МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

учреждение образования

«Гродненский государственный университет имени Янки Купалы»

Факультет математики и информатики

**Кафедра современных технологий программирования**

СТЕЛЬМАШЕНКО АНДРЕЙ НИКОЛАЕВИЧ

**Разработка приложения для управления процессом организации публичных событий**

Курсовая работа

студента 4 курса специальности

1-40 01 01 «Программное обеспечение информационных технологий»

дневной формы получения образования

|  |  |
| --- | --- |
|  |  |
|  |  |

Гродно, 2016

**РЕЗЮМЕ**

Стельмашенко Андрей Николаевич

Курсовая работа – «Разработка приложения для управления процессом организации публичных событий», 33 страницы, 15 иллюстраций, 1 таблица, 2 листинга, 5 использованных источника.

Ключевые слова – событие, лента событие, архитектура, паттерн проектирования, сервис, пользовательский интерфейс, серверная часть, хранение данных.

Цель исследования – изучить способы хранения и передачи данных, протоколы аутентификации, способы защиты веб-приложений от несанкционированного доступа, а также современные подходы к реализации пользовательского интерфейса.

Объект исследования – паттерны объектно-ориентированного проектирования, протоколы аутентификации, реляционные базы данных, Web API.

Предмет исследования – Microsoft SQL Server, React, Redux, OAuth, Google API.

Методы исследования – теоретический анализ, изучение литературы, наблюдение, эксперимент.

Курсовая работа представляет собой реализацию приложения с многоуровневой архитектурой. Процесс проектирования и программирования каждого уровня предоставляет возможности для глубокого и всестороннего изучения различных современных технологий программирования и проектирования. Сложность работы обеспечивается использованием различных технологий HTML, CSS, JavaScript, C#, комбинированное использование которых позволяет создать завершённый программный продукт. Благодаря тому, что приложение имеет несколько уровней, оно является хорошей площадкой для совершенствования навыков программирования в условиях, в которых сквозная функциональность должна быть минимизирована для обеспечения меньшей связности компонент.

**SUMMARY**

Course work – “Development of applications for managing the process of organizing public events” is research in the field of computer science. Relational databases are subject for scientific investigation. The most important part of investigation is research of database structure and access to data from application. Main tools for developing are: HTML, CSS, JavaScript, C#, Linq2DB, AutoMapper. Application architecture analyzes with usage of Resharper architecture tools.

**СОДЕРЖАНИЕ**

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ, УСЛОВНЫХ ОБОЗНАЧЕНИЙ 6](#_Toc469661296)

[ВВЕДЕНИЕ 7](#_Toc469661297)

[ГЛАВА 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 9](#_Toc469661298)

[1.1 Основные аспекты 9](#_Toc469661299)

[1.2 Обзор существующих решений 9](#_Toc469661300)

[ГЛАВА 2 ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ 12](#_Toc469661301)

[2.1 Объектная модель данных 12](#_Toc469661302)

[2.2 Физическая модель базы данных. 14](#_Toc469661303)

[2.3 Доступ к данным. 15](#_Toc469661304)

[2.4 Клиентское приложение 23](#_Toc469661305)

[2.5 Процесс аутентификации 25](#_Toc469661306)

[ГЛАВА 3 РЕАЛИЗАЦИЯ 26](#_Toc469661307)

[3.1 Доступ к данным. 26](#_Toc469661308)

[3.2 Защита данных. 29](#_Toc469661309)

[3.3 Реализация ленты событий. 31](#_Toc469661310)

[3.3 CRUD-операции 32](#_Toc469661311)

[3.4 Процесс регистрации 34](#_Toc469661312)

[ЗАКЛЮЧЕНИЕ 35](#_Toc469661313)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 36](#_Toc469661314)

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ, УСЛОВНЫХ ОБОЗНАЧЕНИЙ

|  |  |  |
| --- | --- | --- |
| API | Application Programming Interface | набор готовых классов, процедур, функций и структур, предоставляемых библиотекой или сервисом для использования во внешних программных продуктах. |
| CSS | Cascading Style Sheets | формальный язык описания внешнего вида документа, написанного с использованием языка разметки. |
| HTML | HyperText Markup Language | язык для структурирования и представления содержимого всемирной паутины. |
| REST | Representational state transfer | стиль архитектуры программного обеспечения для распределенных систем, как правило, используется для построения веб-служб. |
| СУБД | Система управления базами данных | совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных. |

**ВВЕДЕНИЕ**

В настоящее время в эпоху быстро развивающихся интернет технологий всё большее влияние на жизнь современного человека оказывает компьютеризации. В связи с этим в сфере организации мероприятий происходят существенные изменения направленные на автоматизацию процессов управления событиями, публикацию информации о них, сбор данных об участниках и их отзывах о мероприятиях.

Крупные компании, обладающие большим бюджетом могут позволить себе заказать услугу сопровождения своего мероприятия у сторонней организации, которая позаботится о приглашении гостей, сборе информации, продаже билетов, рекламной акции, размещении информации о событии на специализированных ресурсах. Однако есть большое число компаний, чья бизнес-модель предполагает проведение различны, часто бесплатных мероприятий – конференции, митапы, организация вечеринок с рекламной целью. При этом у данных компаний зачастую отсутствует достаточный объём средств на оплату комплексных услуг сторонних компаний.

Перспективным направлением видится создание сервиса, обеспечивающего автоматизированное сопровождение мероприятий. При этом, данный сервис должен быть оптимизирован не для управления событием, а для привлечения участников событий, так как это есть основная цель организаторов события. Основные функции приложения помимо непостредственного управления событием (создание, изменение, удаление, публикация): фильтрация, поиск, регистрация различных типов (буду участвовать, возможно буду участвовать, не интересует), сбор отзывов, отслеживание информации об отменённых регистрация, интеграция с календарями (Google, Outlook).

Цель дипломной работы – разработать программное обеспечение для предоставления услуг в сфере организации публичных событий.

В рамках данной курсовой работы предусмотрено решение следующих задач:

1. Создание базы данных приложения.
2. Создание гибкой архитектуры приложения, которая в дальнейшем позволит встраивать новые функциональные модули с меньшими трудозатратами.
3. Реализация процесса аутентификация.
4. Реализация прототипа клиентского приложения с базовыми функциями: создание события, регистрация на событие, написание отзыва о событие.

ГЛАВА 1  
АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 **Основные** аспекты

При реализации приложения основное внимание стоит уделить удобству использования приложения на мобильных устройствах. Также необходимо предусмотреть процесс аутентификации пользователя через социальные сети, так как большинство из них при отсутствии такой функциональности незамедлительно покинут приложение. Интеграция с календарями, социальными сетями и адаптивный дизайн дадут приложению возможность выделиться среди ряда существующих решений.

1.2 Обзор существующих решений

Eventbrite. Основная задача ресурса сопровождение крупных событий. Обладает возможность организации платных событий, поддерживает функционал по продаже электронных билетов и их валидацию.

Преимущества:

1. Возможность организации платных событий.
2. Продажа билетов.
3. Система категорий.
4. Поиск по геолокации.

Недостатки:

1. Обязательная регистрация (нет логина через аккаунт в социальной сети).
2. Нет синхронизации с календарями.
3. Нельзя оставить отзыв.
4. Нет возможности создать статус «Возможно пойду»

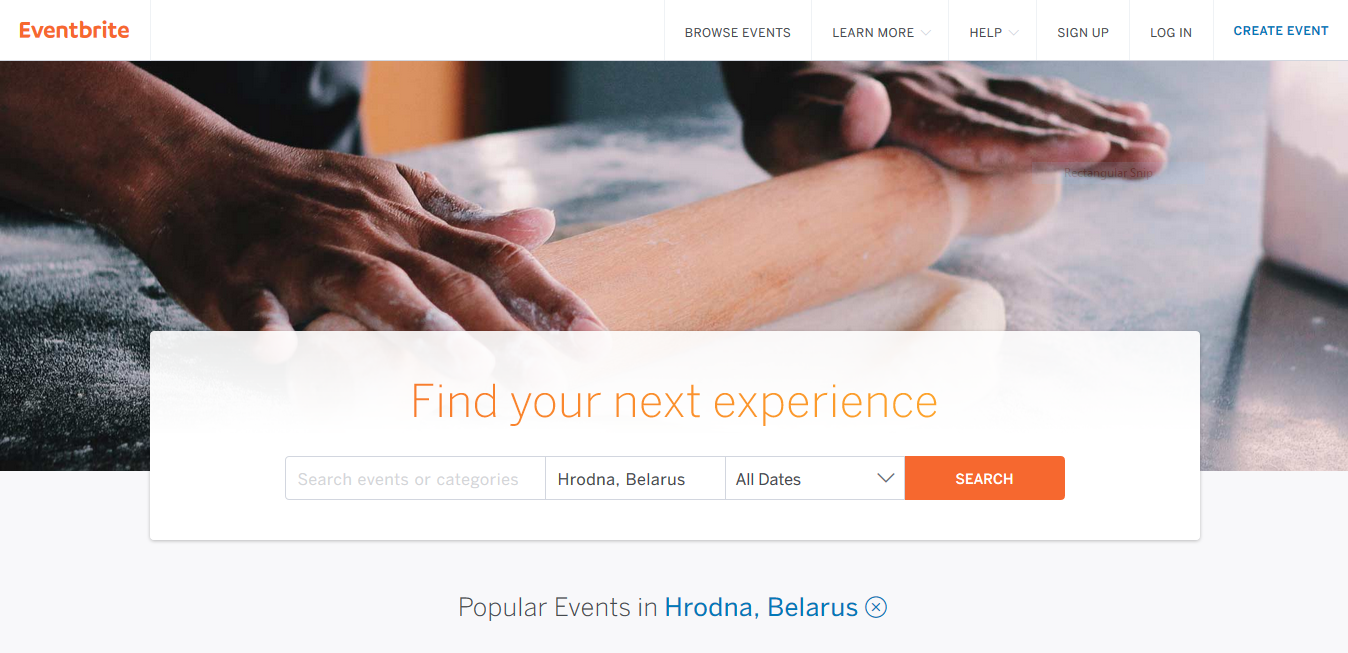


Рисунок 1.1 Eventbrite

TicketLeap – сервис, основной функцией которого является продажа билетов, не предоставляет площадку для публикации информации о событиях.

