Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Саратовский государственный технический университет

имени Гагарина Ю.А.»

Факультет Международный факультет прикладных информационных технологий

Направление Информатика и вычислительная техника

Кафедра Информационные системы и технологии

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

«Разработка библиотеки для генерации семантических данных»

Выполнена студентом группы

м1-ИВЧТ21 Апсаликовым М.Ю.

Руководитель работы к.ф.-м.н. доцент кафедры ИСТ Вагарина Н.С.

Допущен к защите

Протокол № \_\_\_\_\_\_\_\_\_\_ от \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_\_г.

Зав. кафедрой,

руководитель магистерской программы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.А. Сытник

Саратов 2016

# Содержание

[Содержание 2](#_Toc452455239)

[Введение 4](#_Toc452455240)

[1 Аналогичные программные продукты 7](#_Toc452455241)

[2 Теоретическая часть 16](#_Toc452455242)

[2.1 Основные понятия Семантического Веба 16](#_Toc452455243)

[2.2 Краткое описание техник, практик и методологий разработки 21](#_Toc452455244)

[3 Обоснование актуальности разработки фреймворка 27](#_Toc452455245)

[3.1 Описание текущего состояния Web 2.0 27](#_Toc452455246)

[3.2 Описание текущего состояния Web 3.0 30](#_Toc452455247)

[3.3 Постановка проблемы развития Web 3.0 31](#_Toc452455248)

[3.4 Пути решения проблемы развития Web 3.0 32](#_Toc452455249)

[3.5 Выбранный путь решения 33](#_Toc452455250)

[4 Разработка библиотеки 36](#_Toc452455251)

[4.1 Использованные программные и языковые средства 36](#_Toc452455252)

[4.2 Описание архитектуры 38](#_Toc452455253)

[4.3 Описание модуля Репозиторий 41](#_Toc452455254)

[4.4 Описание модуля Валидатор 45](#_Toc452455255)

[4.5 Описание модуля Контроллер 46](#_Toc452455256)

[4.6 Описание модуля Семантический контроллер 48](#_Toc452455257)

[4.7 Инверсия зависимостей в библиотеке 50](#_Toc452455258)

[4.8 Производительность 51](#_Toc452455259)

[4.9 Тестирование 53](#_Toc452455260)

[5 Разработка демонстрационного приложения 56](#_Toc452455261)

[5.1 Использованные программные и языковые средства 56](#_Toc452455262)

[5.2 Описание архитектуры 57](#_Toc452455263)

[5.3 Описание логики 57](#_Toc452455264)

[5.4 Описание контроллеров 57](#_Toc452455265)

[5.5 Описание семантических сервисов 57](#_Toc452455266)

[5.6 Краткое описание клиентской части приложения 57](#_Toc452455267)

[Заключение 58](#_Toc452455268)

[Список использованных источников 61](#_Toc452455269)

# Введение

В настоящее время объем информации в Web таков, что эффективный поиск и обработка ее становятся все более затруднительными. В связи с этим делаются попытки поиска новых методов и подходов к созданию, предоставлению и обработке информации. Один из основных подходов на настоящий момент для решения этой задачи – применение технологий Семантического Веба (Semantic Web). Концепция Семантического Веба была принята и продвигается Консорциумом W3C (World Wide Web Consortium), разрабатывающим и внедряющим технологические стандарты для сети Интернет. Основным из результатов Консорциума по продвижению идеи Семантического Веба является стандартизация ключевой технологии семантического описания ресурсов RDF (Resourse Definition Framework). Развитие RDF началось в девяностых годах. Первая официальная спецификация была опубликована консорциумом W3C в 1999 году. Новая спецификация появилась в 2004 году. Концепции и технологии Семантического Веб нашли свое применение во некоторых отраслях, например в таких как образование[1] и наука.

Однако, в силу ресурсозатратности разработки семантических приложений, в коммерческой среде гениальная идея Семантического Веба почти не используется[2]. В связи с этим представляется перспективным создание фреймворка, который упростит создание таких приложений, сделав их привлекательными для бизнеса. Основная задача фреймворка в том, чтобы автоматизировать создание RDF-графов из обычного пользовательского кода.

На сегодняшний день аналогичных продуктов на рынке нет. Это связано с рядом причин. Во-первых, надо признать, что технологии Семантического Веб еще недостаточно распространены. Во-вторых, генерация данных из кода на основе рефлексии и аналогичных ей механизмов – это технология относительно новая. В-третьих, не все языки программирования могут позволить себе создание такого фреймворка.

Возможность создания такого фреймворка есть на платформе .NET средствами языка C#. Более того, именно этот язык используется при разработке коммерческих Web-приложений наиболее часто. На данном языке написаны такие современные фреймворки как ASP.NET и EntityFramework. Именно они и будут использованы при создании разрабатываемого семантического фреймворка.

Актуальность создания фреймворка обусловлена тем, чтобы предоставить коммерческим разработчикам возможность использовать семантические технологии в своих программных продуктах с минимальными трудозатратами.

Целью данной работы является разработка фреймворка построения Web-приложений для автоматизированной генерации семантических данных. Для достижения этой цели было необходимо решить следующие задачи:

* изучить основные понятия и подходы технологии Семантического Веба;
* проанализировать рынок аналогичных программных продуктов;
* выполнить анализ требований к фреймворку;
* выбрать программные и языковые средства для разработки;
* организовать процесс разработки;
* разработать фреймворк;
* разработать демонстрационное приложение, использующее фреймворк;
* выполнить отладку, тестирование и оптимизацию созданного фреймворка;
* выполнить отладку, тестирование и оптимизацию созданного демонстрационного приложения.

Объектом исследования являются технологии Семантического Веба. Предмет исследований – фреймворки для построения приложений на .NET, включая их возможности автоматической генерации RDF данных.

В качестве методов исследования использовались методы теоретического исследования, такие как изучение литературы, теоретический анализ, синтез, формализация, мысленное моделирование, сравнение.

Структурно работа состоит из введения, заключения и пяти глав. В первой главе производится анализ аналогичных программных продуктов. Вторая часть является теоретической. Она состоит из двух подглав: в первой рассматриваются основные понятия Cемантического Веб, во второй – описание методологий разработки. В третьей главе обосновывается актуальность разработки фреймворка. Глава состоит из пяти подглав, среди которых описания текущего состояния Web 2.0 и Web 3.0, постановка проблемы развития Web 3.0, описание путей ее решения и постановка задачи для выбранного пути решения. В четвертой главе описан процесс разработки самого программного продукта. Глава состоит из девяти подглав, в которых описаны использованные программные средства, архитектура, модули Репозиторий, Валидатор, Контроллер и Семантический контроллер, объяснена реализация инверсии зависимостей, проанализирована производительность и разъяснен процесс тестирования. Последняя глава посвящена разработке демонстрационного приложения и состоит из шести подглав, среди которых описания программных и языковых средств, архитектуры, логики, контроллеров, семантических сервисов и клиентской части.

Результаты работы докладывались и опубликованы в научной литературе, а именно в вестнике СГТУ за 2015 год[3] и в журнале «Наука, техника и образование»[4].

