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 чтения и обработки новостной ленты Twitter**

Курсовая работа

студента 3 курса специальности

1-40 01 01 «Программное обеспечение информационных технологий»

дневной формы получения образования

|  |  |
| --- | --- |
|  | Научный руководитель  Скращук Владислав Сергеевич  старший преподаватель кафедры современных технологий программирования |
|  |  |

Гродно, 2015

**РЕЗЮМЕ**

Стельмашенко Андрей Николаевич

Курсовая работа – «Разработка приложения для чтения и обработки новостной ленты Twitter», 20 страниц, 5 иллюстраций, 3 использованных источника.

Ключевые слова – короткие сообщения, лента сообщений, микроблог, Twitter, 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 – микроблоги, протоколы аутентификации, документо-ориентированные базы данных, Web API.

Предмет исследования – Twitter, OAuth, RavenDb.

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 – “The development of the systems of short messages view and processing” is research in the field of computer science. Social network Twitter is subject for scientific investigation. The most important part of investigation is research of message lines. Main tools for developing are: HTML, CSS, JavaScript, C#. Application architecture analyzes with usage of Resharper architecture tools.

**СОДЕРЖАНИЕ**

[**СОДЕРЖАНИЕ** 4](#_Toc469241881)

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ, УСЛОВНЫХ ОБОЗНАЧЕНИЙ 5](#_Toc469241882)

[**ВВЕДЕНИЕ** 6](#_Toc469241883)

[ГЛАВА 1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ 8](#_Toc469241884)

[1.1 Основные аспекты 8](#_Toc469241885)

[1.2 Обзор существующих решений 8](#_Toc469241886)

[ГЛАВА 2 ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ 10](#_Toc469241887)

[2.1 Объектная модель данных 10](#_Toc469241888)

[2.2 Физическая модель базы данных. 12](#_Toc469241889)

[2.3 Доступ к данным. 13](#_Toc469241890)

[2.4 Клиентское приложение 20](#_Toc469241891)

[ГЛАВА 3 РЕАЛИЗАЦИЯ 23](#_Toc469241892)

[3.1 Система аутентификации 23](#_Toc469241893)

[3.2 Реализация авторизации приложения в Twitter 24](#_Toc469241894)

[3.3 Получение и сохранение коротких сообщений 25](#_Toc469241895)

[3.4 Реализация пользовательского интерфейса 26](#_Toc469241896)

[ЗАКЛЮЧЕНИЕ 29](#_Toc469241897)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 30](#_Toc469241898)

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. |
| NoSQL | Not only structured query language | ряд подходов, направленных на реализацию хранилищ баз данных, имеющих существенные отличия от моделей, используемых в традиционных реляционных СУБД с доступом к данным средствами языка SQL. |
| REST | Representational state transfer | стиль архитектуры программного обеспечения для распределенных систем, как правило, используется для построения веб-служб. |

**ВВЕДЕНИЕ**

В настоящее время в эпоху быстро развивающихся интернет технологий всё большее влияние на жизнь современного человека оказывают различного рода социальные сети. Данные сервисы предназначены для построения, отображения и организации социальных отношений между пользователями. Большинство социальных сетей предоставляют функционал для коммуникации посредством текстовых сообщений, а также средства для визуализации отношений посредством социальных графов. В последнее время наиболее популярными социальными сетями попадающими под данное описание являются Twitter, Facebook, Вконтакте, LinkedIn и другие.

Быстрый рост степени покрытия сетью Интернет территории земного шара привёл к стремительному росту числа пользователей и числа сообщений отправляемых ими. Так ежедневно пользователи Twitter отправляют около 500 миллионов коротких сообщений (твитов), а количество активных пользователей системы составляет около 300 миллионов[1]. В этих условиях социальные сети приобрели популярность среди различного рода организаций предоставляющих рекламные услуги, что в свою очередь привело к росту количества нежелательных сообщений содержащих рекламу. Так же для современных социальных сетей характерен вирусный эффект – массовая пересылка сообщений, ссылок на видео и другой контент другим пользователям. Данный эффект создаёт некоторые неудобства в виде переполненных лент новостей или личных сообщений, поэтому высокую актуальность приобретает предварительная обработка отображаемых сообщений перед попаданием их в ленту пользователя. Данная проблема будет рассмотрена на примере микроблога Twitter, позволяющего публиковать короткие сообщения.

Цель курсовой работы – разработать программное обеспечение для обхода ограничений Twitter API по количеству просматриваемых пользовательских сообщений и реализовать функцию блокировки рекламы

В рамках данной курсовой работы предусмотрено решение следующих задач:

1. Создание простейшего клиентского приложения для сервиса Twitter.
2. Реализация защиты приложения от несанкционированного доступа.

ГЛАВА 1  
АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 **Основные** аспекты

Для передачи данных между клиентским приложением и сервером Twitter предоставляет два вида API: REST API и потоковое API. Для передачи сообщений и иного контента созданного до настоящего момента применяется REST API. Потоковое API обеспечивает получение данных в режиме реального времени [2]. При этом, в зависимости от популярности клиентского приложения определяются ограничения на количество запросов к API. Данный тип межпроцессного взаимодействия реализуется с помощью использования протокола HTTP.

Для работы с API Twitter предлагает ряд открытых библиотек: Linq2Twitter, TweetSharp, Tweetinvi, Twitter4J, twitcurl и другие. Для платформы .Net на языке C# реализованы библиотеки TweetSharp и Linq2Twitter.

В связи со спецификой реализаций клиентских приложений возникает необходимость комбинированного использования двух типов API.

1.2 Обзор существующих решений

Наиболее распространённым клиентским приложением для Twitter является официально рекомендованное приложение TweetDeck (рис. 1.1). Широко распространены Trillian, MetroTwit, Twitterrific и другие. Однако лишь TweetDeck предоставляет возможность фильтрации сообщений.