Преимущества:

1. Хорошо защищённые билеты.
2. Использования социальных сетей для входа в систему.
3. Присутствует встроенный календарь.

Недостатки:

1. Не предоставляет площадку для публикации информации о событиях.
2. Не позволяет собирать отзывы.
3. Нет синхронизации с календарями.

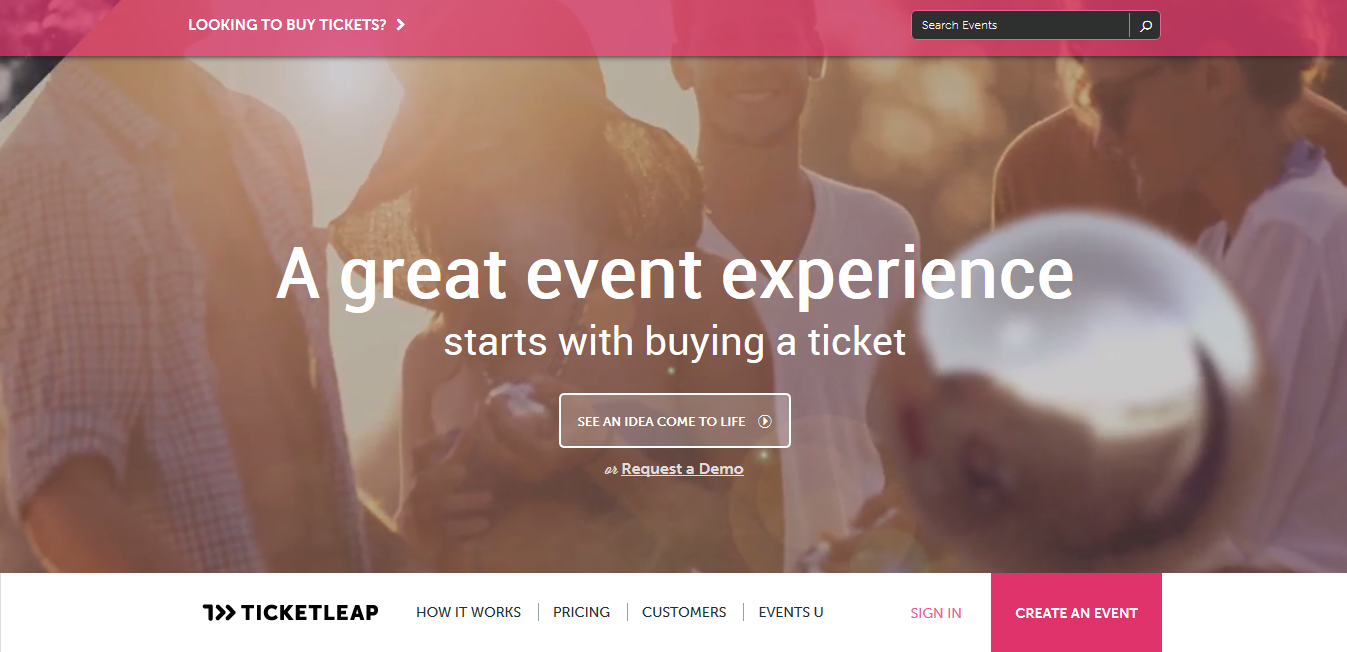


Рисунок 1.2 TicketLeap

1.3 Анализ эмоциональной окраски текста

Инструменты анализа семантической окраски текста полезны в маркетинге и управлении репутацией [10], то их эффективность часто оценивается на сборниках обзоров, которые обычно содержат мнения людей, выраженные на естественном языке, часто вместе с соответствующей (числовой) оценкой, определяющей суждение. Наиболее часто используемым набором данных для оценки качества алгоритмов является сборник из двух тысяч отзывов о фильмах на английском языке, которые были разделены экспертами в области психологии на 3 группы: негативные, нейтральные и позитивные[25]. Широкое распространение получил подход под названием «Пакет слов». Основная идея которого заключается в поиске специфических слов, которые наиболее сильно влияют на эмоциональную окраску текста. Эффективность этого подхода на упомянутой ранее коллекции отзывов составляет 87.2%. На современном этапе развития технологий в данной области исследования фокусировались на различных векторных представлениях текста, включая векторные представления с дополнительными признаками, представляющими семантические различия между словами [36] или векторные представления с весами на основе tf-idf [24]. Такие подходы обычно дают точность более 90% на наборе данных одной предметной области для которой изначально был настроен алгоритм, при на произвольных наборах данных качество оценки значительно падает. Альтернативные подходы, основанные на словарях, как правило, демонстрируют более низкую точность в наборе данных одной предметной области, но, как правило, более надежны во всех на наборах, составленных из случайно отобранных текстов [34]. Кроме того, основанные на словарях подходы могут быть легко обобщены относительно других языков с помощью словарей [19]. Было показано, что довольно простая структура анализа настроений на основе лексики имеет точность до 59,5% на полном наборе данных обзора фильмов [10]. Более сложный подход к анализу настроений на основе анализа словосочетаний имеет среднюю точность 68.0% на 1900 документов из данных обзора фильмов [33]. Приблизительно 5-ти процентный прирост точности по сравнению с анализом словосочетаний дают алгоритмы, анализирующие структуру текста: наличие сложных предложений, вопросительных, восклицательных, размер предложений. Несмотря на то, что современные подходы к анализу настроений на основе словарей рассматривают новые перспективные направления: анализ структурных и семантических аспектов контента, они обычно не получают информацию из потенциально важных маркеров настроения, которые широко используются в сегодняшнем пользовательском контенте. В качестве таких маркеров выступают смайлы (эмодзи, эмотиконы). Отличительной особенностью смайлов является то, что их эмоциональная окраска не зависит от предметной области, к которой принадлежит текст. Смайлы являются важными ориентирами для оценки настроений в современно пользовательском контенте. Ключ к сбору информации из смайликов заключается в понимании того, как они соотносятся с общим настроением текста. В публичных источниках не было найдено исследований, в которых анализировалось совместное использование лексически словарей и словарей смайлов для анализа стилистической окраски текста. Одной из целей данной дипломной работы является разработка подобного алгоритма и применение его для анализа отзывов о мероприятиях.

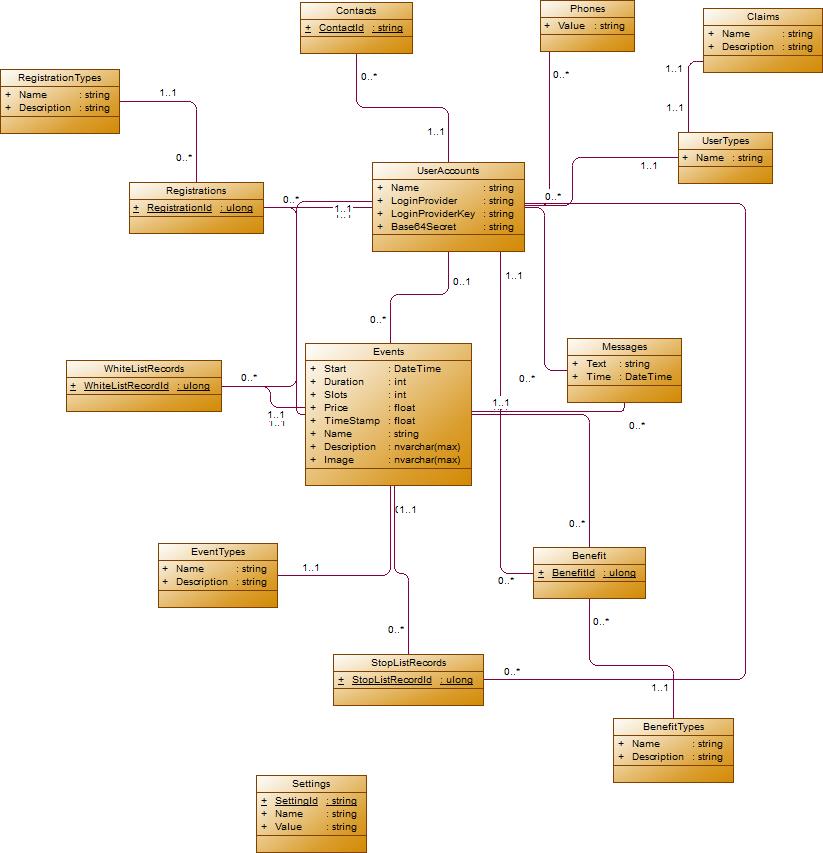
ГЛАВА 2  
ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