# Аналогичные программные продукты

Современный рынок программного обеспечения предлагает колоссальный по объему список программных продуктов. Для того, чтобы информационное решение было успешным, перед его разработкой обычно проводится бизнес-анализ его востребованности. Одной из важнейших составляющих этого анализа является исследование конкурирующих программных проектов. Цель этого исследования – приобретение понимания положительных и отрицательных сторон существующих аналогов. Программные продукты несовершенны, и, выявив наиболее критичные для современного пользователя отрицательные стороны, можно выдвинуть некоторые идеи, реализовав которые можно привлечь потенциального потребителя. Этот факт, бесспорно, является главной задачей анализа конкурирующий программных продуктов, который выполнен в данной главе магистерской диссертации.

Важно также не забывать и про положительные возможности конкурирующего программного обеспечения. Проводя анализ положительных сторон аналогичных проектов, следует знать, что они приобрели свою популярность за счет этих свойств. Здесь нужно помнить о нескольких вещах. Если при разработке опустить реализацию положительной возможности конкурирующего продукта, то можно получить такой эффект, когда пользователи продолжат использовать продукт-аналог, даже не взглянув на разрабатываемый. Также, если усовершенствовать эту функциональность, то есть вероятность, что часть пользователей откажется от решения конкурентов и отдаст свое предпочтение продукту с улучшенной функциональностью.

Анализ положительных сторон рынка программных решений, также может принести пользу (в случае с продуктами с открытым исходным кодом) тем, что есть возможность подсмотреть как именно реализуются те или иные функциональности проектов. Это поможет избежать нерациональных алгоритмов, ненадежной архитектуры и большого количества других неприятностей. Избегание вышеизложенных проблем значительно экономит ресурсы, затрачиваемые на разработку и поддержку.

В любом случае, анализ рынка современного программного обеспечения позволяет с достаточной точностью определить приоритеты, а именно: продукт с какой функциональностью будет востребован.

Взгянув на продукт-конкурент, есть возможность предположить, какая аудитория будет использовать разрабатываемый продукт. Например, при разработке JavaScript-библиотеки для рисования графов, можно с полной уверенностью предположить, что использовать её будут, в основном, Web-программисты, специализирующиеся на разработке интерфейсов (Front-End). Используя все эти данные, при разработке библиотеки классов, можно адаптироваться под сообщество и предоставить им такой интерфейс, к которому они привыкли. Это увеличит лояльность аудитории к продукту. К примеру, в среде .NET-разрабочиков используется конвекция наименования асинхронных методов с суффиксом -Async. Использование библиотеки, не подчиняющейся этой конвенкции будет непривычно большинству профессиональных разработчиков на языке C#. Вероятно, они отметят это, и в сообществе программный продукт незначительно упадет в популярности.

В данной работе исследуются аналогичные программные продукты к фреймворку генерации семантических данных для платформы .NET Framework. Откровенно говоря, несмотря на все разнообразие информационного контента, полных аналогов разрабатываемого продукта не существует как для самого .NET, так и вообще на рынке. На это есть ряд причин.

Во-первых, самая очевидная причина – это общая нераспространенность семантических технологий в мире. К сожалению, Web 3.0 не настолько быстро развивается, как хотелось бы, по причине того, что количество людей и организаций, заинтересованных в его развитии материально достаточно мало. В Web 3.0 не существует инструментов для быстрого и эффективного продвижения сайтов. Он не популярен среди обычных рядовых конечных пользователей. Более того, даже многие профессиональные Web-разработчики не слышали о семантическом Web вообще.

Во-вторых, генерация каких-либо данных из кода на основе рефлексии или аналогичных ей механизмов – это технология относительно новая. Десять лет назад никто не мог представить, что в промышленном программировании SQL код сможет генерироваться «на лету» из обычного кода одного из объектно-ориентированных языков[5]. Более того, даже сегодня, далеко не все платформы и языки программирования могут себе это позволить. На платформе .NET на момент написания данной работы самая развитая инфраструктура для генерации кода на других языках. Сам механизм называется Expression trees (деревья выражений) [6]. Суть его в том, что при написании кода, он не компилируется, а преобразуется в дерево выражений, которое может быть использовано для анализа, изменения, динамической компиляции, преобразования в другой язык программирования, в том числе SQL. Возможность активно используется библиотеками, такими как Entity Framework[7].

В-третьих, платформа .NET в основном ориентирована на коммерческий сектор разработки программных решений [8]. Крупные компании не особо заинтересованы в использовании семантических технологий. Они зависимы от мнения инвесторов, которые обычно отрицательно относятся к новым и непроверенным технологиям. Не без оснований считается, что это рискованные вложения. Следовательно, несмотря на благоприятную инфраструктуру, вероятность встретить под .NET аналогичную библиотеку значительно снижена.

Совокупность вышеуказанных причин полностью объясняет отсутствие аналогичного программного продукта под платформу .NET.

В конечном итоге, было решено проводить анализ семантических ORM-библиотек под другие платформы. Перечисленные продукты не являются полноценными аналогами разрабатываемого приложения, так как они не работают с источниками сущностей. В разрабатываемом приложении источником данных является база данных, доступ к которой осуществляется через EntityFramework. Также, анализируемые библиотеки, в отличие от фреймворка, разработанного в рамках текущей магистерской диссертации, не предоставляют разработчикам полноценную архитектуру. При проведении анализа необходимо учесть такие параметры как:

* особенности библиотеки;
* распространенность библиотеки;
* наличие архитектуры;
* особенности языка, на котором написана библиотека;
* наличие документации и поддержки;
* сложность использования.

Всего было найдено три семантических ORM-библиотеки. Проведем анализ первой из них. Она называется RDF-mapper[9]. На момент написания статьи имеет 5000 скачиваний и всего три релиза.

Сильные стороны:

* кроссплатформенность;
* открытый исходный код;
* отложенное выполнение;
* использование Rails методов (интерфейс представлен в стандартном для Ruby стиле).

Слабые стороны:

* почти полное отсутствие документации;
* малое покрытие Unit-тестами (10%);
* SPARQL-адаптер реализован на самом базовом уровне (только чтение);
* нет поддержки REST;
* поддерживается только две сериализации RDF;
* полное отсутствие архитектуры.