Рисунок 1.1 TweetDesk – официальный клиент

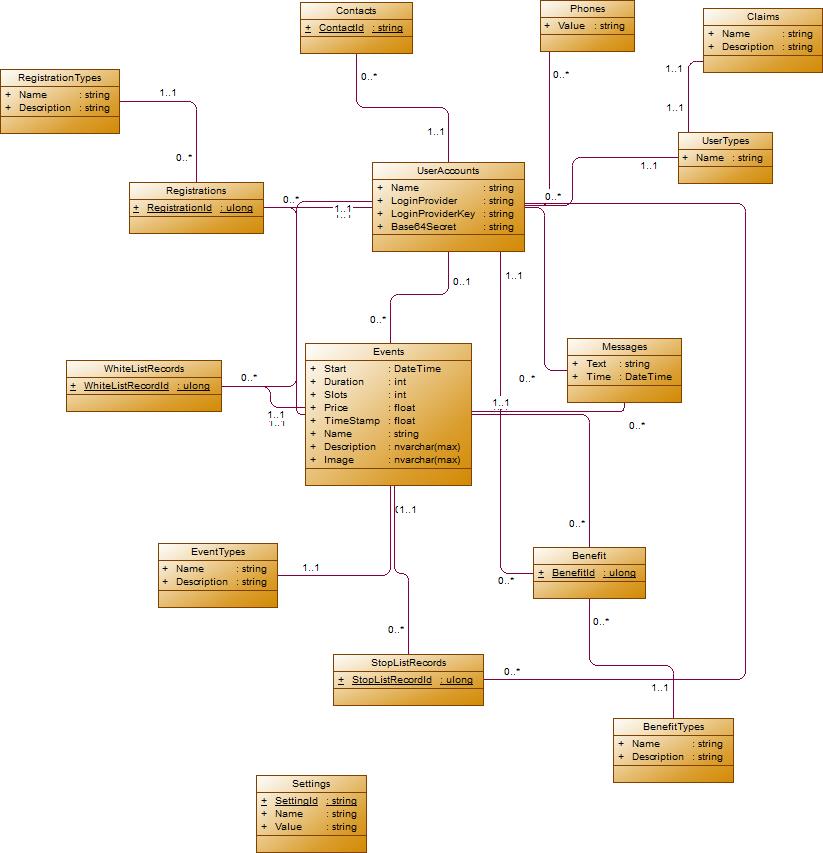
ГЛАВА 2  
ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

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. Поощрение (Benefit) – описывает дополнительные возможности пользователя по взаимодействию с событием.
8. Тип поощрения (BenefitType) – описывает общие характеристики для группы схожих поощрений (отношение один ко многим).
9. Контакт (Contact) – сущность, описывающая право одного пользователя получить информацию о другом пользователе.
10. Чёрный список (StopList) – массив записей (StopListRecord), вида событие-аккаунт, которые ограничивают возможность регистрации пользователя при попадании пользователя в это список.
11. Чёрный список (WhiteList) – массив записей (WhiteListRecord), вида событие-аккаунт, которые дают право на регистрацию если по умолчанию такая функция недоступна.
12. Настройка (Setting) – именованная сущность описывающая параметры конфигурации системы (не относится к предметной области).

Объектная модель данных представлена на рисунке 2.1.



**Рис. 2.1 Объектная модель данных**

2.2 Физическая модель базы данных.

Физическая модель базы данных построена на основании объектной модели с введением дополнительных таблиц для реализации отношения многие ко многим. Также в структуру базы данных введены суррогатные ключи типа uniqueidentifier которые в терминологии платформы .Net называются GUID.



**Рис. 2.2 Физическая модель базы 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. В силу вышеупомянутого подхода к оптимизации производительности можно с уверенностью ответить на вопрос, который беспокоит сотни миллионов пользователей Twitter-а: когда появиться функция редактирования сообщений? Ответ: она не появится до тех пор, пока Twitter не откажется от подхода оптимизации производительности, основанном на замене операции редактирования операциями чтения, удаления записи.

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 это и ей подобных утечек в абстракции.

Для упрощения реализации слоя доступа данных удобно использовать ORM фреймворки. Любой ORM фреймворк спроектирован так, чтобы быть наиболее универсальным. Наибольшей проблемой при интеграции стороннего решения в разрабатываемую систему является различие их интерфейсов. Некоторые фреймворки оперируют такими понятиями как таблицы, некоторые – множествами. Очень часто одинаковые по смыслу операции имеют разные названия: get и load, select и map, delete и remove. В общем случае данная проблема может быть описана следующим образом: необходимо преобразовать интерфейс класса таким образом, чтобы он был совместим с интерфейсом, который ожидает класс-клиент.

Для решения данной задачи подходит паттерн проектирования «Адаптер», который структурирует классы и объекты, обеспечивая совместимость интерфейсов. Существуют две возможные реализации паттерна «Адаптер». Первая реализация основана на использовании множественного наследования (рис. 2.3) и представляет собой адаптер класса.



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

Вторая реализация применяет композицию объектов и является адаптером объектов.

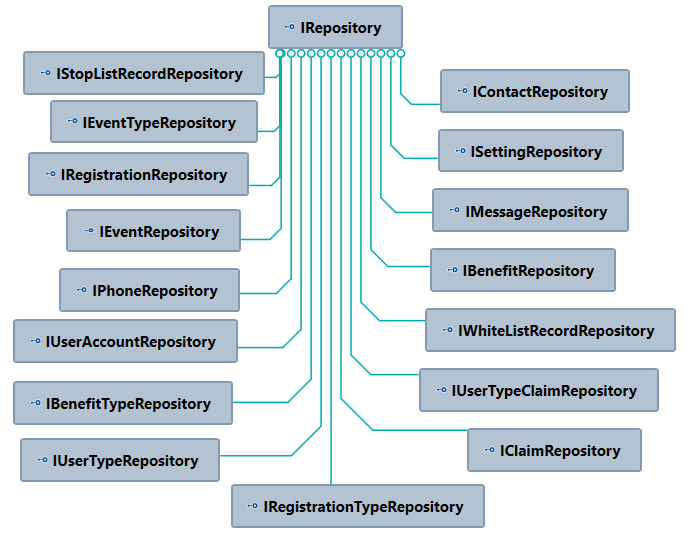


**Рис. 2.4 Схема классов паттерна «Адаптер объекта»**

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.

Основной задачей адаптеров слоя доступа к данным является сокрытия ORM-фреймворка. С точки зрения пользователя наиболее является такой объект доступа к данным, чей интерфейс будет подобен коллекции. Данным критериям соответствует такая разновидность паттерна адаптер как «Репозиторий». Интерфейс репозитория включает в себя следующие методы: Get, Insert, Delete, Update. Для каждой таблицы в базе данных создан свой интерфейс репозитория, наследующий методы базового интерфейса и при необходимости расширяет допустимый набор методов. В терминологии паттерна «Адаптер» интерфейс IRepository соответствует Target, ORM-фреймворк – Adaptee, класс, реализующий интерфейс IRepository – Adapter. Иерархия репозиториев изображена на рисунке 2.4.