2.1 Объектная модель данных

В результате анализа предметной области была сформирована система система объектов, состоящая из следующих сущностей:

1. Событие (Event) – представляет собой модель, которая содержит информацию о событии и является основной сущностью системы.
2. Тип события (EventType) – сущность, предоставляющая общую информацию о событиях одного типа (отношение один ко многим).
3. Пользовательский аккаунт (UserAccount) – модель содержащая данные для идентификации пользователя в системе.
4. Тип пользователя (UserType) – описывает функционал доступный труппе пользователей (отношение один ко многим).
5. Право (Claim) – сущность, описывающая право выполнения операции конкретным типом пользователя (отношение многие ко многим).
6. Сообщение (Message) – комментарий пользователя к определённому событию (отношение многие к одному).
7. Контакт (Contact) – сущность, описывающая право одного пользователя получить информацию о другом пользователе.
8. Чёрный список (StopList) – массив записей (StopListRecord), вида событие-аккаунт, которые ограничивают возможность регистрации пользователя при попадании пользователя в это список.
9. Чёрный список (WhiteList) – массив записей (WhiteListRecord), вида событие-аккаунт, которые дают право на регистрацию если по умолчанию такая функция недоступна.
10. Настройка (Setting) – именованная сущность описывающая параметры конфигурации системы (не относится к предметной области).

Объектная модель данных представлена на рис 2.1.



**Рис. 2.1 Объектная модель данных**

2.3 Доступ к данным.

Для большинства современны web-приложений основной функцией является предоставления доступа к данным. Интерфейс типичного приложения водится к предоставлению пользователю следующих типичных процедур для манипулирования данными: добавление (create), чтение (read), обновление (update), удаление (delete). Данный набор операций известен как CRUD-операции. Данная концепция используется не только серверами управления баз данных (СУБД), но и основным протоколом передачи данных HTTP. В таблице 1 представлены соответствия между CRUD-операциями, операторами языка SQL, типами HTTP запросов (HTTP-verbs).

Таблица 1.

|  |  |  |
| --- | --- | --- |
| Операция | Оператор языка SQL | Операция в протоколе HTTP |
| Создание (create) | INSERT | POST |
| Чтение (read) | SELECT | GET |
| Редактирование (update) | UPDATE | PUT или PATCH |
| Удаление (delete) | DELETE | DELETE |

Схожий подход используется также в нерялиционных базах (документо-ориентированные СУБД, графовых СУБД и т.д.), при этом в серверах баз данных использующихся в высоконагруженных системах широкое распространение получил подход к увеличению производительности и отказоустойчивости основанный на отказе от предоставления операции изменения. Для имитации данной операции используется комбинация трёх оставшихся команд: чтение, удаление, вставка. При в отличии от традиционной операции изменения такой подход изменяет первичный ключ (индекс в таблице) и временную метку. Такой подход используется наиболее распространённом NoSQL (not only SQL) решении в СУБД Cassandra. Данная СУБД используется во множестве высоко нагруженных систем, наиболее известной из которых является сервис Twitter, который отображает информацию опубликованную пользователем в хронологическом порядке. Трёхуровневая архитектура предполагает наличие независимого слоя доступа к данным. При этом его детали реализации должны быть скрыты от уровня сервисов. К деталям реализации относятся то, каким образом осуществляется подключение к СУБД (тип ORM фреймворка), какой язык запросов используется (SQL, Transact-SQL, CQL, Pig), какие промежуточные модели данных используются и как реализовано кеширование. Архитектура слоя данных должна выполнять ещё одну немаловажную задачу – сокрытие текущих абстракций (leaky abstraction).

Текущая абстракция – термин в разработке программного обеспечения, описывающий абстракцию, которая предоставляет подробную информацию о деталях внутренней реализации и её ограничениях, которые в идеале должны быть спрятаны. Наличие утечек в абстракциях является серьёзной проблемой, поскольку цель абстракций заключается в управлении сложностью, скрывая ненужные детали от пользователя. Иногда для определения этого термина используют понятия дырявая абстракция, излучающая абстракция.

Закон текущих абстракций гласит: все нетривиальные абстракции являются текущими. Из этого следует, что в попытках построить большую систему с хорошими абстракциями, мы всё равно не сможем скрыть все детали реализации вытекающие из абстракции. Лучшее, что мы можем сделать в данной ситуации это локализовать место, в котором нам приходится вдаваться в детали реализации, которые не скрыты абстракцией.

Язык SQL был создан, чтобы абстрагироваться от процедурных шагов, нужных для запросов к базе данных. Вместо этого он позволяет описать, что именно запрашивается, и пусть база данных сама догадается, какие процедурные шаги для этого нужны. Но в иных случаях некоторые запросы SQL на порядки медленнее, чем другие, логически им эквивалентные. Известный пример: некоторые сервера SQL значительно быстрее отрабатывают запрос where a=b and b=c and a=c, чем where a=b and b=c, хотя результат, конечно, тот же самый. Программисту на SQL вроде бы и не следует заботиться о процедуре, только о спецификациях. Но иногда абстракция протекает, что приводит к значительным, а иногда и критическим потерям в производительности, так что приходится лезть во внутренности планировщика запросов и смотреть, что там не так, и как заставить его работать эффективнее. Эта проблема не решается и ORM фреймворками, поэтому задачей слоя данных является инкапсуляция этой и ей подобных утечек в абстракции.

2.4 Клиентское приложение

Архитектура клиентского приложения должна быть построена таким образом, чтобы обеспечить наиболее удобный подход к написанию бизнес-логики и при этом уменьшить количество технического кода не связанного с предметной областью, к такому коду относится функционал по манипуляции с DOM-деревом, отслеживание состояния приложение, низкоуровневый контроль за выполнением асинхронных запросов.

У любого приложения есть некоторое состояние, от которого зависит набор и вид отображаемых компонентов. Состояние может быть двух типов: изменяемое и неизменяемое. С первым типом всё понятно: состояние компонента изменяется и в ответ на это изменяется DOM-дерево. Второй же подход к работе с состоянием является неочевидным. Возникает вопрос: как изменяется вид компонентов если состояние неизменно? Вид компонента изменяется из-за того, что состояние становится новым, то есть изменяется не значение, которое хранится по ссылке, а сама ссылка. Этот подход более эффективен, так как проще отслеживать изменения. Если мы имеем дело с изменением значений переменных, то чтобы отследить изменения необходимо иметь в памяти копию всего состояния и сравнивать каждое значение. Если отслеживать только изменение ссылок, то в памяти достаточно хранить только набор ссылок в древовидном виде, что уменьшает количество объектов для сравнения и делает сложность поиска изменений логарифмической, поэтому в процессе разработки будет использован второй подход.

Центром всего приложения является пользовательский интерфейс (View), с которым взаимодействует пользователь. Пользовательский интерфейс формируется на основе данных, размещённых в контейнере состояния. Взаимодействие пользователя с интерфейсом приводит в действие Dispatcher, который формирует объект Action, содержащий информацию о действии пользователя. Обработчик событий (Reducer), генерирует новое состояние либо возвращает старое неизменённое состояние в зависимости от алгоритма обработки события и предыдущего состояния. Новое состояние попадает в хранилище, тем самым завершает циклическое движение данных.



**Рис. 2.7 Поток данных в клиентском приложении**

С точки зрения файловой структуры приложения целесообразно использовать фрактальную структуру. Основной идеей фрактальной структуры является группировка файлов в директории не по типу файлов, а по содержащейся в них функциональности. Малые приложения могут быть построены с использованием плоской структуры каталогов, с директориями для компонентов, контейнеров и т.д. Однако эта структура не масштабируется и может серьезно повлиять на скорость разработки по мере роста проекта. Изначально разрабатываемое с фрактальной структурой приложение может органично управлять своей архитектурой с самого начала процесса разработки. Эта структура обеспечивает множество преимуществ, которые могут быть не сразу очевидны:

1. Маршруты можно разделять в блоки (chunks) используя алгоритмы разделения и слияния кода. Это означает, что полное дерево зависимостей для каждого маршрута может быть опущено из исходного пакета и затем загружено по требованию.
2. Поскольку логика является автономной, маршруты можно легко разбить на отдельные репозитории и осуществить привязку приложению как отдельной компоненты для гибкой, высокопроизводительной разработки и масштабирования.

Фрактальная структура состоит из трёх основных компонентов:

1. Ствол – один для всего приложения, является маршрутизатором.
2. Ветвь – маршрутный узел приложения.
3. Лист – сборка, состоящая из компонентов представления, логики по работе с данными и контейнеров.

Ствол приложение содержит функциональность для поддержки фрактальной архитектуры:

1. Модуль динамического подключения обработчиков событий reducers.js
2. Модуль инициализации хранилища данных createStore.js
3. Основная страница приложения index.js
4. Модуль загрузки компонентов приложения main.js
5. Базовый контейнер приложения AppContainer.js
6. Набор общих для всех маршрутов компонент в директории /components
7. Макет приложения в директории /CoreLayout, который представляет собой компонент без состояния.