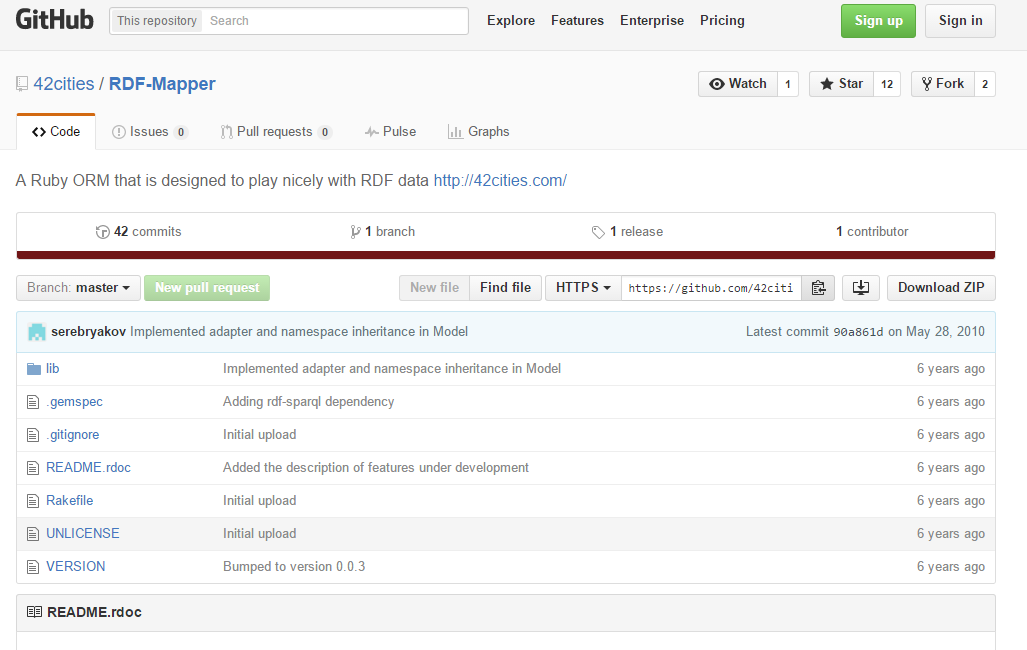
Очевидно, что данная библиотека недоработана. Использовать ее в коммерческих целях желающих нецелесообразно. Нецелесообразность использования в коммерческих целях также подтверждается количеством скачиваний библиотеки, которое ясно говорит о том, что она имеет критически низкую популярность. Как было сказано ранее, бизнес-пользователи избегают непроверенных технологий. Это осложняется тем, что продукт покрыт Unit-тестами всего на 10%[10]. Это очень низкий показатель. Сам программный продукт позиционируется как легковесная библиотека для преобразования данных в семантическую форму, однако, даже с этой задачей она не справляется успешно с точки зрения промышленной разработки. Возможно, она разовьется в будущем, однако сегодня ее можно списать со счетов.

Рисунок 1 – Репозиторий GitHub проекта RDF-mapper на языке программирования Ruby.

Также стоит отметить, что библиотека написана на Ruby, который не сильно популярен в промышленной среде в силу своих особенностей. Сам язык создавался как интерпретируемое средство создания приложений в Web 2.0 и содержит мощные средства для MVC-архитектуры. Однако, его слабой стороной является производительность. Это логично, так как язык интерпретируемый. Также, большими рисками для проектов на Ruby является низкое количество сторонних библиотек.

 Второе рассматриваемое программное решение – библиотека SurfRDF[11], написанная на другом интерпретируемом языке программирования Python.

Рисунок 2 – Пример простейшего FOAF проекта с использованием SurfRDF (язык программирования Python).

Сильные стороны:

* кроссплатформенность;
* открытый исходный код;
* хорошая документация;
* проект относительно активно развивается;
* поддержка пяти сериализаций RDF.

Слабые стороны:

* слабая типизация сущностей;
* сложная и запутанная логика;
* низкая производительность;
* отсутствие Unit-тестов.

Библиотека, написанная на Python, выглядит более конкурентноспособной, чем аналог на Ruby. В продукт вложено немалое количество усилий свободных разработчиков. Об этом свидетельствует относительно хорошая и полная документация с примерами. Каждый метод, модуль и пространство имен описаны в документации четким и понятным английским языком. Отдельно стоит отметить то, что в библиотекой поддерживается пять диалектов RDF.

Несмотря на положительные стороны библиотеки, она имеет ряд минусов. Большая часть из них связана с языком, на котором она написана. Python – это интерпретируемый язык с низким уровнем развития метапрограммирования. Изначально, язык разрабатывался как скриптовый язык для управления операционными системами, в частности UNIX. Для разработчика использование данного языка в целях разработки семантической библиотеки означает очень низкую производительность, а так же невозможность использовать механизмы рефлексии в удобном виде.

Теперь перейдем к обзору библиотеки, написанной под платформу, которая является главным конкурентом .NET на рынке коммерческих приложений. Нетрудно догадаться, что это платформа Java. Библиотека носит название RDFBeans[12] и имеет открытый исходный код, который расположен на Sourceforge. Также как и RDF-mapper, на момент анализа, библиотека не привлекла большого внимания свободных разработчиков.

Сильные стороны:

* кроссплатформенность;
* открытый исходный код;
* отсутствие развития (последний коммит был сделан в 2013 году);
* максимально использует особенности и механизмы языка Java.

Слабые стороны:

* отсутствие документации;
* отсутствие Unit-тестов;
* отсутствие архитектуры.

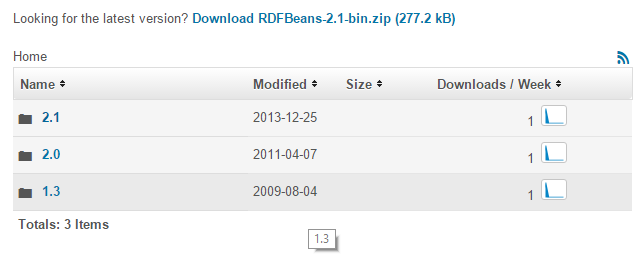
Несмотря на то, что продукт не развивается и заброшен, его не стоит оставлять без внимания. Более того, при разработке фреймворка под .NET в первую очередь необходимо учесть опыт именно этого программного продукта. В нем можно увидеть следующие важные архитектурные моменты.

Рисунок 3 – Даты последних коммитов в репозиторий проекта RDFBeans

(язык программирования Java).

Во-первых, использование аннотаций для получения недостающих данных при маппинге. В .NET вместо аннотаций используются аттрибуты, но общий смысл остается тем же. Во-вторых, независимость сущностей от каких-либо интерфейсов и абстрактных классов. Она достигается при помощи метапрограммирования, в частности механизмом рефлексии. Это дает возможность использовать любые сущности, даже если они лежат в скомпилированных библиотеках классов. В-третьих, доступ к семантическим данным реализован через интерфейсы. Это означает, что даже если библиотека не реализует какую-то сериализацию, то ее может написать сам программист.

Произведя анализ, можно сделать вывод, что библиотек для автоматизации работы с семантическими данными сегодня немного, но они есть. Библиотек, полностью автоматизирующих процесс синхронизации данных начиная с Web 2.0 представления до семантических сервисов через логику и базу данных на сегодняшний день нет. Технически, это возможно, однако в силу ранее описанных причин такая библиотека пока не создана. Такие решения существуют в виде конкретных ручных реализаций на конкретных проектах. По мере набора популярности семантическим Web, также появятся решения в виде шаблонов проектирования.

# Теоретическая часть

## Основные понятия Семантического Веба

Рассмотрим основные понятия Семантического Веб, также известного как Web 3.0. Создатель Всемирной паутины Тим Бернерс-Ли в далеком 2003 году определил несколько этапов развития WWW[13]:

* Web 1.0;
* Web 2.0;
* Web 3.0;
* Web 4.0.