**Рис. 2.5 Иерархиея репозиториев**

Наиболее сложным с точки зрения реализации является метод Get. Существует ряд проблем:

1. Как реализовать получения одного или нескольких объектов.
2. Как передавать критерий для выборки объектов.
3. Как каком уровне формировать запрос на выборку объектов.
4. Как избежать дублирования кода, если одинаковый запрос нужен в разных местах не делая из интерфейса перечисления списка всевозможных выборок.
5. Как организовать выборку только нужных полей из таблицы.

Решением первой проблемы является введение двух методов Get и GetList. Предполагать, что список из одного элемента это тоже список является нецелесообразным, так как ожидается наличие большого числа запросов, ответом на которые будет строго один элемент и каждый раз выбирать первый элемент из списка нецелесообразно.

Получать объекты из базы можно несколькими путями: использовать хранимые процедуры, передавать SQL код серверу напрямую, использовать LINQ-выражения. Первый подход нарушает инкапсуляцию и порождает размытие бизнес-логики. Задача сервера баз данных – хранить данные и предоставлять к ним доступ, но никак не реализовывать бизнес-логику. Второй подход крайне неэффективный, так как придётся передавать простые строки. Это значит, что не будет ни подсветки синтаксиса, ни подсказок при написании кода, ни ошибок компиляции при неправильно написанном запросе. LINQ – гибкий механизм без недостатков первых двух подходов: бизнес-логика не уходит на сервер баз данных, есть подсветка кода, ошибки синтаксиса будут отловлены при компиляции. Недостатком являются затраты процессорного времени на трансляцию LINQ-выражения в SQL код, однако данный недостаток компенсируется кешированием.

Строить запросы необходимо на уровне слоя доступа к серверу баз данных, это его основная функциональность. При этом должна быть возможность написать такой запрос на уровне сервисов (для избежания перекомпиляции уровня доступа к данным, при написании тестового когда и кода имитирующего работу не реализованного сервиса или прототипа).

Решение четвёртой проблемы осложняется тем, что при решение первой проблемы количество точек входа в систему было ограничено двумя, это значит, что все запросы можно разделить не более чем на две группы с различными интерфейсами. При этом запросы первой группы должны возвращать объект не являющийся коллекцией элементов, а запросы второй группы возвращают коллекцию элементов, при этом стоит учесть, что объект со значение null это всё равно корректный объект, пустая коллекция и коллекция с одним элементом – также корректные результаты выполнения запроса, а вот получение от сервера баз данных коллекции вместо одного объекта является ошибкой целостности данных и ведёт к возникновению исключительной ситуации во время выполнения программы.

В общем случае необходимо иметь механизм позволяющий определить семейство взаимозаменяемых алгоритмов с разным поведением. Исходя из данного определения составленного на основе требований к архитектуре можно применить паттерн проектирования «Стратегия».



**Рис. 2.6 Схема классов паттерна «Стратегия»**

Участники:

1. Strategy – стратегия, общий для всех поддерживаемых алгоритмов интерфейс. Экземпляр класса Context использует этот интерфейс для вызова конкретного метода определённого в классе ConcreteStratege.
2. ConcreteStrategy – конкретная стратегия, реализующая алгоритм и предоставляющая к нему доступ через интерфейс объявленный в классе Strategy.
3. Context – контекст, использует алгоритм, реализованный в классе ConcreteStrategy через интерфейс Strategy.

Типизированная стратегия также позволит решить пятую проблему: частью алгоритма построения запроса является операция Select, которая при создании объекта требуемого класса запросит значения только нужных полей из таблицы в базе данных.

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**

2.5 Клиентское приложение

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**

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.

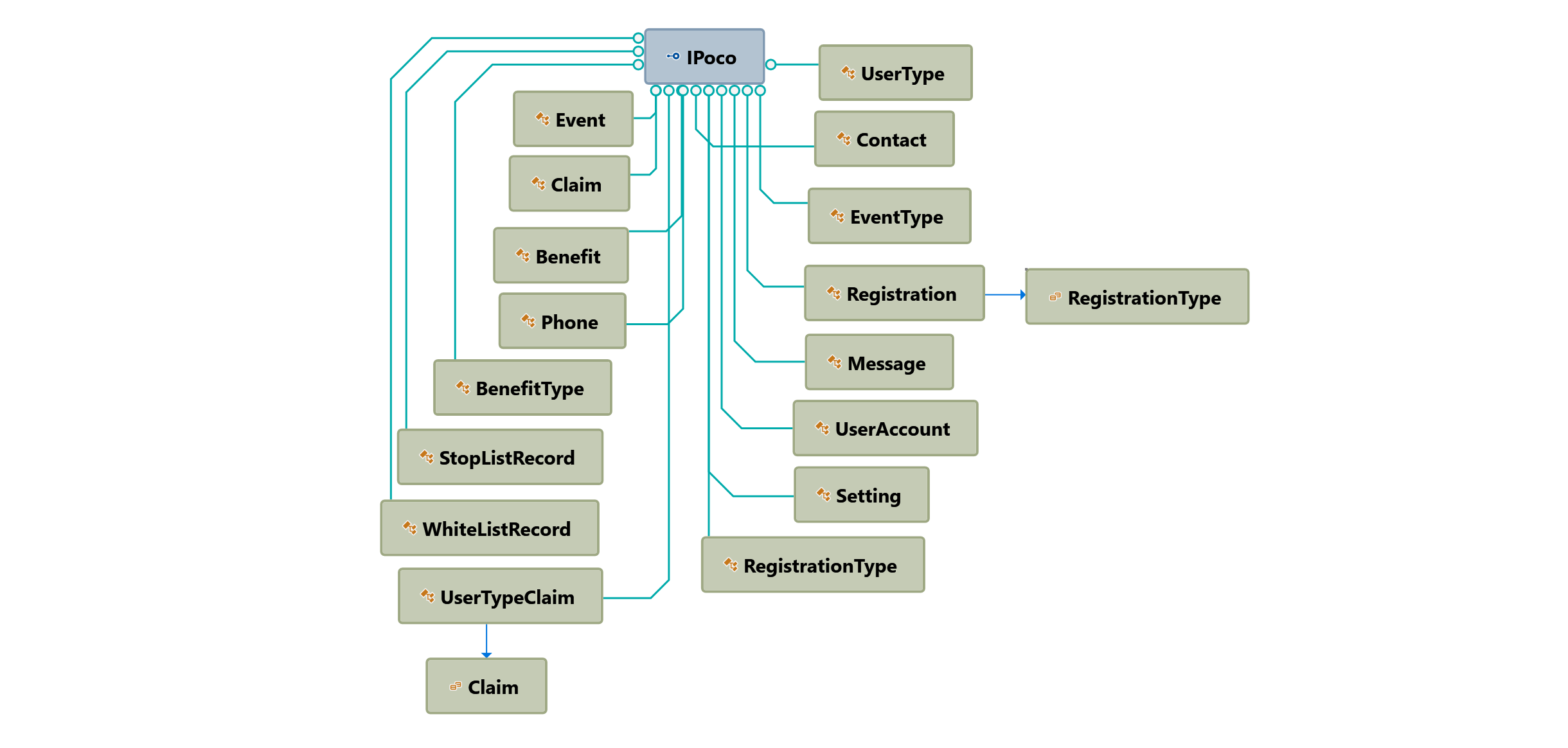
Таблица – сущность, отображающая реальную таблицу в базе данных, при этом имя таблицы Linq2BD и таблицы в базе данных могут не совпадать. Клиенты используют таблицу через интерфейс схожий с интерфейсом обычных .Net-коллекций. Таблицы реализует IQueriable интерфейс для формирования запросов и выглядит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

