()

{

PasswordHash = password,

ChangeDate = \_currentDate,

UserId = \_id

};

}

**public** **override** IEnumerable<IEvent> **Expected**()

{

**yield** **return** **new** **User\_PasswordChangedEvent**()

{

ChangeDate = \_currentDate,

NewPassword = password,

UserId = \_id

};

}

[Test]

**public** **void** **Test**()

{

**Validate**();

**DispatchEvents**(() =>

{

var user = \_userDocumentService.**GetById**(\_id);

Assert.**AreEqual**(user.Password, password);

Assert.**AreEqual**(user.LastPasswordChangedDate.Day, \_currentDate.Day);

Assert.**AreEqual**(user.LastPasswordChangedDate.Minute, \_currentDate.Minute);

Assert.**AreEqual**(user.LastPasswordChangedDate.Second, \_currentDate.Second);

Assert.**AreEqual**(user.ResetPasswordToken, **null**);

});

}

}

1. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

6. 1 Установка программного средства

Для того, чтобы запустить данные программный продукт, вы должны иметь смартфон на базе операционной системы Android с версией не ниже 4.0.

А также предустановленное приложение Google Play, которое позволит вам скачать разрабатываемый программный продукт. Так же помимо это вам необходимо установить Google Play Services. Данное приложение предоставляет Api для определения местоположения мобильного устройства.

И так первое что вам нужно сделать – это зайти в приложение Google Play и скачать приложение, введя в строке поиска Geofy.

6.2 Использование программного продукта

Для того, чтобы программа работала корректно, у вас на мобильном устройстве должен быть доступ в интернет и разрешено определение местоположения в настройках телефона.

При первом запуске приложение, вы увидите данное окно (см. рисунок 6.1)

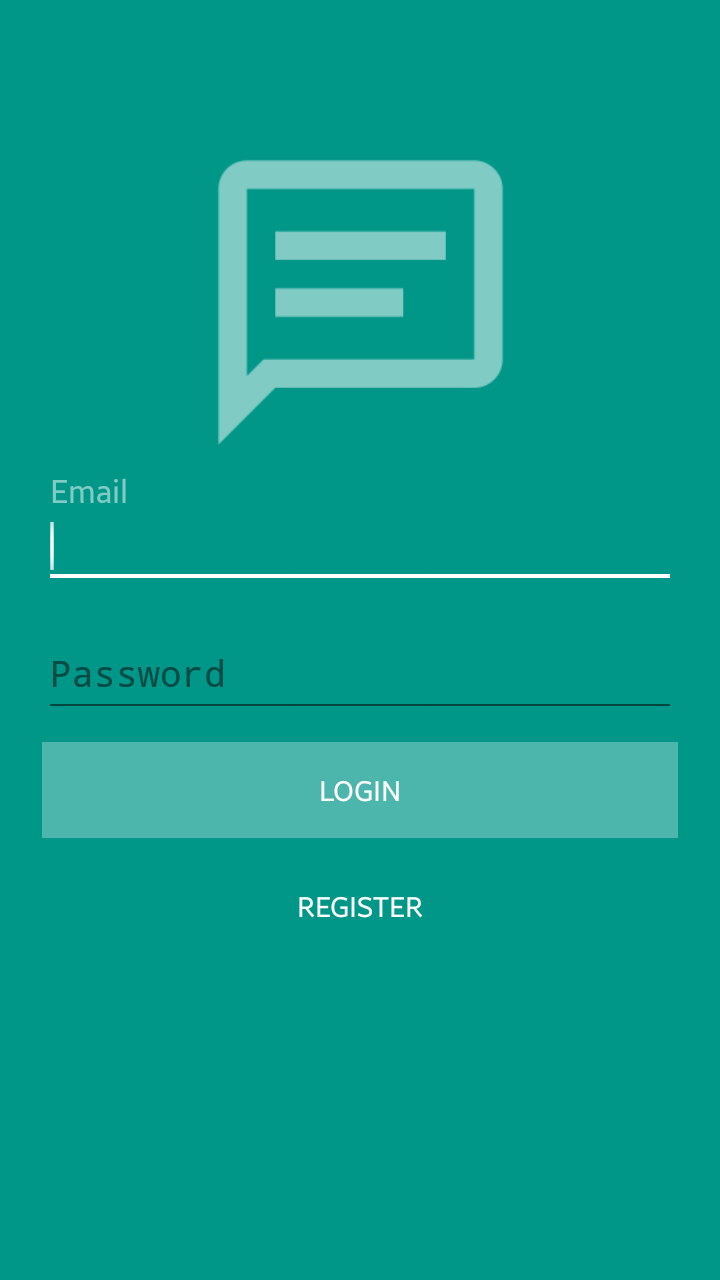


Рисунок 6.1 – Окно входа

Если вы уже зарегистрированный пользователь, вы можете ввести логин и пароль и перейти к главному окну программы. В противном же случае вам необходимо пройти пользовательскую регистрацию, нажав кнопку *Register.* И вы увидите следующие окно приложения (см. рисунок 6.2).