Первый этап развития WWW – это Web 1.0. Он длился с самого появления Всемирной паутины до 2001 года. Все сайты представляли собой статическое содержимое, которое создавалось, удалялось и модифицировалось разработчиками самого сайта. Разработка производилась при помощи преимущественно табличной верстки, также можно было встретить сайты, состоящие целиком из обычного текста. Это время также характеризуется «войной браузеров», последствия которой Web-разработчики всего мира стараются исправить и сегодня. Технологически, сайт Web 1.0 представляет собой набор файлов, которые отправляются клиенту при запросе.

На смену статическим сайтам с 2001 года начали приходить динамические. Главное отличие динамических сайтов в том, что они могут изменять свое собственное содержимое. Этот факт является критическим преимуществом, так как предоставляет колоссальные возможности как для бизнеса, так и для обычных людей. Динамические сайты – это эпоха Web 2.0, которая существует на текущий момент [14]. Технологически, сайт Web 2.0 представляет собой Web-приложение, которое размещается на приложении веб-сервере. Веб-сервер в свою очередь отправляет содержимое клиенту, предварительно его обрабатывая.

Web 2.0 имеет колоссальную популярность, развивается большими шагами, однако имеет большую проблему. Сбор, обработки и систематизация данных всех сайтов невозможна в связи с тем, что информация предоствляемая сайтами предназначается для человека, а не для машинной обработки. С вышеуказанные проблемы пытаются решить поисковые системы. Очевидно, что несмотря на то, что в разработку поисковых роботов вкладываются миллионы долларов, они решают проблему лишь частично.

Для систематизации информации во Всемирной паутине была предложена идея Семантического Веб[15]. Идея заключается в том, чтобы создать надстройку над Web 2.0 технологиями, которая позволит считать данные с сайта в унифицированном формате[16]. Иными словами, программа, заходя на сайт получит документ с описанием содержимого, написанный на языке, который она может проанализировать и сделать логические выводы.

Идея Web 3.0 была реализована в виде стека технологий (технологии указаны в порядке повышения уровня абстракции) :

* URI;
* XML;
* RDF;
* RDF Schema;
* OWL;
* OWL Schema;
* SPARQL;
* WSDL.

Наиболее значимы для рассмотрения 4 технологии: URI, RDF, OWL и SPARQL.

Рассмотрим детально что такое URI. Uniform Resource Identifier – это строковое представление какого-либо ресурса. URI может быть двух типов:

* URL (Uniform Resource Locator). Представляет собой ссылку на ресурс, находящийся в сети Интернет;
* URN (Uniform Resource Name). Представляет собой любую строку, которой можно однозначно идентифицировать конкретный объект реального мира.

Из описания вариантов выражения URI можно сделать вывод, что технологии Семантического Веб работают как с сущностями, находящимися в сети Интернет, так и находящимися вне него. Обычно URI представляет собой строку вида:

{Схема}://{Доменное имя}:{Порт}{Путь}?{Запрос} #{Фрагмент}.

Где, Схема – необязательная часть URI, которая представляет собой имя схемы, которая указывает как обрабатывать его в приложениях. Например: http, https, ftp.

Доменное имя – это имя сайта в DNS. Является обязательной для URL, но не для URN. Пример: eu.battle.net.

Порт – номер порта, который используется приложениями для подключения к ресурсу. Часть является необязательной и используется только в URL. Представляет собой число от 0 до 65535.

Path – основная часть URI, которая означает путь к ресурсу. Обычно организован иерархически. Например: /resources/images/.

Запрос - необязательная часть URI необходима для передачи параметров, например: ?id=184367374.

Фрагмент – часть конкретного ресурса. В насыщенных Web 2.0 одностраничных приложениях (Single Page Application) используется вместо Запроса. Пример: #yellowonly.

При описании идеи Семантического Веб было указано, что идея предусматривает считывание данных с сервера в унифицированном формате. В стеке технологий Web 3.0 этот унифицированный формат – язык RDF (Resource Description Framework). Абстрактно, RDF представляет собой неотсортированную коллекцию утверждений, которые называются триплеты[17]. Триплет, в свою очередь, представляет собой модель «субъект – предикат - объект», где субъект – сущность, которую необходимо описать. Предикат – свойство субъекта, а объект – значение этого свойства. В качестве примера можно привести кодировку утверждения «Металлическая кружка». Здесь объект – кружка, предикат – материал, объект – металл.

Для лучшего понимания с точки зрения программирования можно привести аналогию с классом в объектно-ориентированном программировании. Возьмем класс Контрагента страховой компании, у которого есть поля Фамилия, Имя и Отчество (см. Рисунок 4).

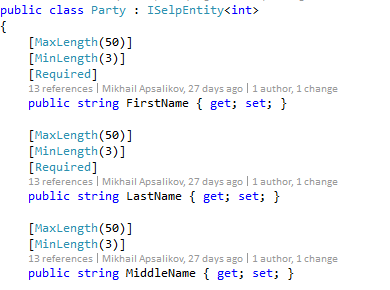


Рисунок 4 – Класс Контрагента страховой компании.

Для полного описания сущности в формате RDF здесь будет три триплета. Субъектом в каждом из них является Контрагент, предикатами соответственно Фамилия, Имя и Отчество. Объектами триплетов будут значения, содержащиеся в сущности на момент выполнения. Итого триплеты будут иметь вид как на Рисунке 5.

Триплеты могут быть сериализованы по-разному. Способ их сериализации называется диалектом (или сериализацией). Самыми распространенными диалектами являются RDF/XML [18], Turtle [19], Notation3 [20] и N-Triples, причем только RDF/XML является официально поддерживаемым консорциумом W3C.

Набор RDF триплетов, который полностью описывает какую-либо предметную область называется онтологией. В стеке технологий Web 3.0 для упраавления онтологиями используется язык OWL (Web Ontology Language). Именно OWL придает семантическим данным смысл и связывает их с предметной областью. OWL построен полностью на RDF и привносит в него такие понятия как класс, свойство, операция, метакласс и многие другие.

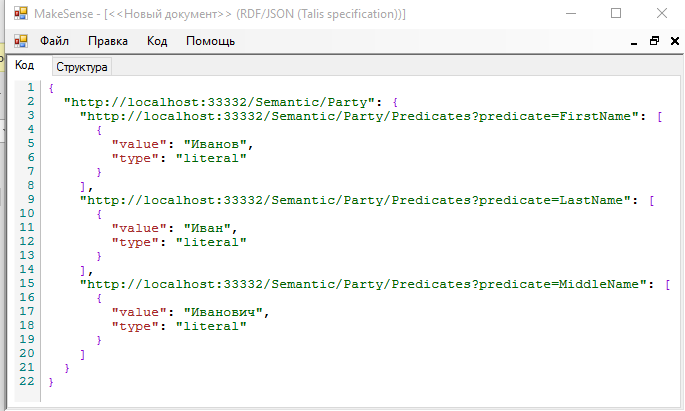


Рисунок 5 – RDF/JSON представление Контрагента страховой компании.

OWL представляет собой URI из особого пространства имен, которые, во-первых, могут быть использованы клиентским приложением Semantic Reasoner для получения логических выводов. Во-вторых, при ручном создании онтологии можно избежать логических ошибок, явно указывая какая сущность что собой представляет в рамках предметной области.