Помимо вышеперечисленного в приложении присутствуют директории со статическим контентом, файлы конфигурации, сервер, который осуществляет загрузку приложения, эти части приложения являются общими для большинства известных структур приложений.

Ветви приложения с фрактальной архитектурой являются системообразующими элементами. Их задачей является организация маршрутизации приложения. Каждая ветвь содержит не менее одного листа, а также может содержать одну или несколько ветвей. При этом для ветви B с маршрутом /route-b которая является дочерней для ветви A с маршрутом /route-a полный маршрут примет вид /route-a/route-b.

Пользовательский интерфейс приложения формируется листьями. Логика их размещения такова, что чаще лист используется на различных маршрутах, тем ближе к стволу дерева он будет находиться. Т.е. если лист отображается на страницах с маршрутами A2 и B2, которые являются дочерними для страницы с маршрутом A1, то он будет расположен в ветви соответствующей странице A1.

2.5 Процесс аутентификации

Аутентификация основана протоколе OAuth 2.0. Данный протокол предусматривает наличие четырёх ролей:

1. Владелец ресурса (Resource Owner) – сущность или пользователь получающий доступ к защищённым ресурсам.
2. Ресурсный сервер (Resource Server) – сервер, который предоставляет доступ к защищённым ресурсам. Данный сервер должен иметь возможность проверять токен выданный сервером авторизации и если токен корректный – отвечать на запрос клиентского приложения.
3. Клиентское приложение (англ. Client Applications) – приложение получающее данные от защищённого ресурсного сервера, данное приложение также запрашивает токен у сервера авторизации, который идентифицирует владельца ресурса.
4. Сервер авторизации (Authorization Server) – сервер, который управляет авторизацией выдавая токены доступа, позволяющие идентифицировать владельца ресурса при запросе данных с ресурсного сервера.



**Рис. 2.8 Схема авторизации**

* 1. Процесс оценки эмоциональной окраски текста.

Наиболее распространённым методом оценки эмоциональной окраски текстового сообщение является метод с использованием словарей, в которых каждому слову поставлено в соответствие численное значение (как правило от -1 до 1), которое описывает его окраску, чем меньше значение – тем более негативная эмоциональная окраска у слова, чем выше значение – тем более положительная окраска. Слова, имеющие модуль коэффициента равный нулю или близкий к нему, считаются нейтральными. У данного подхода есть ряд существенных проблем:

1. Необходимо проводить лингвистический анализ текста и приводить слова к начальной форме: число, время, в некоторых языках род и падеж.
2. Прямой перевод словаря не эффективен из-за неоднозначности перевода: одно и тоже слово может быть переведено разными способами с разной эмоциональной окраской.
3. Не учитывает порядок слов в предложении.

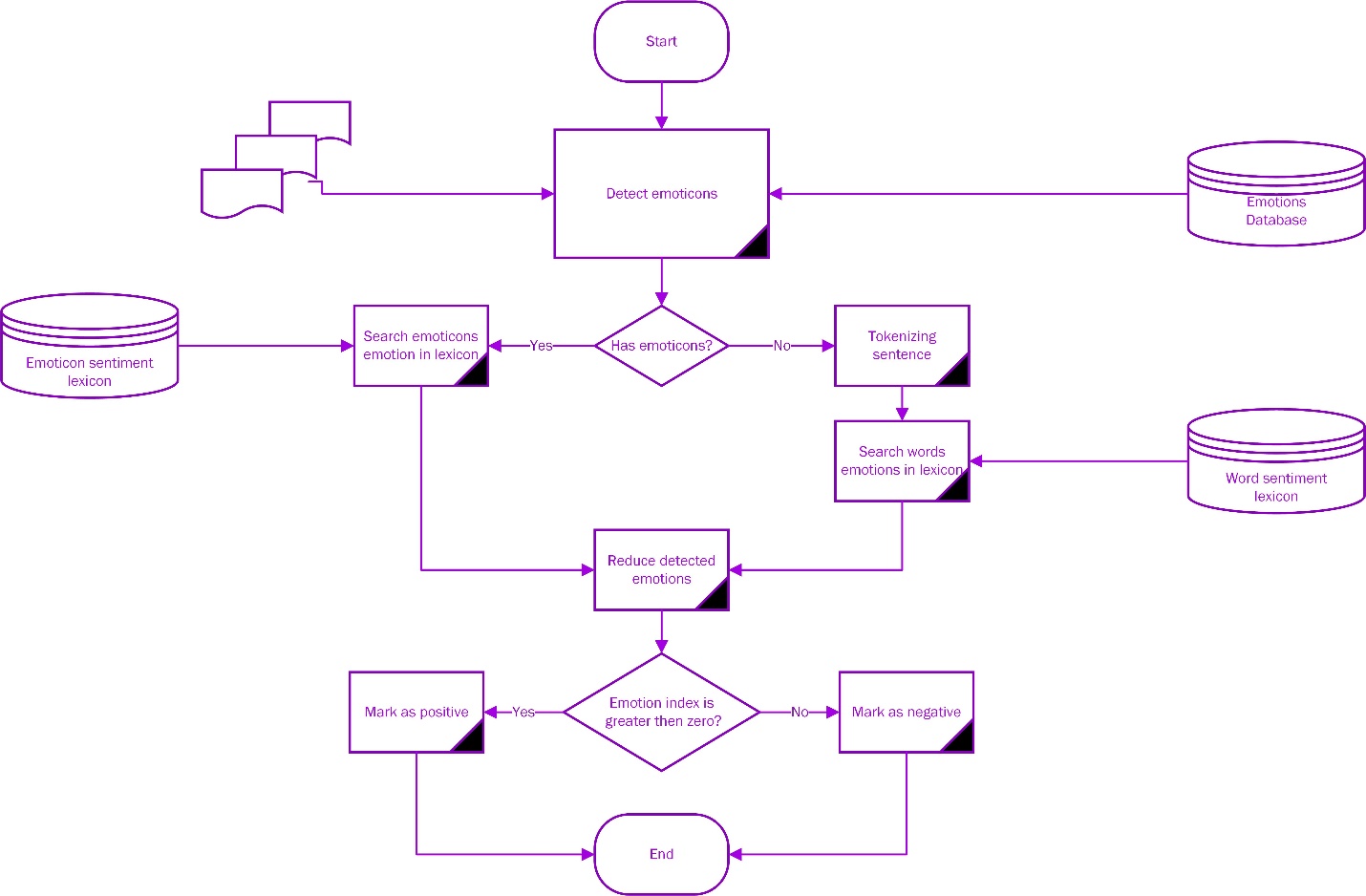
Избавиться от всех выше перечисленных проблем не представляется возможным, однако, учитывая специфику контента, генерируемого пользователем в сети. При написании отзывов, твитов, коротких сообщений других текстов направленных на выражение эмоций пользователи широко используют специализированые средства выражения эмоций – смайлы или их эквиваленты: эмодзи, эмотиконы и так далее. В любом случае все эти графические элементы реализуются с помощью символов доступных с клавиатуры: различные скобки, знаки препинания, буквы цифры, при их набор является достаточно ограниченным. Самые большие словари содержат до нескольких сотен смайлов, при этом в широком обращении находится не более двух десятков наиболее известных смайлов. Важным преимуществом смайлов является независимость их эмоциональной окраски от языка. Что касается обработки данных, то использования словарей эмоциональной окраски твита не уменьшает алгоритмическую сложность операции, однако заметно ускорят работу алгоритма за счёт того, что словарь смайлов имеет на несколько порядков меньшее количество элементов чем обычный словарь. Таким образом оптимальным видится следующий алгоритм оценки эмоциональной окраски сообщения:

1. Определение наличие смайлов в сообщении.
2. а. Поиск обнаруженных смайлов словаре и определение их эмоциональной окраски.

б. Разбиение предложения на слова и оценка окраски каждого слова по словарю.

1. Объединение полученных результатов поиска по словарю.
2. Определение класса сообщения по результатам метрики полученной в пункте 3.

На рисунке 2.1 приведена блок схема описанного алгоритма.

****

ГЛАВА 3  
РЕАЛИЗАЦИЯ

3.1 Доступ к данным.

Для упрощения доступа к данным используется микро-ORM фреймворк Linq2DB. В сравнении с наиболее популярным решением такого рода в среде .Net разработчиков Entity Framework-ом Linq2DB обладает рядом преимуществ:

1. Высокая скорость выполнения запросов.
2. Быстрая инициализация контекста.
3. Не навязывает свою архитектуру.

Данные преимущества обусловлены тем, что в отличие от Entity Framework-а Linq2DB не поддерживает функцию отслеживания изменений, которая является достаточно ресурсоёмкой. Так как при реализации текущего приложения данная функциональность не требуется, то выбор Linq2DB является оптимальным решением.