Класс AuthModule предоставляет функционал для токена авторизации пользователю в соответствии с его логином и паролем. Класс CredentialsStorage предоставляет функционал для хранения учётных данных пользователей, в том числе и пароль в зашифрованном виде. Всторенный в фреймворк NancyFX класс Tokenizer реализующий интерфейс ITokenizer предоставляет функционал для генерации токенов в случае успешной проверки пользовательских учётных данных. При этом нет необходимости хранить эти токены в базе данных, так как алгоритм позволяет проводить проверку подлинности без использования копии оригинального токена. На вход алгоритму поступает список идентификаторов аккаунтов в сервисе Twitter а также адрес пользовательской электронной почты. Этого достаточно для однозначного установления прав пользователя.

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

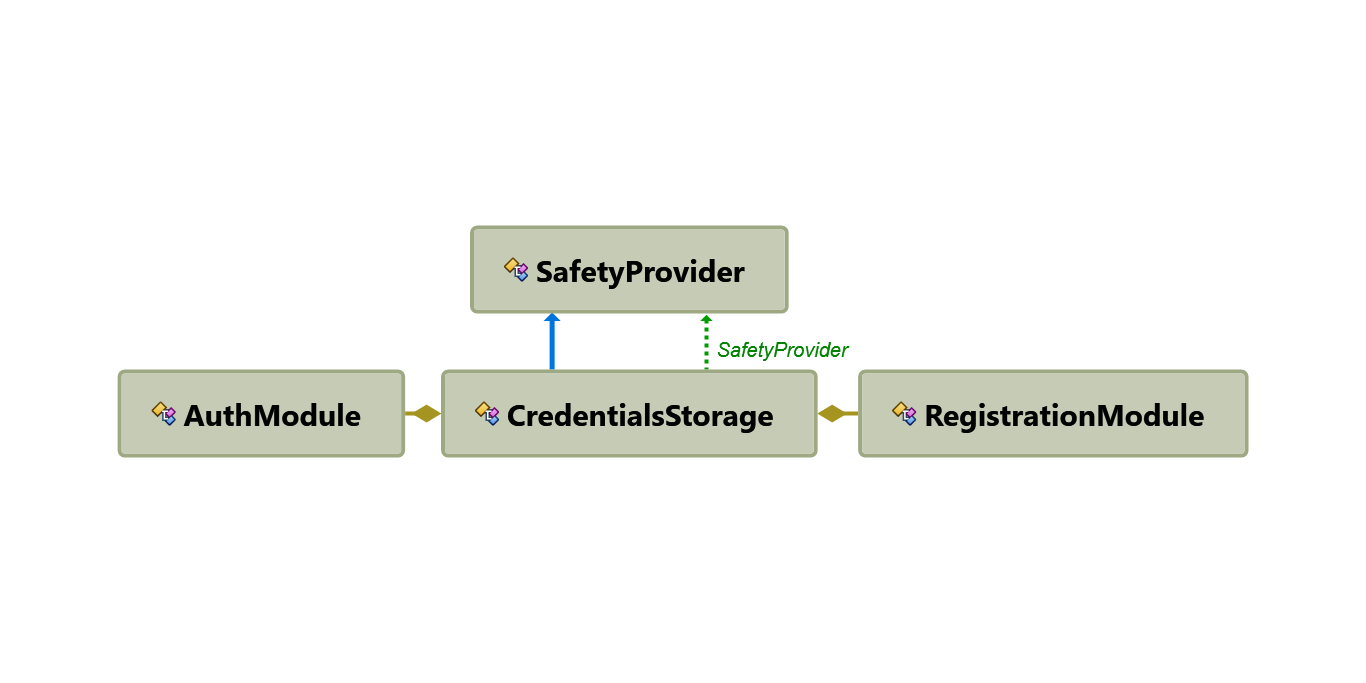
GUID (Globally Unique Identifier) — статистически уникальный 128-битный идентификатор. Его главная особенность — уникальность, которая позволяет создавать расширяемые сервисы и приложения без опасения конфликтов, вызванных совпадением идентификаторов. Хотя уникальность каждого отдельного GUID не гарантируется, 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**

Класс RegistrationModule предоставляет функционал для регистрации новых пользователей в системе сохраняя учётные данные в объект класса CredentialsStorage и генерируя новый токен для идентификации пользователя.

Для обеспечения безопасного хранения паролей в базе данных, а также защиты от несанкционированного доступа разработан класс SafetyProvider который предоставляет функционал для расчёта хеша и проверки пароля. Стоит отметить, что проверка пароля осуществляется за константное время для предотвращение атаки путём подбора пароля по времени отклика системы. При стандартном сравнении ответ о несовпадении хешей паролей будет получен при нахождении первого не совпавшего байта. То есть, для любых хешей на чикающихся с 28 различных комбинаций 0 и 1 в первом байте по наиболее долгому времени отклика системы можно точно установить значение первого байта. Аналогичная операция может быть проделана для последующих байтов. Такой подход существенно сокращает количество обращений к системе для подбора пароля. Ниже представлена схема классов, реализующих аутентификацию в системе.