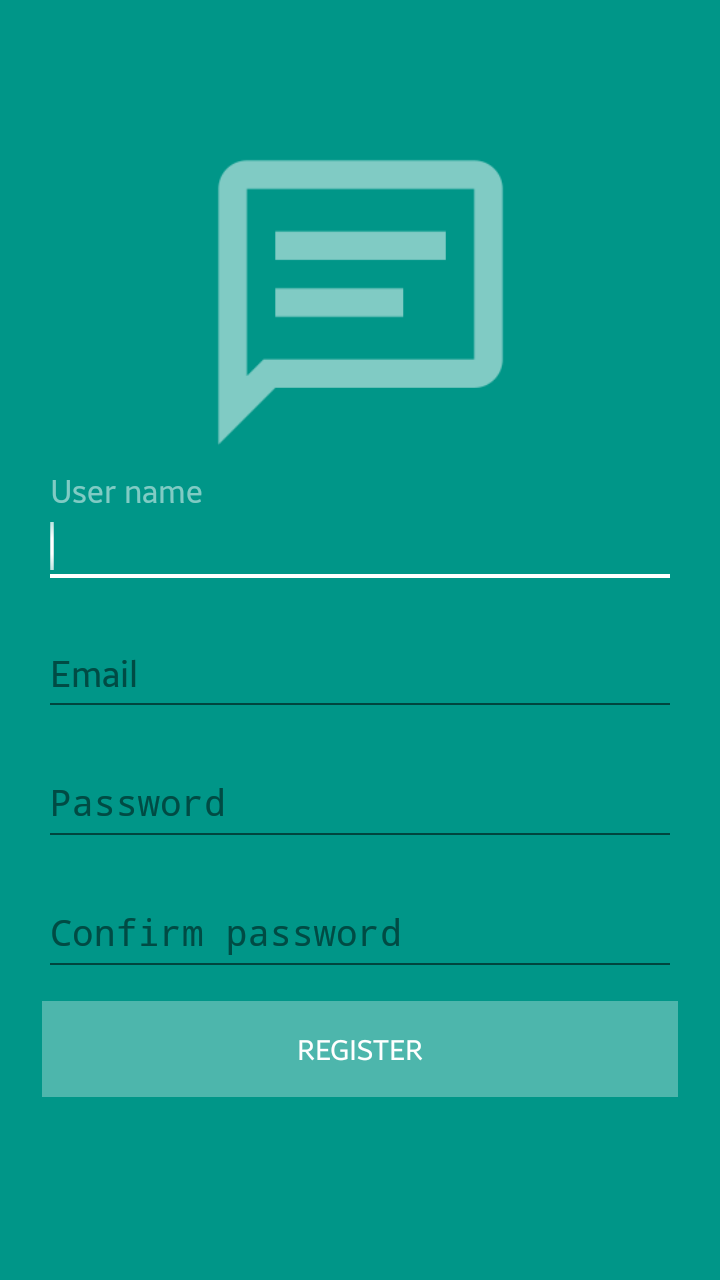


Рисунок 6.2 – Окно регистрации

Здесь вам необходимо предоставить ваши данные системе, а именно:

* краткое имя, которое будет отображаться под каждым вашим сообщением
* емаил, для идентификации пользователя
* пароль
* подтверждающий пароль, который должен совпадать с основным паролем

В случае же, если вы введете какие-то данные неправильно, или вообще забудете их ввести, системой будет показано уведомление (см. рисунок 6.3)