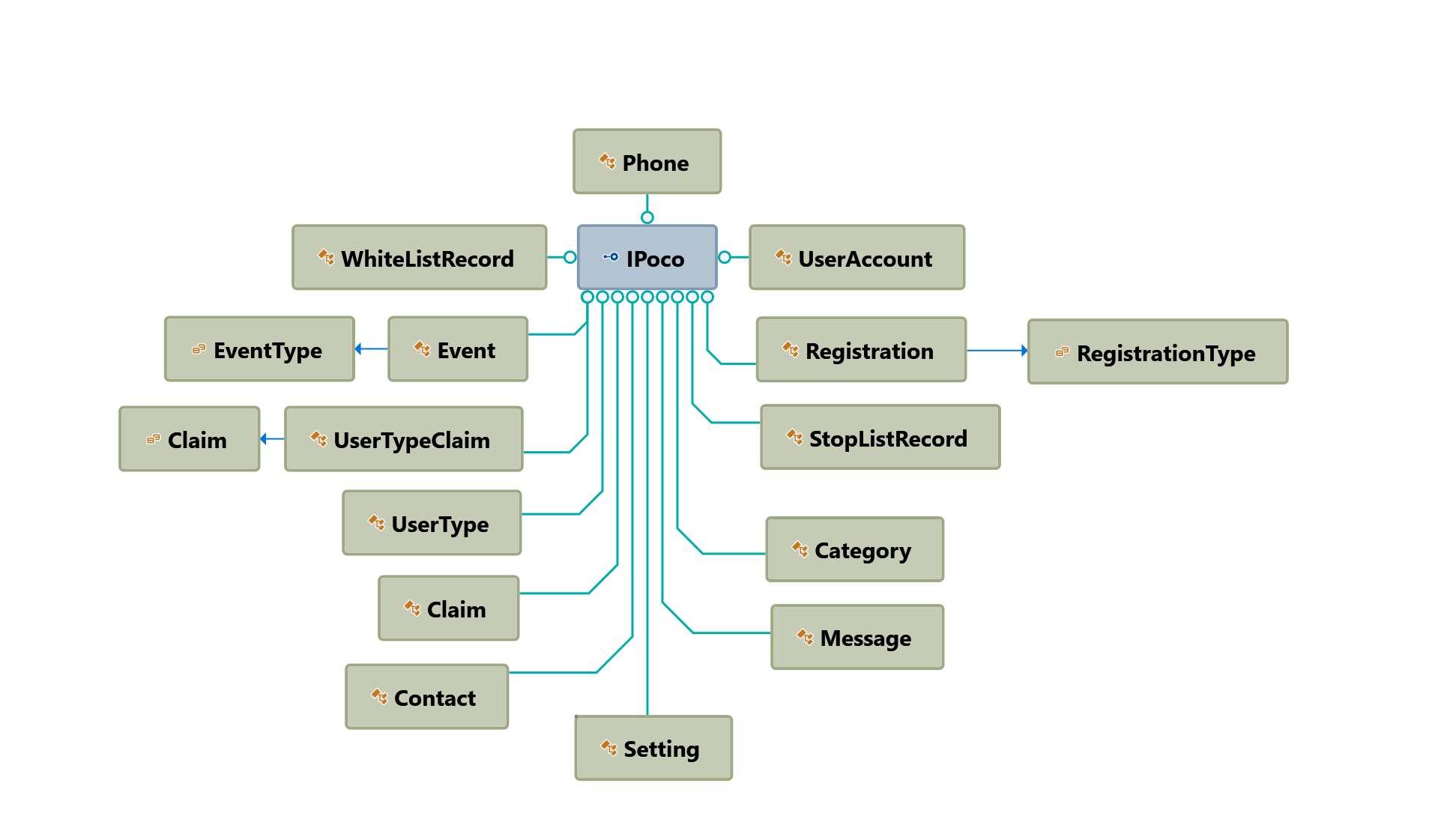
Ключевыми понятиями Linq2DB являются контекст, таблица и POCO-объект. Контекст представляет собой сущность которая обеспечивает подключение к базе данных, при этом данный функционал инкапсулирован внутрь класса, а интерфейс представляет собой набор полей, предоставляющих доступ к таблицам.

Таблица – сущность, отображающая реальную таблицу в базе данных, при этом имя таблицы Linq2BD и таблицы в базе данных могут не совпадать. Клиенты используют таблицу через интерфейс схожий с интерфейсом обычных .Net-коллекций. Таблицы реализует IQueriable интерфейс для формирования запросов и выглядит как массив POCO-объектов.

POCO-объект – объект для хранения данных, моделирует строку таблицы, при этом имя поля объекта может не совпадать с именем столбца. POCO-объект может иметь только поля и не иметь поведения, тогда он называется объектом передачи данных, а может и иметь поведение, тогда это будет значащий объект, при этом любой POCO-объект является независимым от фреймворка и не наследуется ни от одного из его классов.

На рис. 3.1 представлена иерархия POCO-объектов с общим интерфейсом IPOCO – который делает возможным доступ к любому объекту через уникальный идентификатор GUID.

GUID (Globally Unique Identifier) — статистически уникальный 128-битный идентификатор. Его главная особенность — уникальность, которая позволяет создавать расширяемые сервисы и приложения без опасения конфликтов, вызванных совпадением идентификаторов. Хотя уникальность каждого отдельного GUID не гарантируется, общее количество уникальных ключей настолько велико (2128 или 3,4028×1038), что вероятность того, что в мире будут независимо сгенерированы два совпадающих ключа, крайне мала.



**Рис. 3.1 Иерархия POCO-объектов**

3.2 Защита целостности данных.

Для обеспечения работоспособности системы необходимо предусмотреть механизм, который обеспечит наличие хотя бы одной записи в следующих таблицах UserTypes, EventTypes. Также необходимо предусмотреть механизм который будет блокировать создания новых записей в таблице Registration для аккаунтов из чёрного списка в случае обхода логики сервера.

Механизмы выполняющие данный функционал реализованы на основе триггеров приведённых в листинге 3.1.

**Листинг. 3.1 Триггеры**

CREATE TRIGGER AtleastOneUserType

ON UserTypes

FOR DELETE

AS

DECLARE @TABLE\_SIZE INTEGER

SET @TABLE\_SIZE = (SELECT COUNT(\*) FROM UserTypes)

IF @TABLE\_SIZE = 0 ROLLBACK TRANSACTION

GO

CREATE TRIGGER AtleastOneEventType

ON EventTypes

FOR DELETE

AS

DECLARE @TABLE\_SIZE INTEGER

SET @TABLE\_SIZE = (SELECT COUNT(\*) FROM EventTypes)

IF @TABLE\_SIZE = 0 ROLLBACK TRANSACTION

GO

CREATE TRIGGER CheckStopList

ON Registrations

FOR INSERT, UPDATE

AS

DECLARE @TABLE\_SIZE INTEGER

IF EXISTS (SELECT UserAccountId, EventId FROM StopListRecords WHERE UserAccountId IN (SELECT UserAccountId FROM inserted) AND EventId IN (SELECT EventId FROM inserted))

ROLLBACK TRANSACTION

GO

Для обеспечения безопасного процесса создания записей о настройках системы, предотвращения некорректной перезаписи текущих настроек, защиты от создания дубликатов предусмотрена хранимая процедура CreateOrUpdateSetting, которая в зависимости от текущего состояния таблицы производит запись в таблицу либо модификацию существующих данных.

Исходный код приведён в листинге 3.2.

**Листинг. 3.2 Хранимая процедура CreateOrUpdateSetting**

CREATE PROCEDURE CreateOrUpdateSetting @Id UNIQUEIDENTIFIER, @Name NVARCHAR(255), @Value NVARCHAR(255)

AS

IF EXISTS (SELECT \* FROM Settings WHERE SettingId = @Id AND Name != @Name)

THROW 50001, 'Identifier is used', 1;

IF EXISTS (SELECT \* FROM Settings WHERE Name = @Name AND (Value != @Value OR SettingId != @Id ))

BEGIN

UPDATE Settings SET SettingId = @Id, Value = @Value WHERE SettingId = @Id;

RETURN

END;

IF NOT EXISTS (SELECT \* FROM Settings WHERE Name = @Name)

BEGIN

INSERT INTO Settings VALUES (@Id, @Name, @Value)

RETURN

END;

GO

3.3 Реализация серверной части приложения.

Основу серверной части разрабатываемого приложения составляет логика по запросу и записи данных в базу данных. Ключевыми элементами это функциональности являются репозитории. Репозиторий – абстракция, позволяющая скрыть наличие СУБД, взаимодействуя с данными так, если бы они находились в виртуальной памяти процесса. Важной функцией репозитория является сокрытие интерфейса ORM-фреймворка, что в теории может облегчить его замену. То есть репозиторий – модифицированный вариант паттерна «Адаптер», внутри которого помимо вызова функций могут строиться запросы.