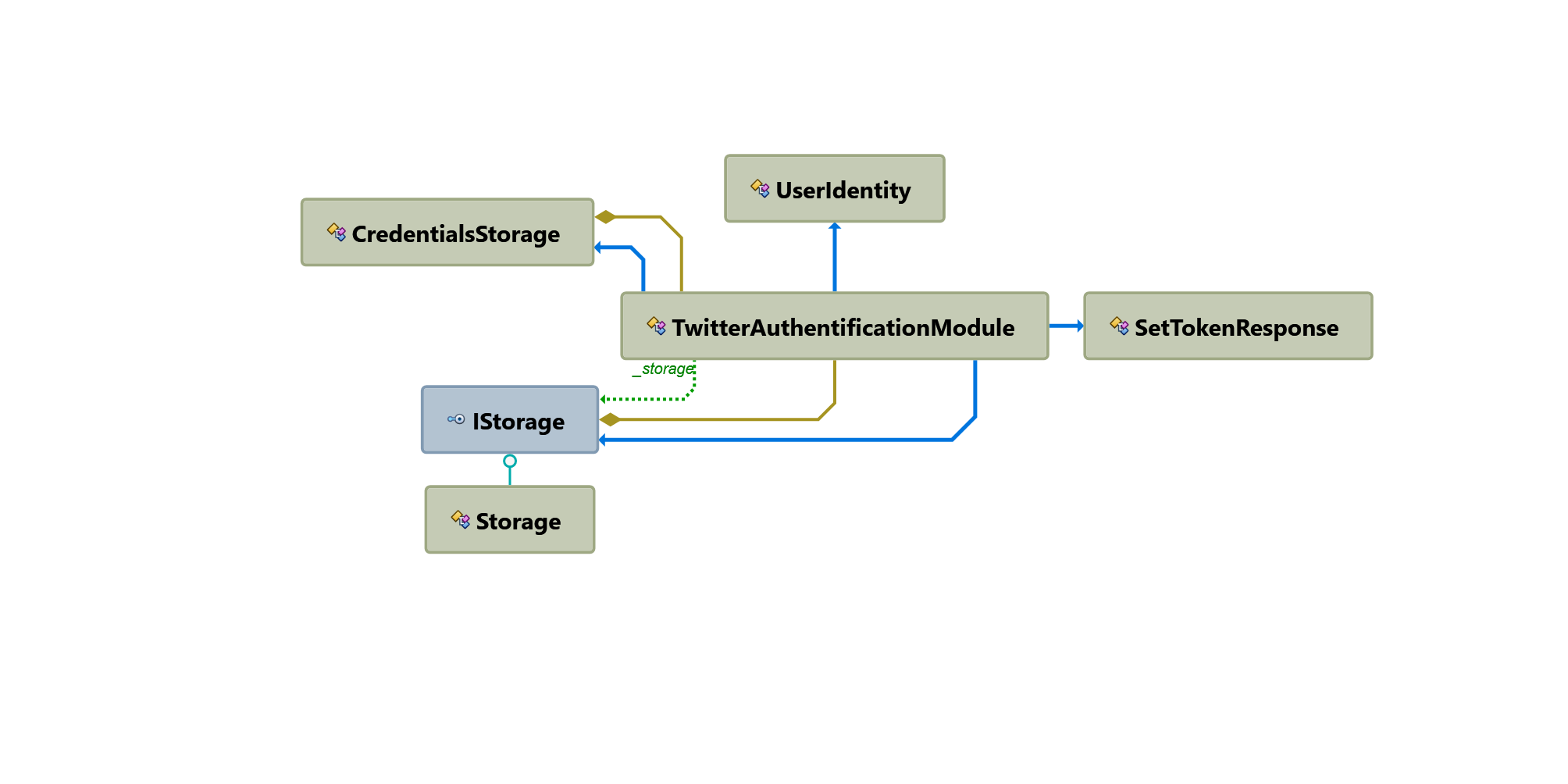


**Рис. 3.2.1 Схема классов реализующих аутентификацию**

3.2 Реализация авторизации приложения в Twitter

После регистрации аккаунта в системе пользователю предлагается предоставить к первой учётной записи в Twitter. Для этого на сервер со стороны клиентского приложения посылается запрос на получение уникальной ссылки для перехода на страницу авторизации Twitter-а, которая генерируется на сервере с использованием двух ключей (consumer key и consumer key secret) предоставляемыми разработчику при регистрации своего приложения в Twitter. Далее клиентское приложение осуществляет переход по полученной ссылке и в случае положительного решения пользователя по предоставлению доступа к своей учётной записи происходит перенаправление к клиенту, где в качестве параметров предаются промежуточные токены авторизации и строка подтверждения. Получив эти данные клиент формирует новый запрос к серверу в котором помимо полученных от Twitter-а данных указывает свой токен авторизации и адрес электронной почты от учётной записи в системе, к которому он желает привязать учётные данные Twitter-а. На стороне сервера проверяется, имеет ли пользователь с данным токеном авторизации изменять учётные данный с данным адресом электронной почты. Если проверка пройдена успешно, то генерируется запрос на получения пользовательских токенов авторизации в Twitter-е. В ответ на который сервер Twitter-а присылает соответствующие токены, которые добавляются в базу данных.

На рис. 3.2.2 представлена схема классов, осуществляющих фильтрацию контента.