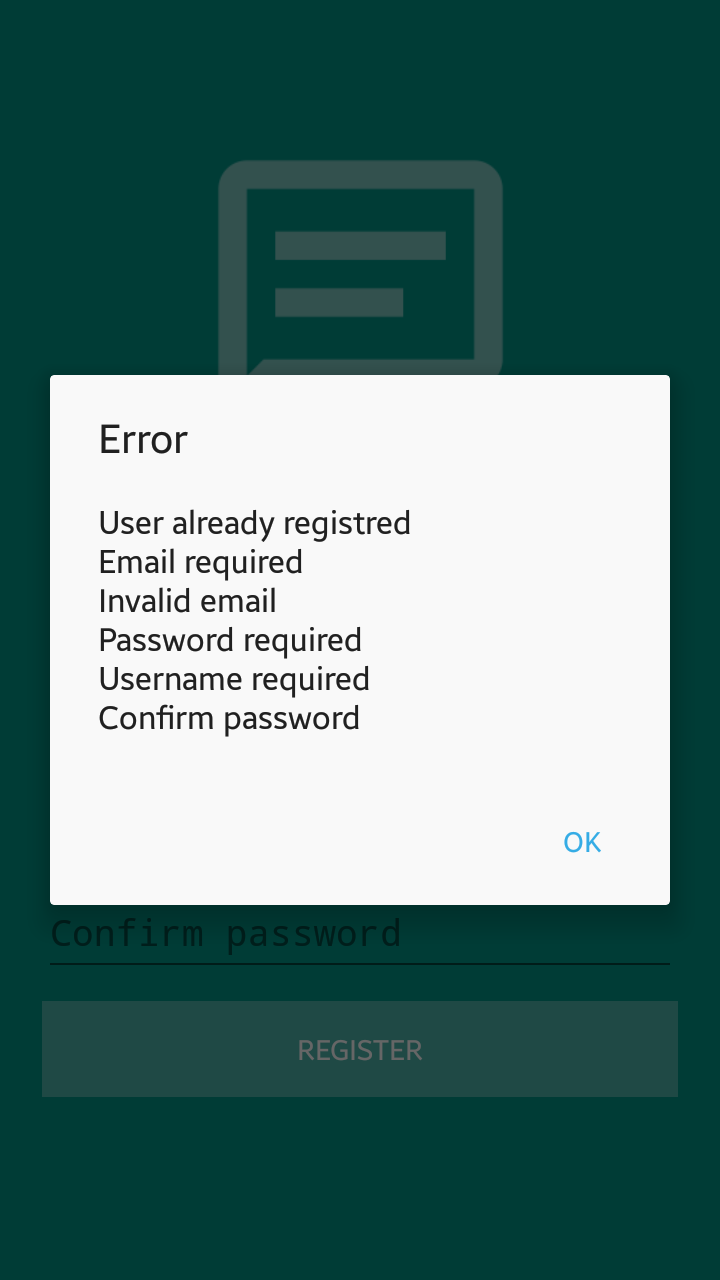


Рисунок 6.3 – Уведомление об ошибках

Если же регистрация произошла успешно, вы будете перенаправлены на главное окно приложения (см. рисунок 6.4).

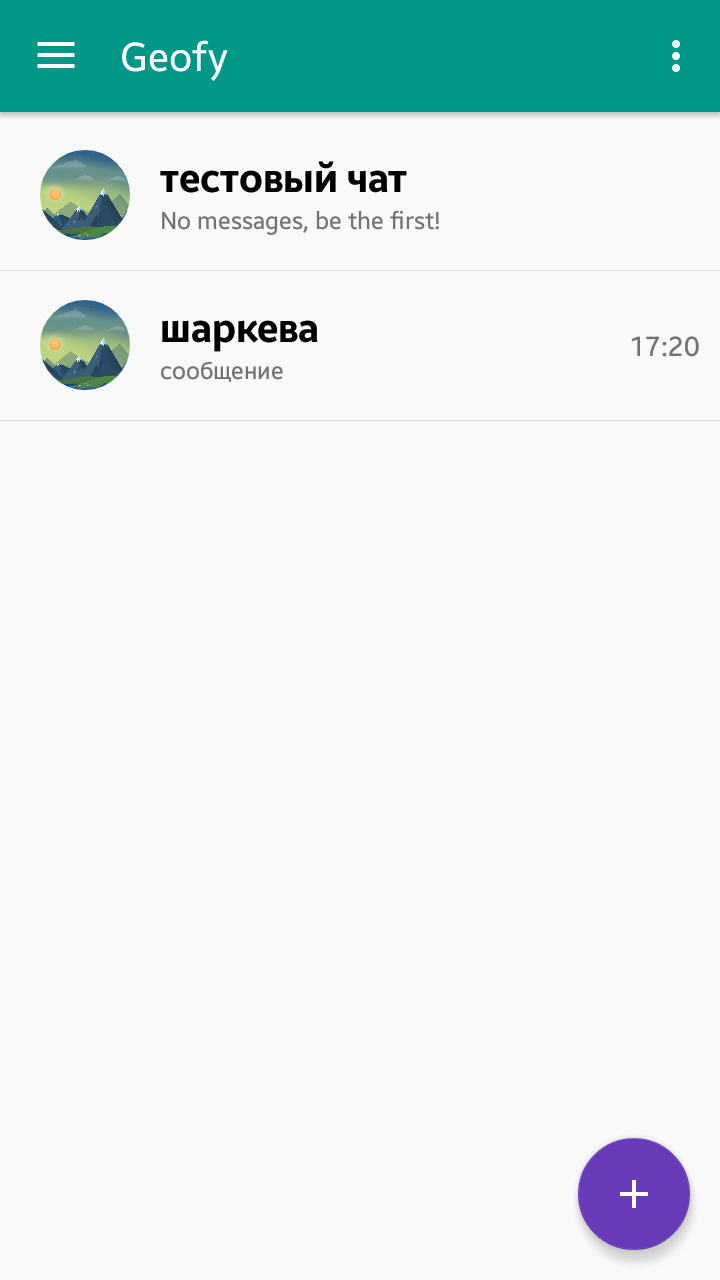


Рисунок 6.4 – Окно просмотра чатов

На данном окне вы можете видеть список всех доступных чатов в зависимости от вашего местоположения (вы находитесь в радиусе действия определенного чата). Как вы могли заметить для каждого чата отображается

* + - название
    - последнее написанное сообщение
    - время последнего написанного сообщения
    - логотип чата

Так же вы можете видеть в правом нижнем углу круглую кнопку, по нажатию на которую вы будете перенаправлены на окно создания чата (см. рисунок 6.5).

На данном окне вы можете видеть интерактивную карту, которая отображает ваше текущее местоположение и радиус действия чата.