Следующим и последним рассматриваемым уровнем абстракции технологического стека Web 3.0 является SPARQL. Это язык запросов, основанный на SQL, который связывает пользователя с онтологией и позволяет ему получать данные которые ему нужны.

## Краткое описание техник, практик и методологий разработки

Современная разработка программного обеспечения стремится к тому, чтобы минимизировать затраты ресурсов, затрачиваемых на создание и поддержку программных продуктов. Под ресурсами в данном случае понимается цена и время. Также значительным фактором является минимизация количества ошибок и повышения качества продукта в целом.

Для управления вышеизложенными параметрами в коммерческих проектах применяются методологии разработки программного обеспечения. В общем случае, методология – это система принципов, совокупность понятий, идей, способов и средств, определяющих стиль разработки программного обеспечения[21]. Методология – это реализация старндарта. Сами стандарты указывают на то, как должно быть, методология оставляет свободу выбора и адаптации.

Конкретные процессы разработки программного обеспечения реализуются через выбранную методологию. Именно она определяет как именно выполняется разработка. Существует большое количество методологий программного обеспечения, таких как каскадная, итеративная, спиральная. Все они имеют свои положительные и отрицательные стороны. В большинстве коммерческих проектов используется группа методологий Agile (гибкие методологии разработки). Эта группа методологий получила широкое распространение в связи с тем, что именно методологии этой группы позволяют быстрее всего реагировать на изменение требований с минимальными потерями времени, денег (в случае, если проект коммерческий) и потерями качества. Также эти методологии обладают дополнительными положительными сторонами, которые являются следствием гибкости разработки.

Группа методологий Agile имеет манифест, который обьединяет все методологии внутри него [22]:

* Люди и взаимодействие важнее процессов и инструментов.
* Работающий продукт важнее исчерпывающей документации.
* Сотрудничество с заказчиком важнее согласования условий контракта.
* Готовность к изменениям важнее следования первоначальному плану.

Так как проект планируется поддерживать и постоянно дорабатывать, то очевидно, что для разработки будет выбрана одна из методологий Agile.

Самой известной методологией разработки Agile является Scrum [23]. Дословно, с английского название переводится как «толкучка». Основной акцент делается на качественном контроле за процессом разработки. Разработка поделена на итерации, которые называются спринтами. Выполнение спринта, все роли, обсужения и артефакты четко регламентированы. Их исполнение и применение контролируется специальным человеком – Scrum-мастером.

По методологии Scrum все задачи с прикрепленным к ним приоритетом складываются в специальный список задач владельцем продукта (список задач называется Backlog).

Спринты занимают 1-4 недели и являются неизменяемыми единицами выполненной работы с точки зрения владельца продукта. Спринт начинается с планирования, в рамках которого команда выбирает из Backlog-а задачи, которые будет выполнять в текущий спринт. Задачи, которые не входят в спринт целиком декомпозируются. Каждый день во время спринта команда собирается и каждый ее участник сообщает о том, что он сделал, что собирается делать и какие вопросы у него есть. Это называется ежедневный митинг, позволяет отслеживать общий прогресс команды и принимать быстрые решения вовремя в случае экстренных ситуаций. По окончании спринта для владельца продукта производится демонстрация новой функциональности рабочего продукта. После сдачи спринта проводится ретроспектива с целью просуммировать командный опыт по результатам спринта.

Положительные стороны Scrum:

* прозрачность для всех участников процесса;
* короткие спринты позволяют быстро применять изменения;
* возможность учиться на своих ошибках как команда (благодаря ретроспективам);
* возможность применять качественные решения;

Однако, у Scrum есть и отрицательные стороны, например постоянный контроль над командой. Это, во-первых, незначительно снижает мотивацию команды. Во-вторых, на процессы уходит значительное количество времени. Также Scrum вынуждает команду по результатам спринта создавать ветки. В случае обнаружения критических ошибок приходится разрабатывать продукт в нескольких ветках, а затем сливать. Это неудобно и требует дополнительных ресурсов.

Следующей методологией разработки ПО является канбан (Kanban). Принято считать, что эта методология разработки наиболее удачно подходит для проектов находящихся на стадии поддержки. Основная идея методологии – минимизация количества одновременно выполняющихся задач. Для этого создается специальная таблица c задачами, на которой видно сколько задач в текущий момент выполняет команда.

Положительные стороны Kanban:

* отсутствие документации;
* прозрачность процесса;
* отсутствие сроков;
* быстрое применение изменений;

Отрицательные стороны:

* отсутствие сроков может повлечь за собой снижение производительности команды, если команда не мотивирована;
* не подходит для разработки новых продуктов.

Рассмотрим следующую методологию разработки – Extreme Programming (XP) [24]. Данная методология предусматривает максимальное упрощение организации процесса разработки в условиях постоянно меняющихся требований. Основной целью методологии является повышение качества продукта в минимальные сроки. Общий смысл методологии в применении двенадцати практик:

* планирование процесса;
* тесное взаимодействие с заказчиком;
* общесистемные правила именования;
* простая архитектура;
* рефакторинг;
* парное программирование;
* 40-часовая рабочая неделя;
* коллективное владение кодом;
* единые стандарты кодирования;
* небольшие релизы;
* непрерывная интеграция;
* разработка через тестирование (TDD).

Процесс XP является неформальным и требует высокого уровня самодисциплины каждого их разработчиков. В XP каждый член команды считается профессиональным самомотивированным работником.

Положительные стороны XP:

* полное отсутствие как документации, так и артефактов;
* отсутствие потерь времени на процессы;
* быстрое применение изменений;
* выполнение задач быстрее, чем в рамках других методологий;
* гарантия качества выполнения задач за счет некоторых практик (TDD, парное программирование, коллективное владение кодом).

Стоит отметить, что данная методология имеет ограничения, которые одновременно и являются ее отрицательной стороной. XP применим только для небольших и средних команд. Для большой команды невозможно применить некоторые из ключевых практик и процессы на проекте превращаются в хаос. Также методология применима только для самодисциплинируемых программистов. В противном случае методологию применять бессмыслено в связи с отсутствием контроля как со стороны менеджмента, так и со стороны программиста. Эта причина является ключевым моментом в том, что самой популярной методологией является Scrum, а не XP, несмотря на то, что XP потенциально предоставляет больший уровень производительности команды.

Все вышеизложенные методологии объединяет отсутствие документации. Однако, не все методологии Agile предполагают ее отсутствие. Например, существует методология RUP (Rational Unified Process) [25]. Это методология разработки программного обеспечения, которая была разработана компанией Rational Software, с целью создать методологию, которая позволит скомбинировать положительные стороны водопадной модели и Agile. Аналогично водопаду, процесс разработки разделен на фазы, результатами которых является соответствующая документация. Однако, в данной методологии требования могут меняться на любой стадии.

Положительные стороны:

* этапы могут выполнять разные команды;
* предсказуемость;
* качество продукта;
* прозрачность процесса, которая не теряет актуальность для команд большого размера.

Отрицательные стороны:

* колоссальные потери времени на процессы.