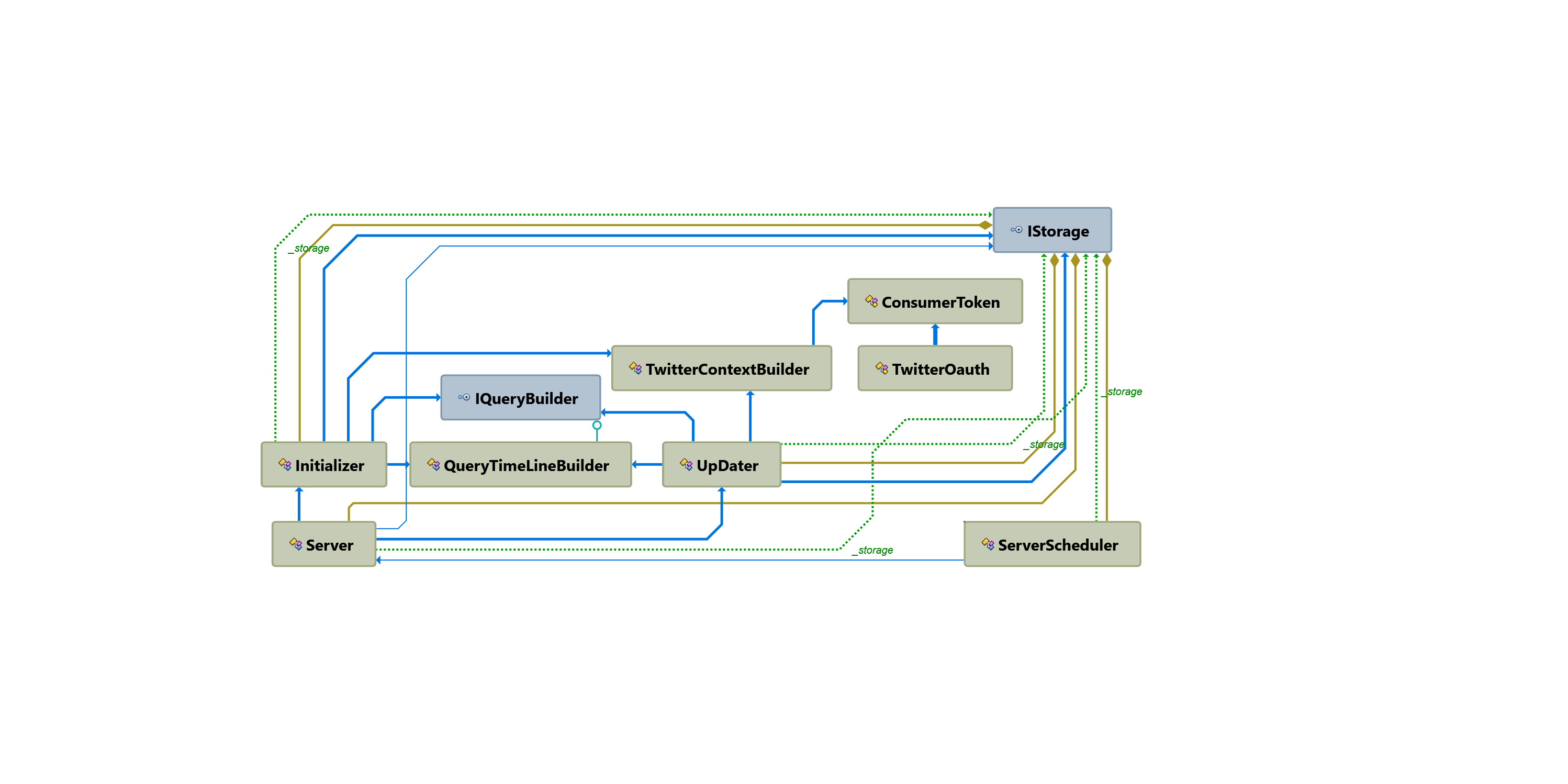


**Рис. 3.2.2 Схема классов реализующих фильтрацию**

3.3 Получение и сохранение коротких сообщений

Получение и сохранение коротких сообщений осуществляется по расписанию. Экземпляр класса ServerScheduler запускает метод Run() у экземпляра класса Server через заданные промежутки времени. При этом сервер выполняет две функции: инициализация аккаунтов и обновление ленты пользователя. Класс Initializer предоставляет функционал для первичной загрузки ленты сообщений заданного пользователя, а класс Updater предоставляет функционал для добавления новых сообщений в ленту. Иными словами Updater добавляет сообщения в начало ленты, а Initializer в конец ленты. Функционирование этих двух классов сводится к инициализации контекста сообщений и с его использованием инициализации очереди сообщений, выбора их из очереди.

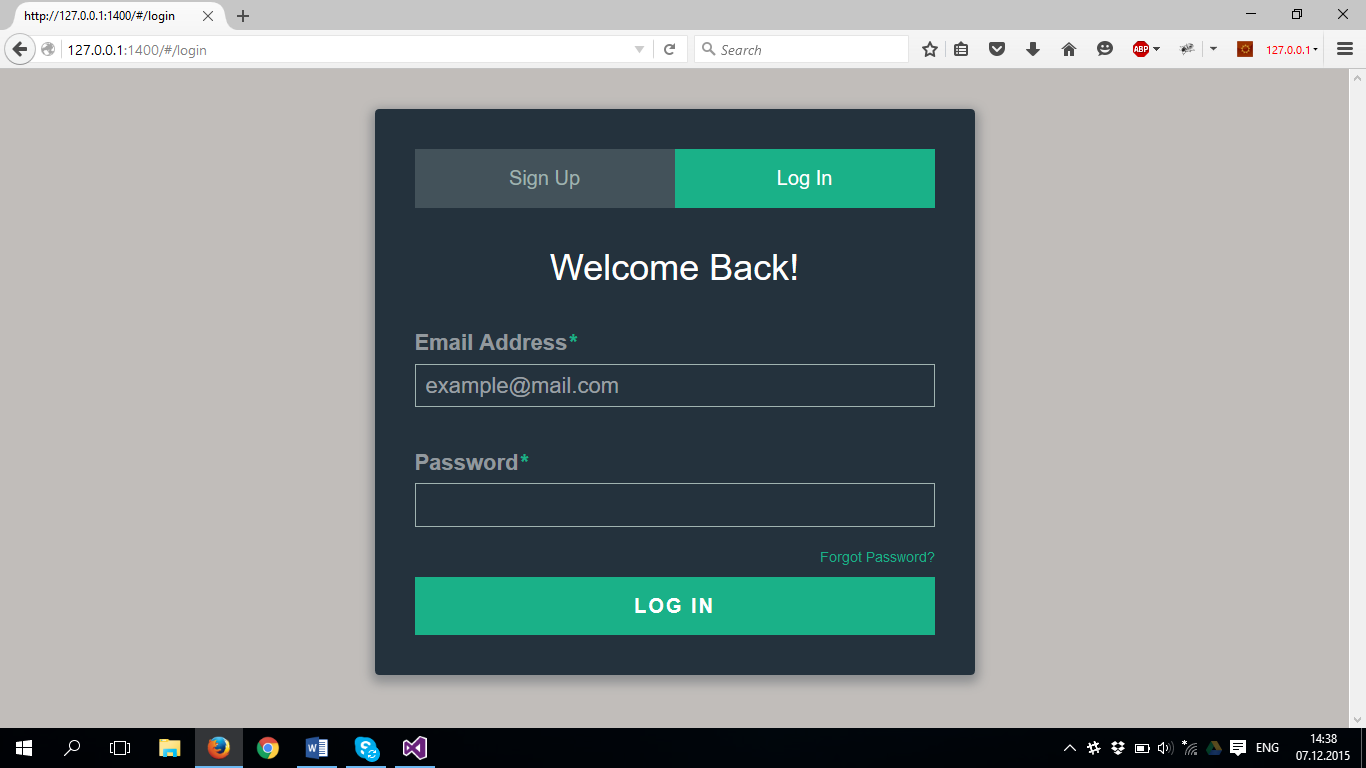
На рис. 3.3.1 представлена схема классов, реализующих получение и сохранение сообщений.