Для создания чата вам необходимо ввести некоторые обязательные данные

* + - название чата
    - радиус действия чата (от 10 до 100м)

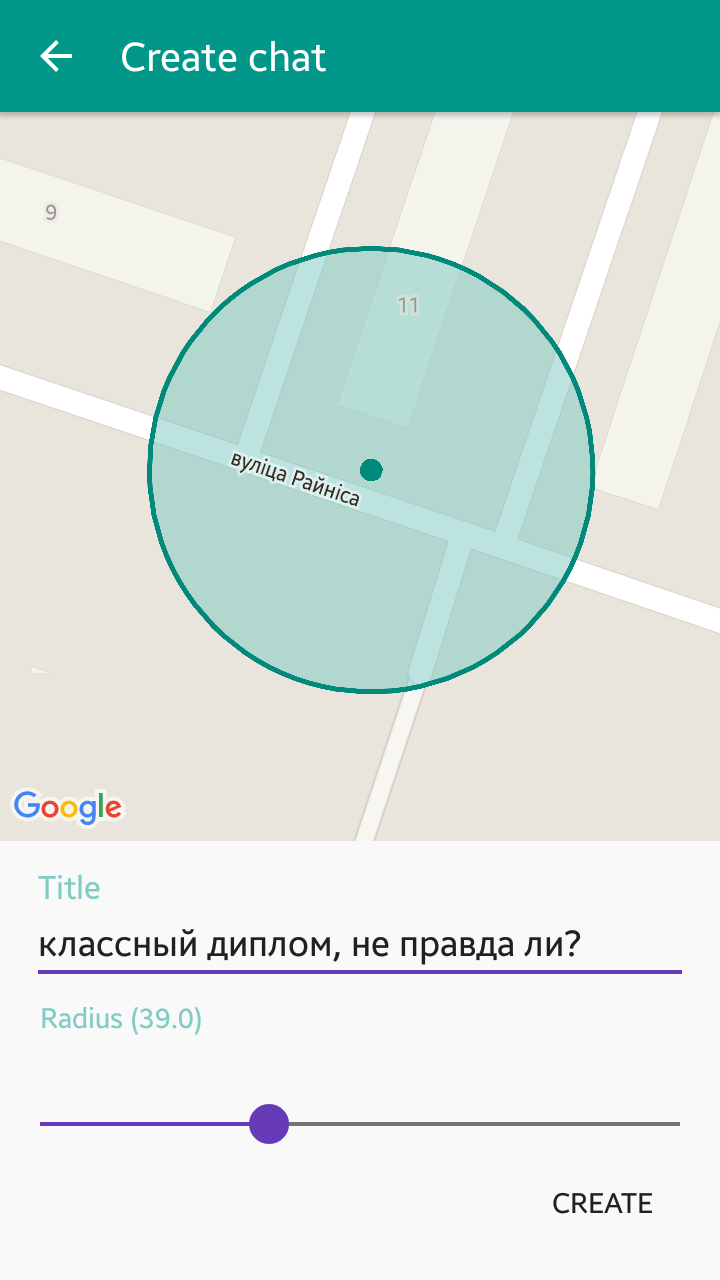


Рисунок 6.5 – Окно создания чатов

И нажать кнопку создать, после этого вы будете перенаправлены на окно чата (см. рисунок 6.6).

Однако, если вы ввели неправильные данные, система покажет вам уведомление (см. рисунок 6.7)