Для разработки библиотеки генерации семантических данных была выбрана методология Extreme Programming. RUP и Scrum были отсеяны ввиду требования документации и соблюдения процессов, времени на написание которой нет. Kanban может использоваться в поддержке фреймворка, однако для его разработки рационально использовать XP. Ограничения методологии Extreme Programming на исполнителя не распространяются, в связи с тем, что команда состоит из одного человека, который достаточно мотивирован, чтобы методология работала.

Также, ключевым моментом является техника написания тестов TDD, которая входит в практики Extreme Programming. Данный подход заключается в том, чтобы сначала писать тесты, а потом уже код. Такая практика позволяет значительно повысить качество кода и снизить время на исправление ошибок ценой небольшого увеличения сроков разработки.

# Обоснование актуальности разработки фреймворка

## Описание текущего состояния Web 2.0

Web 2.0 на сегодняшний день находится на пике своей популярности. Интернет сегодня используется более чем тремя миллиардами человек по всему миру и, открывая страницы сайтов, каждый из них использует технологии Web 2.0. Самыми богатыми компаниями в мире являются те, основная деятельность которых основывается на использовании этих технологий.

Мир Web 2.0 очень динамично развивается. Каждый день в нем появляются тысячи новых Web-приложений, каждое из которых имеет свою аудиторию, выполняет определенные задачи и двигает мировое сообщество к развитию определенных предметных областей. Человечество использует Интернет в самых разных целях. Логично, что используемые при разработке технологии также динамично развиваются. Несколько лет назад большинство Web-приложений представляли собой наборы Web-страниц, оживленные с помощью JavaScript. Сегодня технологии изменились и разработка клиентских частей Web-приложений вышла на новый профессиональный уровень.

Давайте взглянем на современную веб-резработку и её стандарты. В качестве серверной части можно увидеть один из следующих веб-сервисов: Apache, IIS (Internet Information Service), Tomcat. Обычно, они выполняют приложения, написанные популярных Web-платформах.

На первом месте по использованию в качестве Web-платформы на сегодняшний день находится PHP. Он занимает 81.9% рынка[26]. Однако, стоит учитывать специфику данного языка программирования. Данный язык разрабатывался для создания домашних страниц, о чем явно говорит его название: Personal Home Page. В основном, язык в сочетании с фреймворками, такими как Joomla, WordPress, Битрикс, используется для создания простых и дешевых сайтов-визиток. PHP очень редко используется в Enterprise-проектах в силу своей ненадежности и репутации. Более того, язык имеет существенные минусы, не позволяющие построить хорошую архитектуру Web-приложения, а это немаловажный фактор использования крупных коммерческих проектах, так как это означает, что поддержка кодовой базы будет дорогостоящей.

Сайты, требующие хорошего качества разрабатываются на более серьезных платформах, таких как ASP.NET, Spring, node.js. Объединяет все эти технологии одно: они все умеют генерировать контент «на лету». Помимо генерации HTML, CSS и JavaScript вышеперечисленые программные средства умеют обрабатывать данные в формате JSON, принимая AJAX-запросы.

Второй по популярности в Web 2.0 после PHP является ASP.NET. Она занимает 16.9% рынка. Как следствие, можно сделать вывод, что это самая распространенная Web-платформа в крупных коммерческих проектах. Библиотека ASP.NET является самым мощным фреймворком для построения Web-приложений на текущий момент. Также, она предоставляет возможности программировать на языке C# в среде Visual Studio, которые являются одними из самых удобных средств разработки программного обеспечения. Платформа содержит в себе такие технологии, как ASP.NET WebForms, ASP.NET MVC и ASP.NET WebAPI. Последние две технологии являеются самыми современными на текущий момент. Для большинства запускаемых проектов приложений под платформу .NET выбирают эти библиотеки.

Несмотря на то, что формально клиентская часть Web 2.0 существовала и продолжает существовать только в трех основных технологиях: HTML, CSS и JavaScript, проанализировав клиентские части современных Web 2.0 приложений, можно увидеть, что сайты, написанные на чистом JavaScript ушли в прошлое. Хорошей практикой в современной Web-разработке является «толстый» клиент. Иными словами, часть бизнес-логики реализуется на стороне клиента с помощью JavaScript. Однако реализовать это с помощью чистого JavaScript очень ресурсозатратно. В мире разработки Front-End быстро набрали популярность клиентские JavaScript-фреймворки, которые значительно упрощают разработку.

Лидерами на рынке JavaScript-фреймворков являются Angular 1, React и Backbone. Каждый из них, обладая своими положительными и отрицательными сторонами, значительно оптимизирует затраты ресурсов на разработку «толстого» клиента. Например React позволяет быстро создавать качественные и переиспользуемые контролы. Его конкурент Angular.js позволяет в считанные минуты заставить страницу работать в режиме прототипа.

Упоминая о JavaScript-фреймворках, нельзя не рассмотреть Angular 2. Этот фреймворк не является следующей версией Angular 1, а позиционируется как новое средство разработки. На момент написания данной магистерской диссертации фреймворк не обладает большой популярностью в связи с тем, что выпущен был недавно и не успел набрать популярность. Однако, согласно прогнозам аналитиков, фреймворк займет одну из лидирующих позиций, в силу наибольшей производительности и предоставления возможности разрабатывать сайты с минимальными затратами ресурсов.

Также одной из самых современный технологий в мире разработки Web 2.0 приложений можно назвать одностраничные приложения (SPA – Single Page Application). Суть такого приложения в следующем: вместо серверного роутинга используется клиентский. Это означает, что вся логика отображения ложится на клиентскую часть приложения. Пользователю предоставляется возможность асинхронной загрузки страниц и данных через AJAX. Клиент общается с сервером через REST-сервис, что значительно экономит серверные ресурсы, так как исчезает необходимость каждый раз генерировать и отсылать разметку (этим занимается клиент). Все современные фреймворки, в том числе Angular 1, Angular 2, Backbone, React и Ember умеют создавать одностраничные приложения.

Следующей широко используемой в коммерческой разработке группой технологий является автоматизация сборки. Сюда входят препроцессоры CSS, минификаторы, шаблонизаторы, инструменты для проверки стиля кода, непрерывная интеграция и множество других технологий. Данная группа технологий позволяет значительно уменьшить затраты на развертывание программных продуктов в Web 2.0 на поставочных серверах, а также косвенно уменьшить время разработки.

Вкратце можно сказать, что технологии Web 2.0 имеют колоссальную популярность, быстро развиваются, адаптируются под требования как физических лиц, так и бизнеса.

## Описание текущего состояния Web 3.0

В мире Семантических технологий ситуация обстоит иначе. В отличие от Web 2.0 сайтов, которые рассчитаны на восприятие человеком, семантические сервисы не имеют большой популярности. Стек технологий с переменным успехом развивается инициативными группами и университетами. Рядовые пользователи и бизнес, которые составляют наибольшую часть пользователей Интернет-технологий, не заинтересованы материально в развитии Семантического Веб.