**Рис. 2.3 Схема классов паттерна «Адаптер класса»**

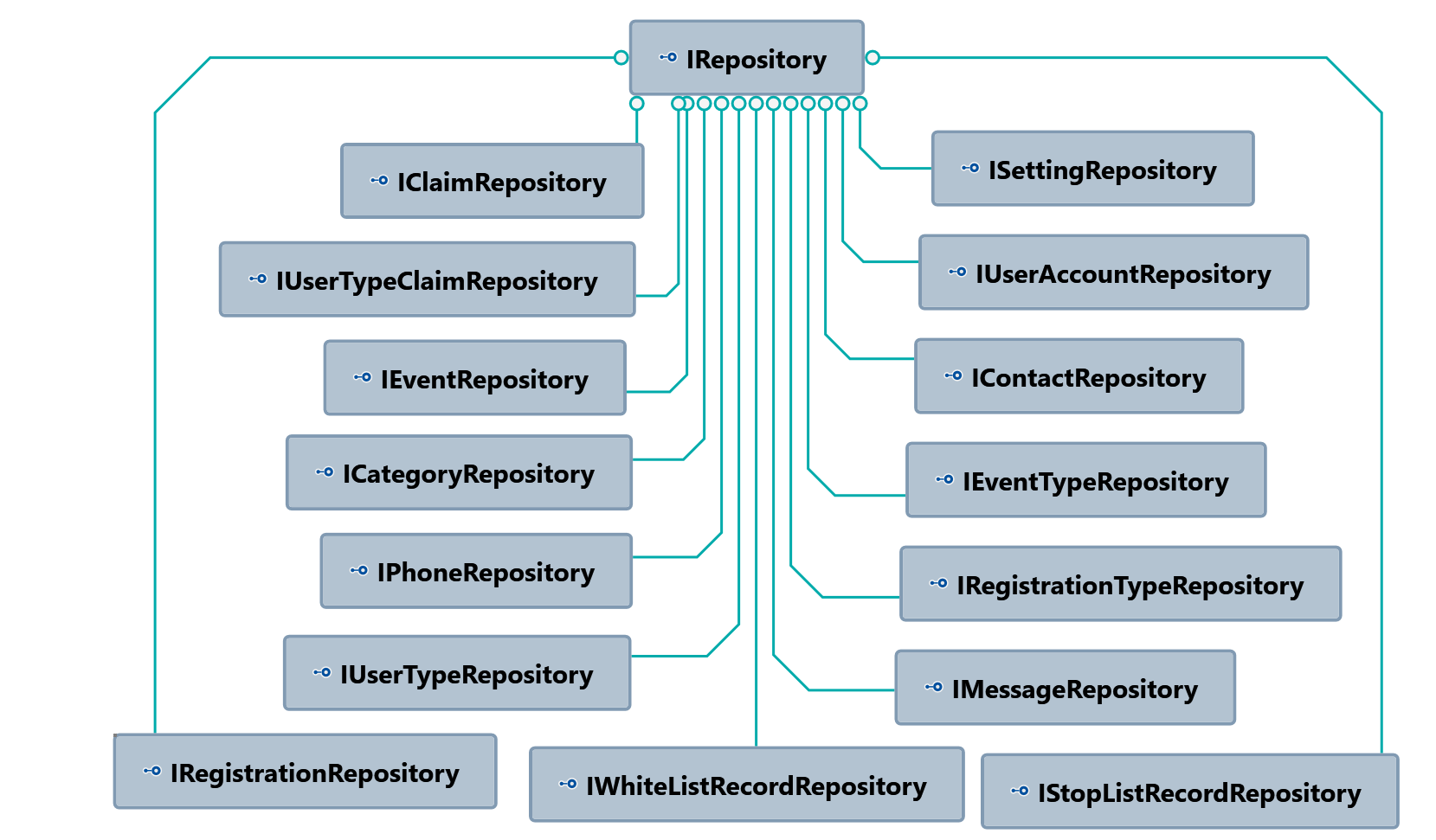
Участники:

1. Target – определяет зависящий от предметной области интерфейс, который использует Client;
2. Client – вступает во взаимоотношения с объектами, удовлетворяющими интерфейсу Target;
3. Adaptee – определяет существующий интерфейс, который нуждается в адаптации;
4. Adapter – адаптирует интерфейс Adaptee к интерфейсу Target.

В терминологии репозиториев паттерн «Адаптер» принимает следующий вид:

1. Target – типизированный интерфейс IRepository<T> и его наследники.
2. Adapter – типизированный класс Repository<T> и его наследники, которые реализуют интерфейс IRepository<T> .
3. Adaptee – абстракция ORM-фреймворка для подключения к базе данных. Для Linq2Db это наследник класса LinqToDB.Data.DataConnection.

Для каждой таблицы в которую будет осуществляться добавление, изменение или удаления данных был создан собственный интерфейс и класс, реализующий его. Для сущностей Claim, EventType, RegistrationType были созданы перечисления, которые ORM-фреймворк при генерации запросов подменяет соответствующим записями в базе данных.



В соответствии с трёхуровневой архитектурой архитектурой бизнес логика приложения располагается в сервисах. В процессе разработки был выявлен ряд проблем, который приводил к необходимости написание большого количества инфраструктурного когда, который не реализовывал бизнес логику. Большая часть сервисов занималась вызовом методов репозиториев. И преобразованием данных в модели уровня сервисов, при это присутствовало большое число зависимостей от различных репозиториев. Чтобы решить этот набор проблем, пришлось разработать новую архитектуру, удовлетворяющую требованиям приложения.

Требования к новой архитектуре:

1. Уменьшение числа идентичных моделей на разных уровнях системы.
2. Уменьшение (а лучше полное устранение) использования AutoMapper.
3. Возможность автоматизации регистрации типов в контейнере зависимостей.
4. Уменьшение числа зависимостей в классах.
5. Хорошая масштабируемость в ширину.
6. Возможность осуществлять логирование действий в одном месте.
7. Общий механизм валидации для каждого обращения от клиента.
8. Совместимость с существующими компонентами системы.

При анализе существующей архитектуры выявилась следующая закономерность: методы сервисов, которые вызываются при получении GET запросов не содержат никакой бизнес логики. Единственной их затачей является вызов соответствующих методов репозиториев и копирование данных между моделями с одинаковыми полями. В тоже время, никакой GET запрос не изменяет состояние системы. Другая часть запросов, преимущественно POST, напротив, только изменяют состояние системы не возвращая никаких данных. В этих запросах уже присутствует бизнес логика и часто требуется работа с доменными моделями перед изменением состояния системы. N-уровневая архитектура требует, чтобы подход к работе с обоими типами запросов был одинаков. Соответственно более сложные архитектурные решения необходимые для методов, изменяющих состояние системы затрагивают методы по запросу данных, создавая излишнюю сложность. Чтобы избежать этой проблемы разделим все действия в системе на команды и запросы.

Запрос – действие в системе, которое возвращает некоторое значение но не изменяет состояние системы.

Команда – действие в системе, которое, изменяет состояние системы и, как правило, не возвращает данные. Последнее замечание является очень важным, так как могут возникнуть ситуации, когда бизнес логика потребует передачи на сторону клиентского приложения данных, генерируемых в процессе выполнения команды. Наиболее часто в роли таких данных выступают уникальные идентификаторы. Однако, идентификатор может быть сгенерирован до выполнения команды, если он удовлетворяет следующим условиям:

1. Генерируется случайным образом.
2. Множество допустимых значений достаточно велико, и обеспечивает низкую вероятность коллизии при генерации идентификатора.
3. Если был сгенерирован идентификатор, то в результате выполнения команды состояние не измениться и возникнет исключение.

Невыполнение хотя бы одного из вышеперечисленных условий делает невозможным предварительную генерацию идентификатора, что делает необходимым осуществление этой операции при выполнении команды с последующей передачей значения вызывающей функции.

Команда и запрос представляют собой DTO, единственной задачей которых является передача данных для изменения сстояниясостояния. Для более наглядного разделения команд и запрос имеет смысл создать интерфейсы-маркеры ICommand и IQuery. Эта процедура придаст большую наглядность, однако она не является необходимой.

Выполнением команд и запросов занимаются обработчики. При этом никакой из обработчиков не проверяет корректность команды или запроса, не осуществляет логирование, не проверяет наличие прав у пользователя и не устанавливает значения по умолчанию. Операция может завершится либо успешно либо вызвать исключение, передача информации об ошибке или невыполнении операции в качестве результата выполнения функции не предусмотрена. Приведённый набор требований позволяет устранить зависимости от хранилища настроек по умолчанию, логировщика, сервисов аутентификации и других компонентов системы которые напрямую не относятся к бизнес логике по выполнению команды или запроса. Все обработчики команд реализуют интерфейс ICommandHandler<in TCommand>, а обработчики запросов - IQueryHandler<in TQuery, out TResult> (листинг 3.1)

public interface ICommandHandler<in TCommand>

where TCommand : ICommand

{

void Execute(TCommand command);