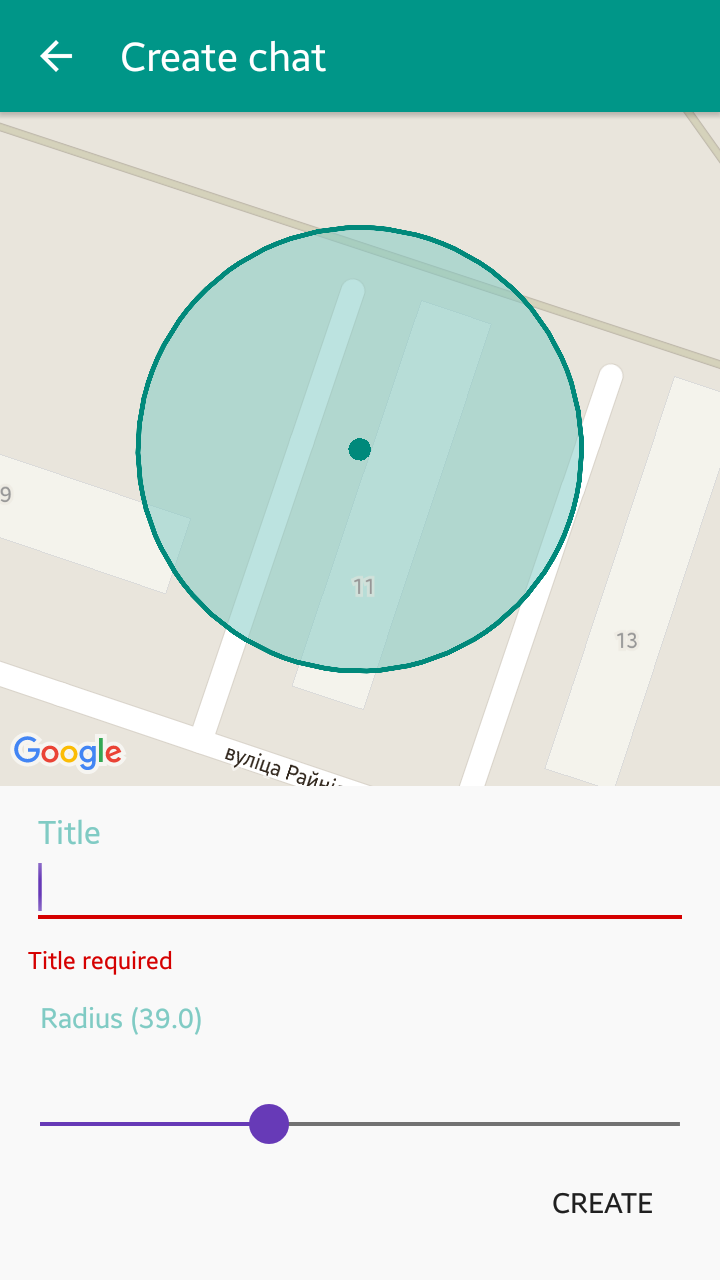


Рисунок 6.7 – Уведомление об ошибках

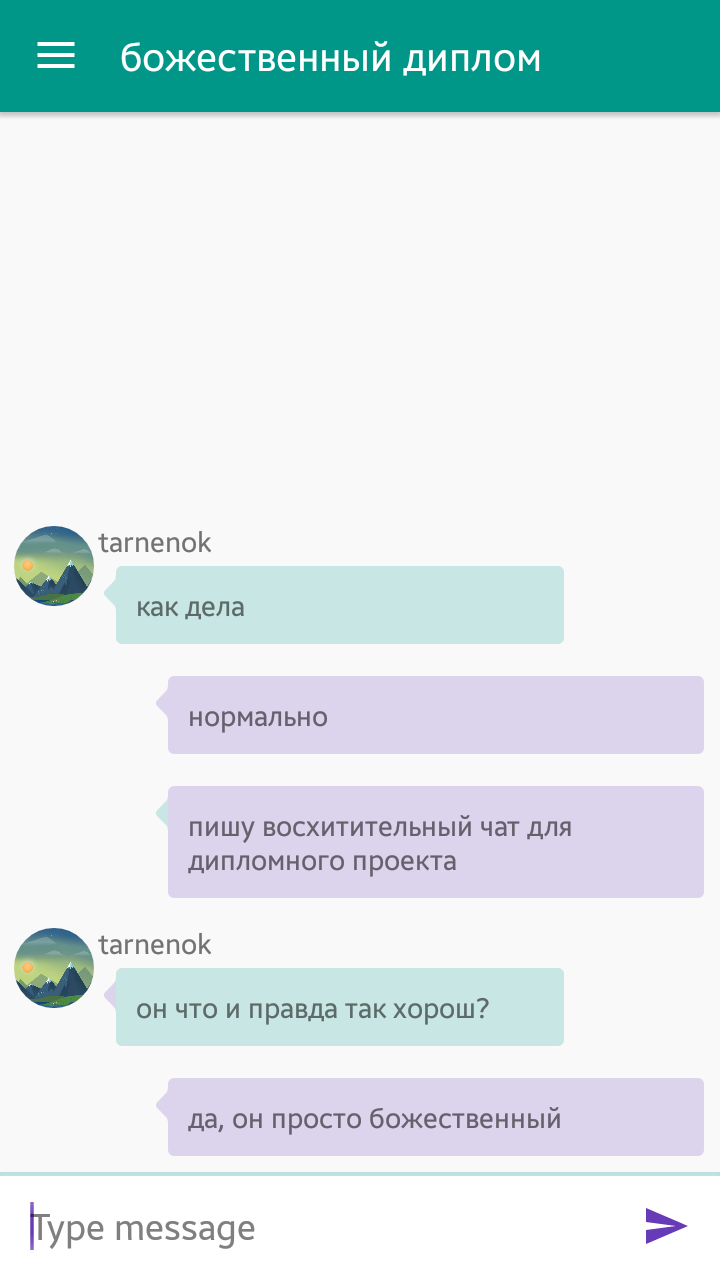


Рисунок 6.6 – Основное окно чата

В этом окне вы можете писать сообщения пользователям чата, а также просматривать другие сообщения. Для чужих сообщений, которые расположены слева отображается краткое имя пользователя и его фотография в системе.

Так же вы можете производить более детальные настройки: а именно устанавливать краткое имя для каждого чата свое. А также если вы являетесь админом этого чата (создатель чата по умолчанию является админом и может назначать админские права другим пользователям). Для более детальных настроек вы должны открыть меню, проведя пальцем по экрану с левой в правую сторону. И в появившемся меня выбрать настройки.

1. ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ЭФФЕКТИВНОСТИ РАЗРАБОТКИ И РЕАЛИЗАЦИИ ЧАТА, УЧИТЫВАЮЩЕГО ГЕОЛАКАЦИОННЫЕ ДАННЫЕ ПОЛЬЗОВАТЕЛЕЙ

7.1 Характеристика чата, учитывающего геолокационные данные пользователей

Чат, учитывающий геолокационные данные пользователей, представляет собой программный продукт в виде двух систем: мобильное приложение и веб сервер. Основное назначение данного программного продукта – сделать этот мир лучше, а также помочь людям упростить их коммуникации между собой.

Разработчиком приложения является компания ООО «Шаркева системс инкорпорайтед». Приложение рассчитано на массовую аудиторию, поэтому им могут пользоваться все, у кого есть смартфон.

Распространение мобильного приложения планируется осуществлять путем размещения его на специальной площадке продажи копий лицензий мобильных приложений для платформы Android – Google Play. Размещение на данной платформе является бесплатным, однако имеются операционные сборы с продажи одной копии лицензии в размере 30%.