**Рис. 3.3.1 Схема классов реализующих получение и сохранение сообщений**

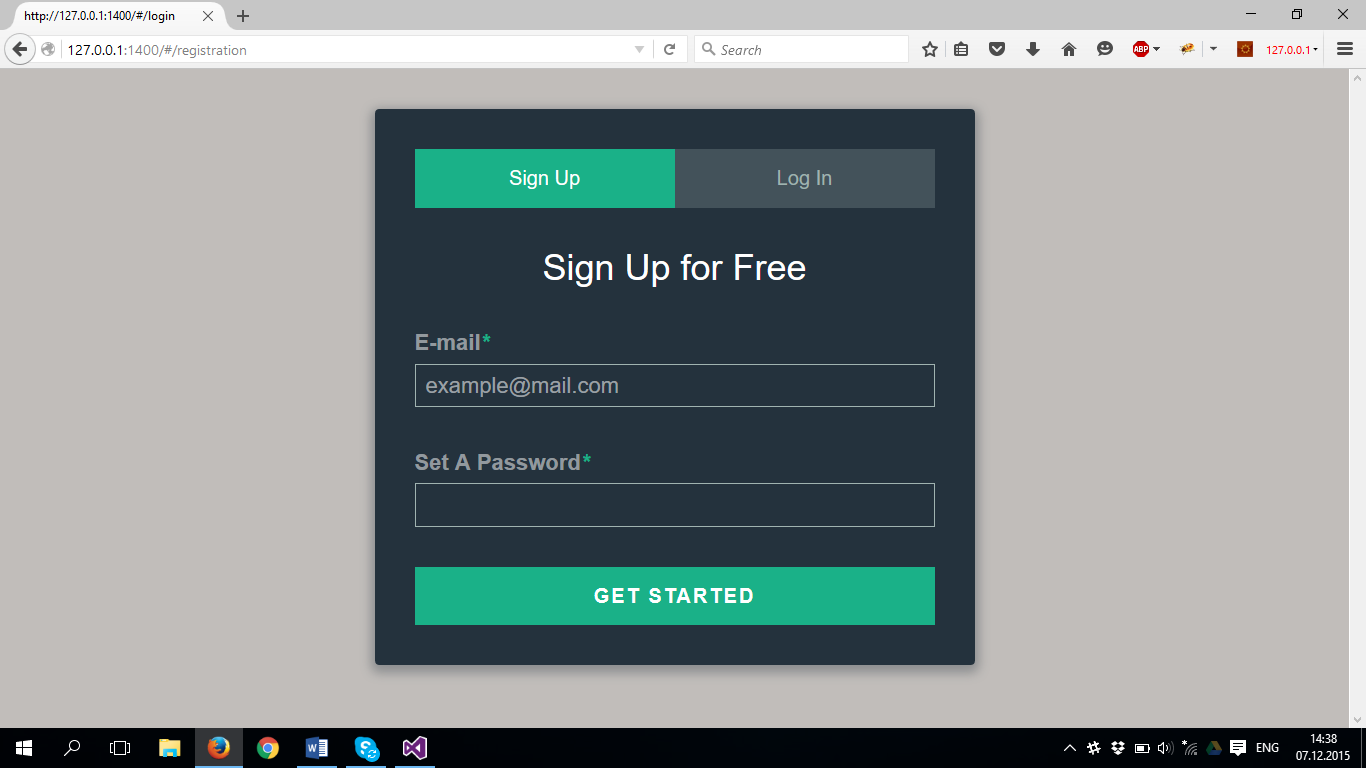
3.4 Реализация пользовательского интерфейса

Пользовательский интерфейс представляе тобой отдельный программынй модуль – сайт, взаимодействующий с серверов посредсвам http-запросов. При входе пользователя на сайт происходит проверка сookies пользователя на наличие учётных данных, в случае их отсутствия происходит перенаправление на страницу входа в систему. Где пользователь должен ввести адрес электронной почты и пароль (рис. 3.4.2).