Отчасти это связано с тем, что в данный момент не существует надежных и проверенных временем схем монетизации, основанных на семантических технологиях. В Web 2.0 одной из популярных средств монетизации является реклама. Она приносит сверхприбыли таким корпорациям как Google, Yandex и другим. В Web 3.0 реклама впринципе невозможна, так как информация предоставляет для машинной обработки и конечный пользователь её даже не увидит.

Можно рассмотреть случай, когда рекламная информация попытается встроиться в RDF как обычная. В этом случае программа Reasoner, которая делает логические выводы и предоставляет их пользователю, просто проигнорирует такую информацию, по двум причинам. Во-первых, рекламная информация не относится к предметной области. Во-вторых, если такая информация не соответсвует действительности, то она будет отсеяна как ложная.

Второй популярной схемой монетизации в Web 2.0 является продажа контента. Учитывая тот факт, что вся информация в Web 3.0 открыта, продажа контента напрямую с помощью семантических технологий невозможна.

В качестве вывода можно сказать, что объективно, Семантический Веб является полезной идеей, однако её популярность в данный момент оставляет желать лучшего. Более того, нет никаких предпосылок для быстрого увеличения его популярности. Для того, чтобы сделать эту технологию популярной и дать ей возможность динамичного развития, необходимо создавать инфраструктуру, средства монетизации и многие другие необходимые вещи.

## Постановка проблемы развития Web 3.0

Как было описано в главе 2.1, в Семантическом Веб данные представлены в виде триплетов, предоставляемых в формате RDF. Web 2.0, в свою очередь, предоставляет информацию человекочитаемыми технологиями, такими как HTML, CSS и JavaScript. Технологии Семантической и Всемирной паутин разные. Это логично, потому что цель одного стека технологий – предоставить информацию алгоритмам, а другого – отобразить её для человека.

В современном IT-бизнесе заказчикам не нужно объяснять целесообразность использования Web 2.0 приложений. Каждый адекватный архитектор знает, что правильно спроектированное и реализованное приложение может либо сэкономить заказчику миллионы долларов, либо их заработать. Стек технологий Web 2.0 давно зарекомендовал себя как выгодное вложение денег.

Иная ситуация с семантическими технологиями. Стек технологий Web 3.0 является новым и его полезность для заказчика неочевидна. Более того, коммерческая среда редко соглашается использовать непроверенные временем технологии, так как это связано с рисками для бизнеса. Разрабатывать Web-приложение только на базе Web 3.0 для заказчика нецелесообразно, в связи с тем, что в таком случае у него не будет приложения, с которым могут работать конечные пользователи. Если приложение является сервисом, то дешевле сделать его на базе Web-сервисов SOAP и REST и расписать реализованный API.

Если же использовать Семантический Веб совместно с Web 2.0 приложением, то это вполне приемлемый вариант. Однако, этот вариант означает, что если заказчику программного обеспечения нужен продукт, реализующий Семантические технологии, то при для его разработки потребуется написать два разных слоя представления: один для человека, а другой - машиночитаемый. Данный факт значительно увеличивает время и стоимость разработки. Как следствие, компании отказываются от использования Семантических технологий в своих проектах и развитие этих технологий обусловлено лишь инициативными группами университетов и Open-Source проектов.

## Пути решения проблемы развития Web 3.0

В качестве решения проблемы, описанной в предыдущей подглаве данной магистерской диссертации предлагается вариант, когда код хотя бы одного из слоёв представления (Семантического или Web 2.0) будет генерироваться автоматически.

С пользовательским интерфейсом в коммерческих приложениях это впринципе невозможно. Во-первых, потому что бизнес всегда будет диктовать новые требования и постоянно модифицировать HTML, CSS и JavaScript код. Во-вторых, сам стек технологий Web 2.0 не предусматривает какую-либо обязательную структуру кода, следовательно задача генерации разметки, скриптов и стилей в автоматическом режиме на основе семантики сущностей приложения возможна только при наличии искуственного интеллекта, что в данный момент нереализуемо.

Ситуация с семантическим предоставлением информации полностью противоположная. Так как и модель базы данных, и семантическое хранилище являются машиночитаемыми, то имеется возможность конвертировать одно в другое и наоборот. Задачу облегчают такие современные средства, как библиотека LINQ.Expressions, которая входит в базовую комплектацию .NET Framework.

Теоретически, данную идею можно реализовать, как минимум, двумя путями. Первый – это использовать его как шаблон проектирования. Иными словами, переписывать для каждого приложения отдельно. Очевидно, что это жизнеспособное, но не самое лучшее решение. Вероятно, этот подход уже используется в каких-либо проектах.

Второй способ – создание библиотеки под определенную платформу. Этот вариант реализации предпочтительнее, так как позволяет написать код один раз для всех проектов на этой платформе, которые пожелают его использовать. Именно этот путь был выбран для данной магистерской диссертации.

## Выбранный путь решения

Итого, требуется разработать библиотеку под платформу .NET Framework (обоснование выбора платформы можно найти в главе 4.1). Библиотека должна быть интегрирована с Entity Framework, ASP.NET WebAPI и dotNetRdf. Библиотека должна состоять из следующих модулей:

* реализация шаблона проектирования «Репозиторий»;
* контроллер WebAPI;
* контроллер семантического сервиса;
* модуль интерфейсов;
* менеджер конфигурации;
* модуль валидации.

Рассмотрим подробнее каждый из модулей. В коммерческих проектах для работы с базами данных используется шаблон проектирования «Репозиторий»[27]. Его суть в том, что он унифицирует доступ к сущностям. В данном случае его необходимо реализовать в библиотеке в связи с необходимостью унификации доступа к данным из контроллеров. Это позволит избежать дублирования кода. Как небольшое дополнение, реализация паттерна в библиотеке позволит опустить его реализацию в самом коммерческом приложении, что снизит затраченные на приложение ресурсы. Шаблон проектирования «Репозиторий» должен быть создан средствами обобщенных классов.

Базовые контроллеры WebAPI и семантического сервиса необходимо реализовать для того, чтобы приложение по-умолчанию поддерживало базовые операции работы с сущностями. Разработчики приложения унаследуют свои контроллеры от базового и получат всю базовую функциональность, не написав ни одной строки кода. Контроллеры не должны быть связаны с соответствующим репозиторием напрямую. Вместо этого следует использовать технологии инверсии управления.

Интерфейсы для контроллеров WebAPI и семантического сервиса спроектированы в целях возможности юнит-тестирования. Это необязательное решение, однако оно может стать ключевым для выбора использования данной библиотеки в коммерческом проекте.

Семантический модуль также должен содержать конвертер данных, который является ядром библиотеки. Необходим для конвертации сущностей в семантические данные для последующего предоставления клиентам. Конвертер данных должен быть реализован при помощи рефлексии и LINQ.Expressions.

Стоит отдельно разработать модуль конфигурации. Так как библиотека планируется к использованию в коммерческих проектах, то необходимо уделить расширяемости большое внимание. Модуль конфигурации должен уметь работать не только со стандартной реализацией конфигурации и применять соответствующие настройки как только конфигурация была изменена, но и дать разработчику клиентского приложения реализовать собственную конфигурацию, реализуя соответствующий интерфейс. Это также может стать решающим фактором в пользу использования данной библиотеки в коммерческом проекте.