Исходя из маркетингового исследования, лицензии на программный продукт будут востребованы на рынке в течение 4 лет; в 2016 году планируется продать 50000 лицензий, в 2017 – 120000 лицензий, в 2018 – 40000 лицензий, а в 2019 – 20000 лицензий.

Экономическая целесообразность инвестиций в разработку и реализацию программного продукта осуществляется на основе расчета и оценки следующих показателей:

* чистый дисконтированный доход (ЧДД);
* срок окупаемости инвестиций ()
* рентабельность инвестиций ().

7.2 Расчет затрат на разработку программного продукта

Затраты на основную заработную плату команды разработчиков () определяется исходя из состава и численности команды, размеров месячной заработной платы каждого из участников команды, а также общей трудоемкости разработки программного обеспечения.

Расчет величины основной заработной платы участников команды осуществляется по формуле:

где *n* – количество исполнителей, занятых разработкой ПП;

– часовая тарифная ставка 1-го разряда, руб.;

– эффективный фонд рабочего времени *i*-го исполнителя;

– количество часов работы в день, ч.;

К – коэффициент премирования (1,5).

В настоящий момент тарифная ставка первого разряда (на 15.05.2016) на предприятии составляет 1 млн. руб. Среднемесячная расчетная норма рабочего времени на 2016 год составляет 22 дня.

Разработкой программного продукта занимались три исполнителя: инженер-программист 13 разряда с тарифным коэффициентом 3,04, руководитель проекта, имеющий 19 разряд с тарифным коэффициентом 4,56 и дизайнер, имеющий 14 разряд с тарифным коэффициентом 3,35.

Расчет основной заработной платы исполнителя приведен в таблице 7.1.

Таблица 7.1 – Расчет основной заработной платы разработчика ПП

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Исполнитель | Разряд | Тарифный коэффициент | Месячная тарифная ставка, руб. | Дневная тарифная ставка, руб. | Плановый фонд рабочего времени, дн. | Заработная плата, руб. |
| Руководитель проекта | 19 | 4,56 | 4560000 | 207272 | 30 | 6218160 |
| Инженер-программист | 13 | 3,04 | 3040000 | 138181 | 60 | 8290860 |
| Дизайнер | 14 | 3,25 | 3250000 | 147727 | 30 | 4431810 |
| Основная заработная плата | 18940830 | | | | | |
| Премия, (50%) | 9470415 | | | | | |
| Основная заработная плата с учетом премии (), руб. | 28411245 | | | | | |

Затраты на дополнительную заработную плату команды разработчиков () включает выплаты, предусмотренные законодательством о труде (оплата отпусков, льготных часов, времени выполнения государственных обязанностей и других выплат, не связанных с основной деятельностью исполнителей), и определяется по формуле:

где – норматив дополнительной заработной платы (20%).

Дополнительная заработная плата исполнителей составляет:

Отчисления на социальные нужды (), в фонд социальной защиты населения и на обязательное страхование, определяется в соответствии с действующими законодательными актами по формуле:

где – норматив отчислений на социальные нужды (34+0,6 %).

Отчисления на социальные нужды составляют:

Расходы по статье «Машинное время» (), включающие оплату машинного времени, необходимого для разработки и отладки программного продукта, определяются по формуле:

где – цена одного машино-часа (20000 руб.);

– количество часов работы в день;

– длительность проекта.

Расходы по статье «Машинное время» составят:

Расходы по статье «Прочие затраты» (), включают затраты на приобретение и подготовку специальной научно-технической информации и специальной литературы и определяются по формуле:

где – норматив прочих затрат (5%).

Расходы по статье «Прочие затраты» составляют:

Затраты по статье «Накладные расходы» (), связанные с необходимостью содержания аппарата управления, вспомогательных хозяйств и опытных (экспериментальных) производств, а также с расходами на общехозяйственные нужны, рассчитываются по формуле:

где – норматив накладных расходов (70%).

Рассчитываем расходы по статье «Накладные расходы»:

Общая сумма затрат по всем статьям сметы () на программный продукт рассчитывается по формуле:

Рассчитаем сумму расходов по всем статьям сметы:

.

Кроме того, потребуются дальнейшие затраты на сопровождение и адаптацию ПП (), которые определяются по нормативу ():

где – норматив расходов на сопровождение и адаптацию (5%);

– смета расходов в целом по ПП без расходов на сопровождение и адаптацию, тыс. руб.

Общая сумма расходов на разработку (с затратами на сопровождение и адаптацию) как полная себестоимость ПП () определяется по формуле:

Рассчитываем общую сумму расходов на разработку:

7.3 Расчет экономического эффекта от продажи программного продукта

Экономический эффект для разработчика программного обеспечения заключается в получении прибыли от его продажи множеству потребителей. Прибыль от реализации напрямую зависит от объемов продаж, цены реализации и затрат на разработку данного программного продукта.

Распространение приложения планируется осуществлять через онлайн-магазин приложений Google – Google Play.

Исходя из маркетингового исследования, лицензии на программный продукт будут востребованы на рынке в течение 4 лет; в 2016 году планируется продать 50000 лицензий, в 2017 – 120000 лицензий, в 2018 – 40000 лицензий, а в 2019 – 20000 лицензий.

На основании маркетингового исследования отпускная цена одной копии лицензии составила 2000 руб.