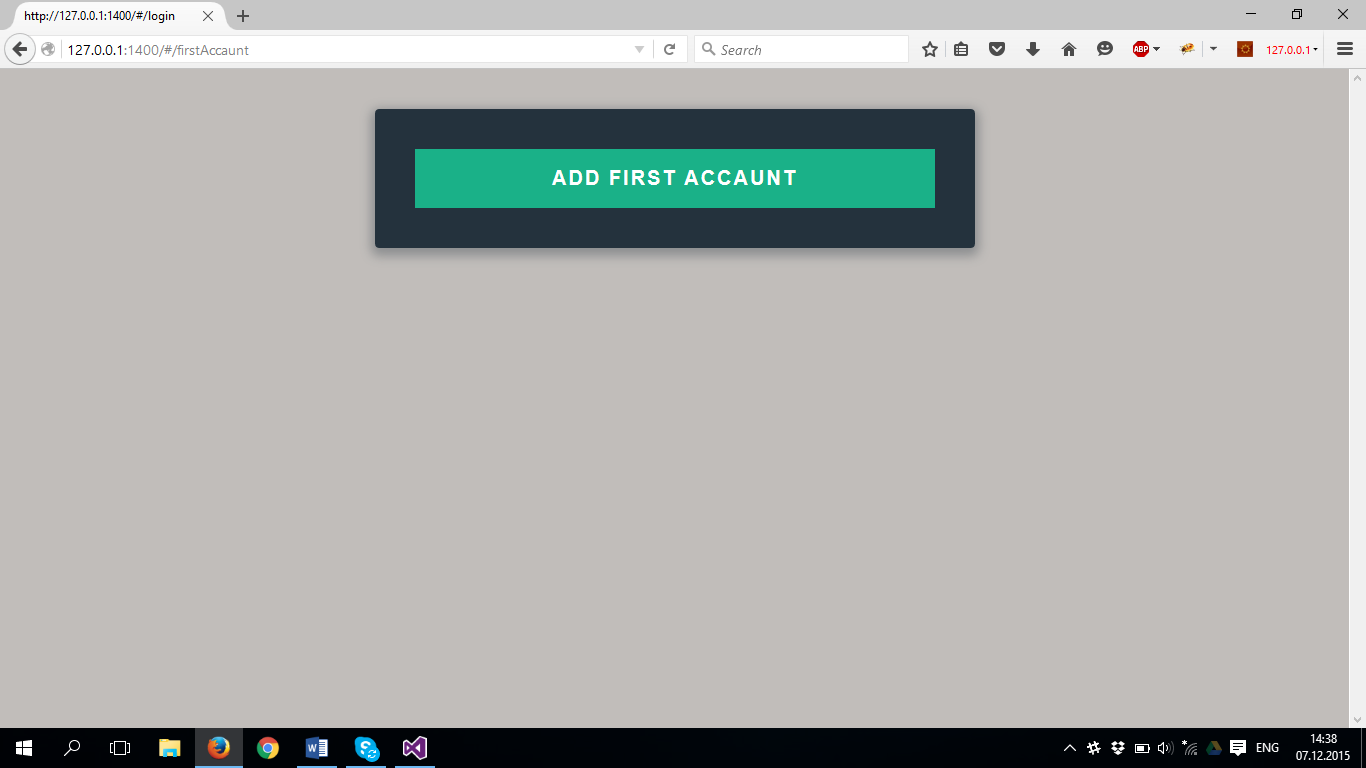
**Рис. 3.4.1 Страница входа в систему**

В случае, если пользователь не имеет аккаунта, то он может вызвать страницу регистрации нажав на кнопку “Sign Up” (рис. 3.4.2).



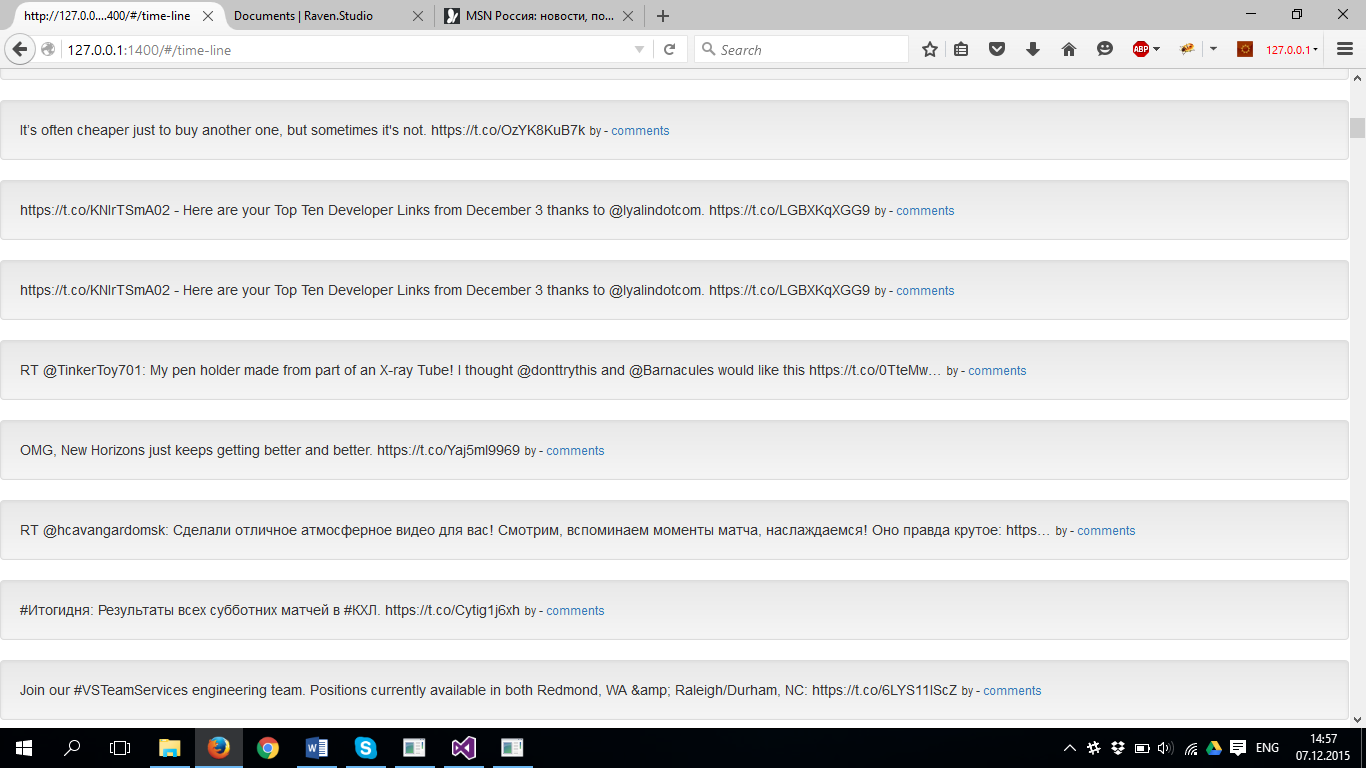
**Рис. 3.4.2 Страница регистрации**

После успешной регистрации пользователю предлагается привязать к своей учётной записи первый аккаунт (рис. 3.4.3).



**Рис. 3.4.3 Добавление первого аккаунта**

После успешного добавления первого аккаунта, а также после успешного входа в систему поьзователь перенаправляется на страницу, где отображаются его лента коротких сообщений, при этом после прокрутки сообщения загружаются динамически (рис. 3.4.4).



**Рис. 3.4.4 Добавление первого аккаунта**

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы было реализовано приложение с графическим пользовательским интерфейсом. Данное приложение является клиентом для сервиса Twitter и предоставляет фунционал, по блокировке рекламы.

В процессе выполнения данной работы были изучены технологии работы с сетью на платформе .Net, современные способы реализации пользовательских интерфейсов. При разработке функционала по опбмену данными с сервисом Twitter были получены навыки использования API, были изученные рапространнённые ошибки и способы их устранения. При реализации системы хранения были изучены подходы к хранению данных на примере нереляционной документо-ориентированной базы данных RavenDb.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Twitter [Электронный ресурс] – Режим доступа:

<https://about.twitter.com/company> - Дата доступа: 24.10.2015

1. Twitter [Электронный ресурс] – Режим доступа:

<https://dev.twitter.com/streaming/overview> - Дата доступа: 03.11.2015

1. OAuth Community Site [Электронный ресурс] – Режим доступа:

<http://oauth.net/> - Дата доступа: 08.10.2015