}

public interface IQueryHandler<in TQuery, out TResult>

where TQuery : IQuery

{

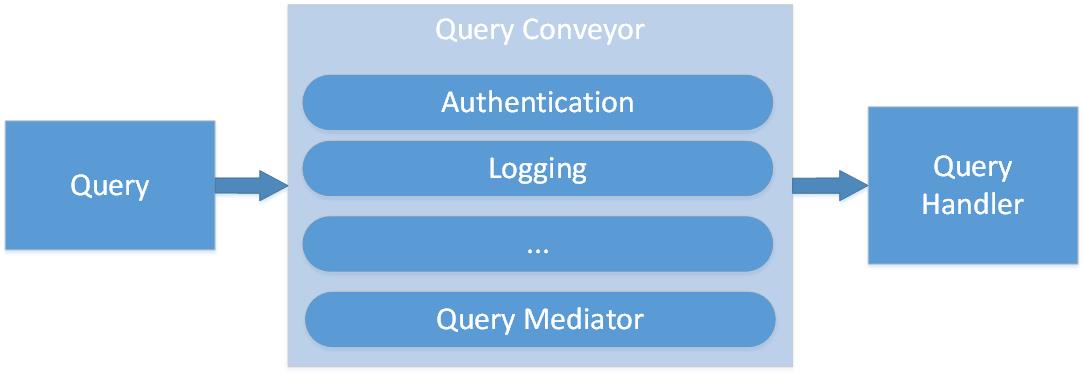
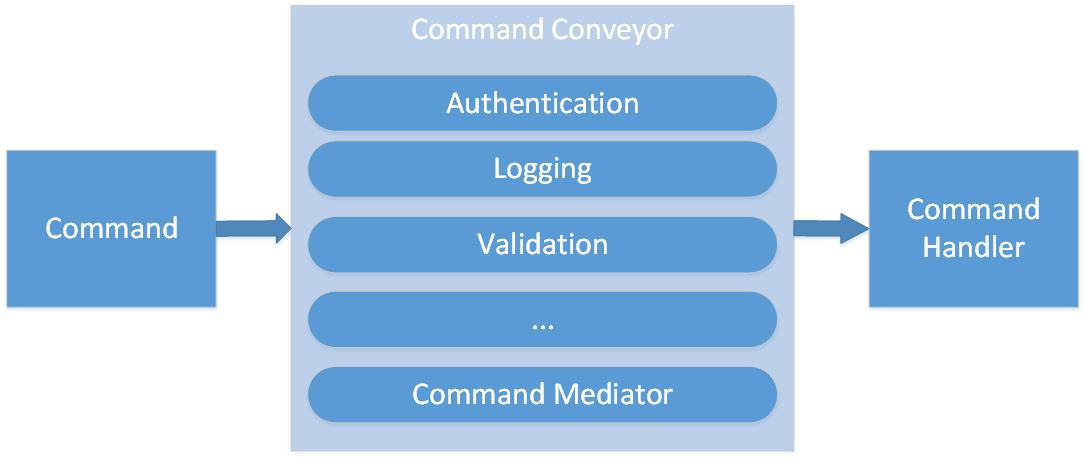
TResult Execute(TQuery query);

}

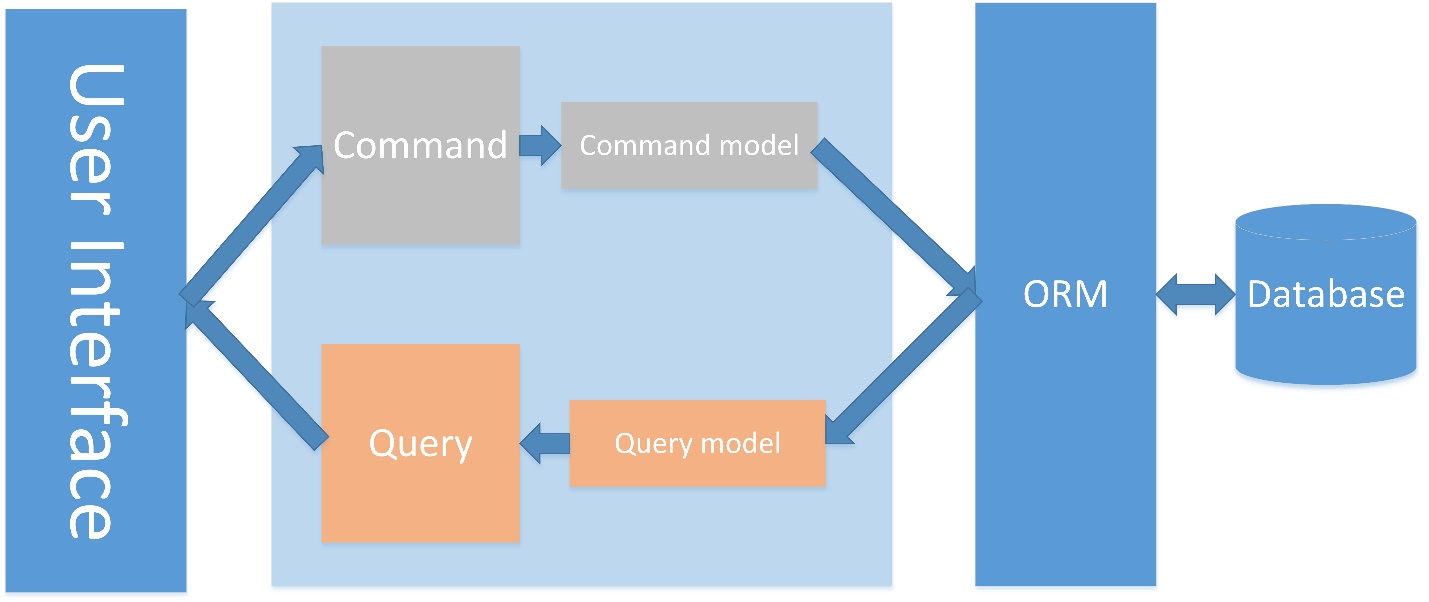
Связующим звеном между командами и обработчиками команд является медиатор команд, а запросов и обработчиков запросов – медиатор запросов. Команды и запросы поступают на соответствующий медиатор, где осуществляется поиск необходимого обработчика и передача команды или запроса на выполнение. Выбор необходимого обработчика осуществляется с использование IoC-контейнера. В связи с этим возможны 5 сценариев работы медиатора:

1. Невозможно найти обработчик, так как команда/запрос имеют null значение.
2. Невозможно найти обработчик, так как он отсутствует в IoC-контейнере.
3. Возникла неоднозначность при выборе обработчика – в IoC-контейнере нашлось несколько типов удовлетворяющих условию поиска.
4. Найден ровно один обработчик, однако возникло исключение при его выполнении.
5. Найден ровно один обработчик, который был успешно выполнен.

Медиатор содержит единственную зависимость от экземпляра класса реализующего интерфейс IDependencyResolver. Стоит отметить, что обработчики команд и запросов реализуют бизнес логику приложения. Инфраструктурная часть логики: логирование, аутентификация, валидация, которая является общей для всех команд или запросов является независимой от обработчиков и входит в жизненный цикл команды/запроса. Все инфраструктурные действия осуществляются до выполнения кода обработчика. При этом поиск советующего обработчика также является этапом жизненного цикла. Для прохождения команд и запросов по этапам их жизненного цикла были спроектированы и реализованы конвейер команд и конвейер запросов. Конвейеры позволяют отделить бизнес логику от инфраструктурной и обеспечить простое масштабирование системы. На рисунке 3.1 представлена логическая структура конвейера команд, а на рисунке 3.2 конвейера запросов.



Введение концепции разделения ответственности между командами и запросами позволило заменить двунаправленный поток данных, который присущ n-уровневой архитектуре, на однонаправленный изображённый на рисунке 3.3.



Подводя итог разработки серверной части приложения стоит отметить преимущества и недостатки архитектуры основанной на разделении ответственности между командами и запросами.

Преимущества:

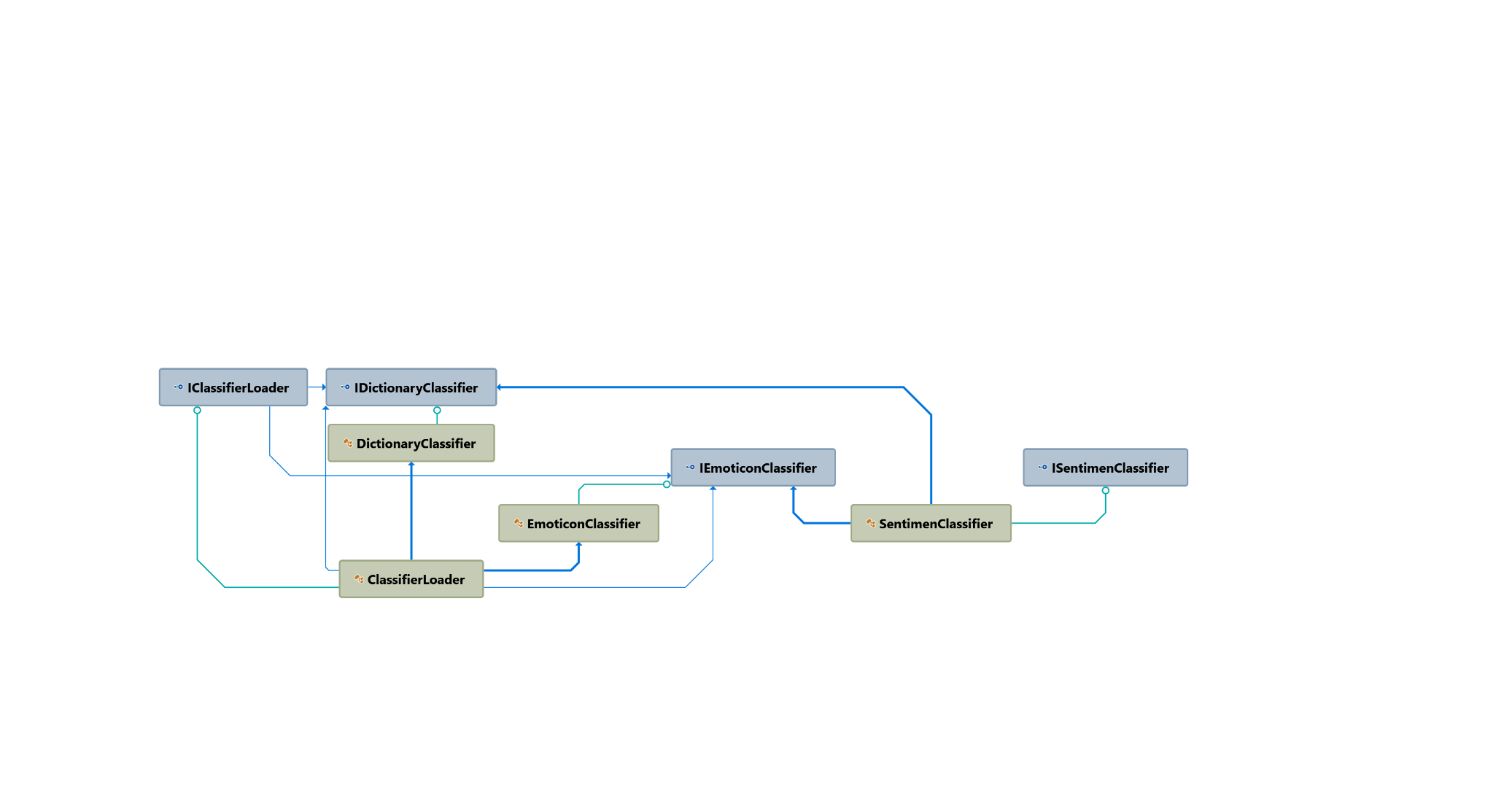
1. Возможность использовать сложные модели для команд и простые модели для запросов.
2. Возможность отделения инфраструктурной логики от бизнес логики.
3. Простой поток данных внутри приложения.
4. Уменьшения числа зависимостей между классами.
5. Возможность частично автоматизировать процесс инициализации IoC-контейнера.

Недостатки:

1. Позднее связывание команды или запроса с обработчиком, не возможно отследить наличие обработчиков на этапе компиляции.
2. Не весь функционал может быть реализован в данной парадигме, например, метод pop стека нарушает описанную парадигму так как одновременно возвращает данные и изменяет состояние.
   1. Разработка фреймворка для оценки эмоциональной окраски текста

Основу фреймворка для оценки эмоциональной окраски текста составляет класс SentimentClassifier реализующий интерфейс ISentimentClassifier. Данный класс агрегирует два классификатора с интерфейсами IEmoticonClassifier и IDictionaryClassifier реализующие методы классификации сообщений на основе анализа анализа смайлов и на основе обработки текста с использованием словарей. Классы EmoticonClassifier DictionaryClassifier реализуют ранее упомянутые интерфейсы. В соответствии с алгоритмом описанным в главе 2. Загрузку данных для анализа (словари смайлов и словари эмоциональной окраски) осуществляет производит экземпляр класса ClassifierLoader реализующий интерфейс IClassifierLoader.

Схема классов разработанного фреймворка приведена на рисунке 3.4



Основной сложностью при работе с фреймворком является обеспечение существования небольшого количества экземпляров класса SentimentClassifier по следующим причинам:

1. Большое время создания обусловленное загрузкой данных из файла.
2. Достаточно большой объём занимаемой памяти из-за необходимости хранения словарей в памяти процесса.

Для обеспечения существования единственного экземпляра заданного класса можно использовать два способа: паттерн «Одиночка» и специальную настройку IoC- контейне “As singleton”. Второй вариант являтся более предпочтительным и будет использоваться в дальнейшем так как избавлен от недостатков паттерна «Одиночка» и не требует написания дополнительных классов обёрток. В добавок ко всему разрабатываемое веб-приложение уже использует IoC-контейнер с требуемой функциональностью.

* 1. Клиентское приложение

Клиентское приложение было реализована на языки javascript, c использованием спецификации ECMAScript 6. При его реализации были использованы следующие библиотеки и инструменты:

1. React, React-Dom, React-Thunk, React-Router – основные библиотеки экосистемы React.
2. Redux, React-Redux – библиотеки, реализующая Redux-архитектуру.
3. React-MDL – библиотека компонентов пользовательского интерфейса.
4. Babel – инструмент для трансляции javascript (ECMAScript 6 в ECMAScript 5).
5. Webpack – инструмент для сборки проекта.
6. Eslint – инструмент для контроля качества кода.
   1. Наполнение тестовыми данными

Для эффективной разработки и тестирования приложения необходимо иметь набор исходных данных, которые максимально соответствуют информации, которая будет поступать от реальных пользователей. Данная задача может быть решена несколькими способами:

1. Наполнение базы данных осуществляется людьми.
2. На основании предположений о поведении реальных пользователей генерируются данные генерируются специально разрабатываемым программным обеспечением.
3. Используются данные, сгенерированные реальными пользователями, которые доступны из публичных баз данных или через открытые API.

При наполнении базы данных реальными людьми возникает ряд проблем: медленная скорость заполнения и однотипность данных. Данные будут однотипные, так как один и тот же человек, повторяя одни и те же действия не сможет обеспечить нужное разнообразие данных. При небольшом объёме данная особенность не является существенной и полученные данные будут достаточно хорошо соответствовать данным, которые в последствии будут генерироваться пользователями.

Использование специально разработанного программного обеспечения позволяет свести к минимуму время генерации данных, однако такой подход неспособен обеспечить достаточное разнообразие данных. Так же, для того, чтобы сымитировать поведение пользователей, необходимо спрогнозировать поведение реальных пользователей, выделить ожидаемые шаблоны поведения и описать их формально, чтобы они могли были быть описаны с использованием какого-либо языка программирования.

Подход, основанный на использовании реальных данных избавлен от всех перечисленных выше недостатков:

1. Отсутствуют затраты на генерацию данных.
2. Данные максимально разнообразны.
3. Данные максимально соответствуют ожидаемым.
4. В данных присутствуют ошибки обусловленные человеческим фактором.

Для реализации такого подхода необходимо решить следующие задачи: найти источник данных, получить данные и адаптировать их под разрабатываемую систему. С источником данных никаких проблем не возникает, так как все сервисы перечисленные в главе 1 имеют открытые web-API. Задача по заполнению базы данных может быть декомпозирована следующим образом:

1. Получить образец данных через web-API.
2. На основании данных сгенерировать языковые конструкции (классы) для дальнейшей работы с данных.
3. Организовать временное хранилище данных.
4. Загрузить необходимый объём данных из источника в хранилище.
5. Выбрать данные, необходимые для разработки.
6. Сгенерировать SQL скрипт для заполнения базы данных.

В результате анализа трафика на сайте EventBriter был обнаружен http запрос для загрузки нужных данных. Образец данных был получен путём отправки GET-запроса по этому адресу. Результатом выполнения запроса стали данные в текстовом формате json.

С использование сервиса <http://json2csharp.com/> были сгенерированы классы языка C#, соответствующие модели данных используемой web-API. Недостатком сервиса json2csharp является то, что имена классов и полей однозначно соответствуют именам полей в json-объекте, однако такой подход не является общепринятым. Mictorsoft, компания разработчик платформы .Net, рекомендует при именовании классов рекомендует использовать Camel-нотацию: первый символ и каждое последующее слово начинается с символа в верхнем регистре. Для сопоставления имён полей объектов класса языка C# с полями json-объектов в процессе серелизации используются атрибуты (метаданные включаемые в исходный код). Для приведения сгенерированных к принятым стандартам была разработана консольная утилита, которая анализирует исходный код на языке C#, определяет имена классов и полей, помечает их необходимыми параметризированными атрибутами и преобразует в соответствии с Camel-нотации. Сгенерированный код может быть использован при разработке приложений без дополнительной обработки.

Для хранения данных была использована СУБД RavenDB. RavenDB представляет собой нереляционную документо-ориентированную СУБД разработанную специально для платформы .Net. В отличие от классических реляционных баз данных RavenDB позволяет сохранять любые серелизуемые объекты без предварительного описания схемы данных. В документо-ориентированном хранилище любой сохранённый объект может быть извлечён по уникальному ключу, также существует возможность делать различные выборки и строить индексы для более быстрого поиска нужных данных. Выбор данной СУБД был обусловлен её схемонезависимостью.

На базе сгенерированных классов было разработанное консольное приложение для загрузки данных с удалённого сервиса по средствам GET-запросов. Приложение загружает данные по протоколу HTTP, производит десерелизацию и сохранят полученные объекты в базу данных для дальнейшего использования.

Последним этапом работы стала разработка приложения, которое из загруженного набора данных производит выборку необходимых полей, и на их основании генерирует инициализирующий SQL-скрипт.

В результате был разработан набор утилит позволяющий осуществить инициализацию базы данных начальными значениями источником которых является web-API. Стоит отметить, применимость разработанного программного комплекса не ограничивается предметной областью приложения, разрабатываемого в рамках дипломного проектирования.