Прибыль от продажи одной лицензии программного продукта определяется по формуле:

где Ц – отпускная цена одной копии лицензии ПП;

НДС – сумма налога на добавленную стоимость;

– сумма налогового сбора компании Google с реализации одной копии ПП;

– сумма расходов на разработку и реализацию;

N – количество лицензий ПП, которое купят клиенты.

Сумма налога на добавленную стоимость рассчитывается по формуле:

где Ц – отпускная цена одной копии лицензии ПП;

– ставка налога на добавленную стоимость (20%).

Рассчитаем сумму налога на добавленную стоимость:

Сумма налогового сбора компании Google с реализации одной копии ПП рассчитывается по формуле:

где Ц – отпускная цена одной копии лицензии ПП;

– ставка налогового сбора магазина Google Play с реализации одной копии ПП (30%).

Рассчитаем сумму налогового сбора:

Так как дополнительные затраты на реализацию не требуются (затраты на размещение на торговой площадке компании Google и реклама уже включены в налоговый сбор компании с реализации одной копии ПП), то сумма расходов на разработку и реализацию рассчитывается по формуле:

Рассчитаем прибыль от продажи одной лицензии ПП по формуле (4.10):

Чистая прибыль от продажи одной лицензии ПП рассчитывается по формуле:

где – ставка налога на прибыль (18%).

Подставив данные в формулу (4.14) получаем чистую прибыль от продажи одной лицензии ПП:

Суммарная чистая годовая прибыль по проекту в целом рассчитывается по формуле:

Прибыль по проекту за каждый код продаж составит:

* 1. Расчет показателей эффективности разработки программного продукта

Для проведения сравнительного анализа размера суммы затрат на разработку ПП и получаемого экономического эффекта необходимо привести их к единому моменту времени – началу расчетного периода, что обеспечит их сопоставимость. Для этого необходимо использовать дисконтирование путем умножения соответствующих результатов и затрат на коэффициент дисконтирования () соответствующего года *t*, который определяется по формуле:

где – норматив приведения разновременных затрат и результатов (нормативная ставка дисконта), в долях единицы в год;

– расчетный год, = 1;

– порядковый номер года.

На 15.05.2016 г. ставка рефинансирования составляет 22%. Используя формулу (4.16) рассчитаем коэффициенты дисконтирования:

Расчет показателей эффективности инвестиций по разработке продукта представлен в таблице 7.2.

Таблица 7.2 – Расчет эффективности инвестиционного проекта по разработке программного продукта.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Показатель | Единицы измерения | Расчетный период | | | |
| 2016 г. | 2017 г. | 2018 г. | 2019 г. |
| РЕЗУЛЬТАТ | | | | | |
| 1 Экономический эффект | руб. |  |  |  |  |
| Коэффициент дисконтирования | доли ед. | 1 | 0,82 | 0,67 | 0,55 |
| 2 Дисконтированный результат | руб. |  | 68978400 | 18786800 | 7711000 |
|  | | | | | |
| 3 Затраты на разработку программного средства | руб. |  |  |  |  |
| 4 Дисконтированные инвестиции | руб. |  |  |  |  |
| ЭКОНОМИЧЕСКИЙ ЭФФЕКТ | | | | | |
| 5 Чистый дисконтированный доход по годам | руб. | -45591341 | 68978400 | 18786800 | 7711000 |
| 6 Чистый дисконтированный доход нарастающим итогом | руб. | -45591341 | 23387059 | 42173859 | 49884859 |

Рентабельность инвестиций () рассчитывается по формуле:

где – среднегодовая величина чистой прибыли за расчетный период.

Среднегодовая величина чистой прибыли за расчетный период определяется по формуле:

где –величина чистой прибыли за *i*-ый расчетный год;

*n* – расчетное количество лет.

Среднегодовая величина чистой прибыли и рентабельность инвестиций соответственно составят:

В ходе технико-экономического обоснования были рассчитаны следующие показатели:

* чистый дисконтированный доход (ЧДД) за четыре года составил 49884859 руб.;
* окупаемость инвестиций () наступает на второй год продаж;
* рентабельность инвестиций () по проекту составляет %.

Исходя из полученных данных можно сделать вывод, что разработанный проект является эффективным и затраты, связанные с разработкой программного продукта экономически целесообразны.

ЗАКЛЮЧЕНИЕ

В данном дипломном проекте был рассмотрел вопрос построения приложения с использованием принципа разработки программного обеспечения CQRS. В рамках дипломного проекта была разработана библиотека, позволяющая программистам облегчить жизнь по реализации принципов CQRS в их разрабатываемых приложениях, а также мобильное приложение «Чат, учитывающий геолокационные данные пользователей»

В ходе реализации данного программного продукта были, были изучены основные недостатки и достоинства принципов CQRS, также изучены принципы реактивного и асинхронного программирования. Рассмотрены основные недостатки построения моделей данных программного обеспечения с использованием Event Sourcing. Получены практические знание в работе с нерелеционными базами данными, изучены их недостатки, такие как: сложность работы с многосвязными данными. А также основные достоинства как: большие возможности для горизонтального масштабирования, высока скорость работы с большим объемом данных и нечеткую схему данных. Так же были рассмотрены основные способы определения местоположения мобильных устройств и способов работы с этими данными.

В результате цель дипломного проекта была достигнута. Было создано программное обеспечение, а также библиотека, позволяющая программистам облегчить жизнь по реализации принципов CQRS. Однако за рамками рассматриваемой темы осталось еще много проблем, связных с изучением принципов CQRS. Одна из которых это увеличение скорости работы регенерации Моделей данных используемых в приложении на основе произошедших событий.

В дальнейшем планируется развивать и довести существующее ПО до полноценной библиотеки, способной решать более широкий класс задач, возникающих в области применения принципов CQRS, а также расширение функционала мобильно приложения. А также перенос мобильного клиента программного продукта на другие платформы: iOS и Windows Phone.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Web Service Activity [Электронный ресурс] / W3C. – Режим доступа: https://www.w3.org/2002/ws/.
2. Филиппов, В. А. Информационные взаимодействия и Web-сервисы / В. А. Филиппов. – М.: Ленанд, 2013. – 144 с.
3. Петин, В. А. API Яндекс, Google и других популярных веб-сервисов / В. А. Петин. – СПб.: БХВ-Петербург, 2014. – 480 с.
4. [Martin Fowler](http://martinfowler.com/) – CQRS [Электронные ресурс] – Режим доступа: <http://martinfowler.com/bliki/CQRS.html>
5. [Command and Query Responsibility Segregation (CQRS) на практике](http://blog.byndyu.ru/2014/07/command-and-query-responsibility.html) [Электронный ресурс] – Режим доступа: <http://blog.byndyu.ru/2014/07/command-and-query-responsibility.html>
6. Введение в CQRS + Event Sourcing [Электронный ресурс] – Режим доступа: <https://habrahabr.ru/post/146429/>
7. Nosql [Электронный ресурс] ­­– Режим доступа: <https://ru.wikipedia.org/wiki/NoSQL>
8. NoSQL базы данных: понимаем суть [Электронный ресурс] – Режим доступа: <https://habrahabr.ru/post/152477/>
9. C# and .NET MongoDB Driver [Электронный ресурс] – Режим доступа: <https://docs.mongodb.com/ecosystem/drivers/csharp/>
10. Seguin, Karl. The Little MongoDB Book [Электронный ресурс] – Режим доступа: <http://openmymind.net/mongodb.pdf>
11. [Kotlin Programming Language](https://kotlinlang.org/) [Электронный ресурс] – Режем доступа: <https://kotlinlang.org/>
12. [Tutorials - Kotlin Programming Language](https://kotlinlang.org/docs/tutorials/) [Электронный ресурс] – Режим доступа: <https://kotlinlang.org/docs/tutorials/>
13. Asynchronous Programming with Async and Await [Электронный ресурс] – Режим доступа: https://msdn.microsoft.com/en-us/library/hh191443.aspx
14. [Davies](http://shop.oreilly.com/product/0636920026532.do#tab_04_2), Alex. Async in C# 5.0 / [Alex Davies](http://shop.oreilly.com/product/0636920026532.do#tab_04_2). ­­– 1fst edition ­­– O’Reilly Media, Inc, 2012. ­– September. – 102p
15. Макконнелл, С. Совершенный код. Мастер-класс / Пер. с англ. / С. Макконнелл. — СПб.: Издательско-торговый дом «Русская редакция», 2005. — 896 с.
16. Рихтер, Джеффри. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#/ Джеффри Рихтер. — 4е изд. — СПб.: Питер, Русская Редакция, 2015. — 896 с.
17. Richter, Jeffrey. CLR via C# / Jeffrey Richter. Microsoft, Developer Reference. — 4-th edition. — One Microsoft Way, Redmond, Washington 98052-6399 : Microsoft Press, 2012. — 896 P.
18. Albahari, Joseph. C# 5.0 in a Nutshell / Joseph Albahari, Ben Albahari. — 5-th edition. — O’Reilly Media, Inc, 2012. — June. — 1062 P.
19. [Коматинени](http://oz.by/people/more9066247.html), C. Android 4 для профессионалов. Создание приложений для планшетных компьютеров и смартфонов/ Пер. с англ. /[С. Коматинени](http://oz.by/people/more9066247.html), [Д. Маклин](http://oz.by/people/more9066248.html) – 4е изд. – СПб.: Издательский дом «Вильямс», 2013. – 880
20. Android developer’s portal [Электронный ресурс] ­– Режим доступа: <https://developer.android.com/develop/index.html>
21. Anko [Электронный ресурс] ­– Режим доступа: <https://github.com/Kotlin/anko/blob/master/README.md>
22. Разработка real-time ASP.NET приложений с помощью SignalR [Электронный ресурс – Режим доступа]: http://merle-amber.blogspot.com.by/2012/11/real-time-aspnet-signalr.html
23. Палицын, В. А. Технико-экономическое обоснование дипломных проектов: Метод. пособие для студ. всех спец. БГУИР. В 4-х ч. Ч. 4: Проекты программного обеспечения / В. А. Палицын. — Минск: БГУИР, 2006. — 76 с
24. Календарь праздников на 2016 год для Беларуси [Электронный ресурс]. — Электронные данные. — Режим доступа: http://calendar.by/2016/#bkm — Дата доступа: 05.05.2016.
25. Ставка рефинансирования | Национальный банк Республики Беларусь [Электронный ресурс]. — Электронные данные. — Режим доступа: http://www.nbrb.by/statistics/MonetaryPolicyInstruments/RefinancingRate/. — Дата доступа: 05.05.2016.