В целях упрощения поддержки приложения необходимо реализовать юнит-тесты для репозитория, контроллера и семантического сервиса. Разработка должна проводиться по методологии TDD (Test Driven Development). Методология TDD подразумевает разработку в три этапа. Первый этап – написание тестов. На этом этапе тесты не должны проходить, так как кода еще нет. Второй этап – написание кода. На этом этапе программист должен любыми способами написать код так, чтобы прошли все тесты. И на завершающем третьем этапе выполняется рефакторинг, код оптимизируется и приводится к читаемому виду.

Для демонстрации работы фреймворка в рамках магистерской диссертации должно быть разработано небольшое приложение из 6 сущностей. Оно демонстрирует основные методы работы с сущностями. В разработке данного приложения планируется использовать Web 2.0 фреймворк Angular, а также, адаптировать это приложение для наглядного тестирования семантической части.

# Разработка библиотеки

## Использованные программные и языковые средства

При разработке в процессе выполнения магистерской диссертации программного продукта была использована платформа .NET Framework версии 4.6.2 (CLR 4.0). .NET Framework – это программная платформа, которая была выпущена Microsoft в 2002 году. Принцип данной платформы заключается в том, что программа, написанная на любом поддерживаемом языке программирования сохраняется в сборку (assembly), содержащаую специальный, независимый от используемого аппаратного обеспечения IL-код (Intermediate Language). Сборка уже выполняется виртуальной машиной CLR (Common Language Runtime). Это позволяет программистам разрабатывать программы на любом языке программирования, а также не заботиться об особенностях аппаратной части.

Платформа .NET была выбрана в связи с рядом значимых преимуществ перед конкурентами, в частности Java:

* «синтаксический сахар» C#, который значительно увеличивает скорость разработки приложения;
* удобство компиляции и внедрения компонентов в операционную систему Windows;
* улучшенная скорость выполнения кода в связи с тем, что CLR оптимизирует IL код под различные архитектуры процессоров;
* самый удобный на рынке фреймворк для разработки Web-приложений ASP.NET (в частности использовался ASP.NET WebAPI 2);
* удобство отладки, предоставленное IDE Microsoft Visual Studio 2015;
* набор технологий LINQ (LINQ to SQL, LINQ.Expressions), которые предоставляют не только богатые возможности работы с коллекциями C#, но и возможности работы с внешними источниками данных, не углубляясь в детали их реализации;
* наличие библиотеки dotNetRdf с открытым исходным кодом.

В качестве отрицательной стороны данной платформы можно отметить отсутствие кроссплатформенности, однако этот минус на сегодняшний день является незначительным в связи, во-первых, со стандартизацией технологий Microsoft. Это значит, что приложение можно запустить и на остальных операционных системах, таких как GNU/Linux, однако это потребует значительно больших затрат ресурсов и времени. Во-вторых, приложение в перспективе планируется перенести на платформу .NET Core, которая является кроссплатформенной и использует язык C# и инфраструктуру NuGET. В данный момент технология является нестабильной и разрабатывать библиотеку на ней сегодня нецелесообразно в связи с большими рисками.

Приложение было написано на языке C#. Это объектно-ориентированный язык, реализующий принципы инкапсуляции, наследования и полиморфизма. Он является наиболее используемым языком в .NET Framework. C# является одним из наиболее используемых языков программирования для разработки прикладного программного обеспечения в промышленной среде.

Для разработки программного обеспечения потребовалось использовать следующие программные продукты:

* IDE Microsoft Visual Studio 2015 Enterprise[28];
* JetBrains ReSharper 10 Ultimate[29];
* MS SQL Management Studio 2014;
* библиотека dotNetRdf для работы с семантическими данными[30];
* ASP.NET WebAPI 2;
* Entity Framework 6.0.

Рассмотрим подробнее данные программные средства.

В качестве интегрированной среды разработки была выбрана Microsoft Visual Studio 2015 Ultimate. Это наилучший из вариантов IDE для разработки на C# для Windows. В качестве альтернатив можно было использовать Visual Studio Code или Mono, однако они значительно уступают в функциональности, поэтому даже не рассматривались на роль IDE для разработки.

Линейка продуктов JetBrains ReSharper Ultimate использовалась для ускорения процесса разработки. Это набор надстроек над Visual Studio, который автоматизирует часто повторяющиеся действия, выполняет статический анализ кода, следит за соблюдением стилистики кода, содержит средства для декомпиляции, запуска тестов и анализа покрытия тестами. В общем случае, данное средство ускоряет процесс разработки в 2-3 раза.

MS SQL Management Studio использовалась для работы с базой данных. В частности, для отладки процесса создания базы данных Entity Framework. Сам Entity Framework используется для управления базой данных. Он полностью создает базу данных из классов C#, генерируя к ней запросы «на лету».

Для работы с RDF-данными была использована библиотека dotNetRdf. Библиотека была внедрена в проект как NuGet-пакет.

## Описание архитектуры

При разработке любого фреймворка самой сложной составляющей работы является проектирование и создание архитектуры. В данной работе архитектура разрабатывалась согласно принципам методологии разработки Extreme Programming. Это означает, что в процессе разработки архитектура (метафора системы в терминах XP) должна изменяться в соответствие с изменениями требований. На момент написания данной дипломной работы приложение имеет архитектуру, основанную на инъекции зависимостей. Основная идея в том, чтобы минимизировать количество зависимостей модулей друг от друга путем выведения интерфейсов. Затем, в сопоставить интерфейсы реализующими их классами только на этапе выполнения.

Вышеизложенный подход был выбран в связи с тем, что он имеет ряд положительных качеств. Во-первых, любой модуль фреймворка при необходимости может быть заменен. Без этого невозможно полноценное юнит-тестирование, а также эта возможность будет полезна при обнаружении ошибок. При исправлении одного из модулей, достаточно будет заменить в работающей системе только его.

Во-вторых, в любой момент можно отключить один или несколько модулей от системы без потери ее работоспособности. Это может быть критично в случае разработки систем, требовательным к производительности. Из данных качеств архитектуры следует также и то, что при публикации фреймворка можно выложить его модульно и не принуждать пользователя использовать те библиотеки, которые ему не нужны. Более того, он может сам написать свою реализацию одного или нескольких из модулей, если его не устраивает реализация поставочного.

Рассмотрим подробнее диаграмму модулей, которую можно увидеть на рисунке 7. Диаграмма модулей сгенерирована программным продуктом ReSharper автоматически. Она отображает все модули, которые есть в решении и связи между ними. Решение состоит из 14 проектов (модулей), которые разделены на две части: сама библиотека и демонстрационное приложение. На диаграмме Library и Application соответственно. Цифры на диаграмме обозначают количество фактических обращений к классам модуля.

Взглянув на диаграмму, можно увидеть, что разработанный фреймворк состоит из пяти независимых друг от друга библиотек:

* Selp.Controller – уровень представления Web 2.0;
* Selp.Semantic – уровень представления Web 3.0;
* Selp.Configuration – модуль конфигурации приложения;
* Selp.Repository – модуль реализации шаблона проектирования Репозиторий;
* Selp.Validator – модуль валидации.

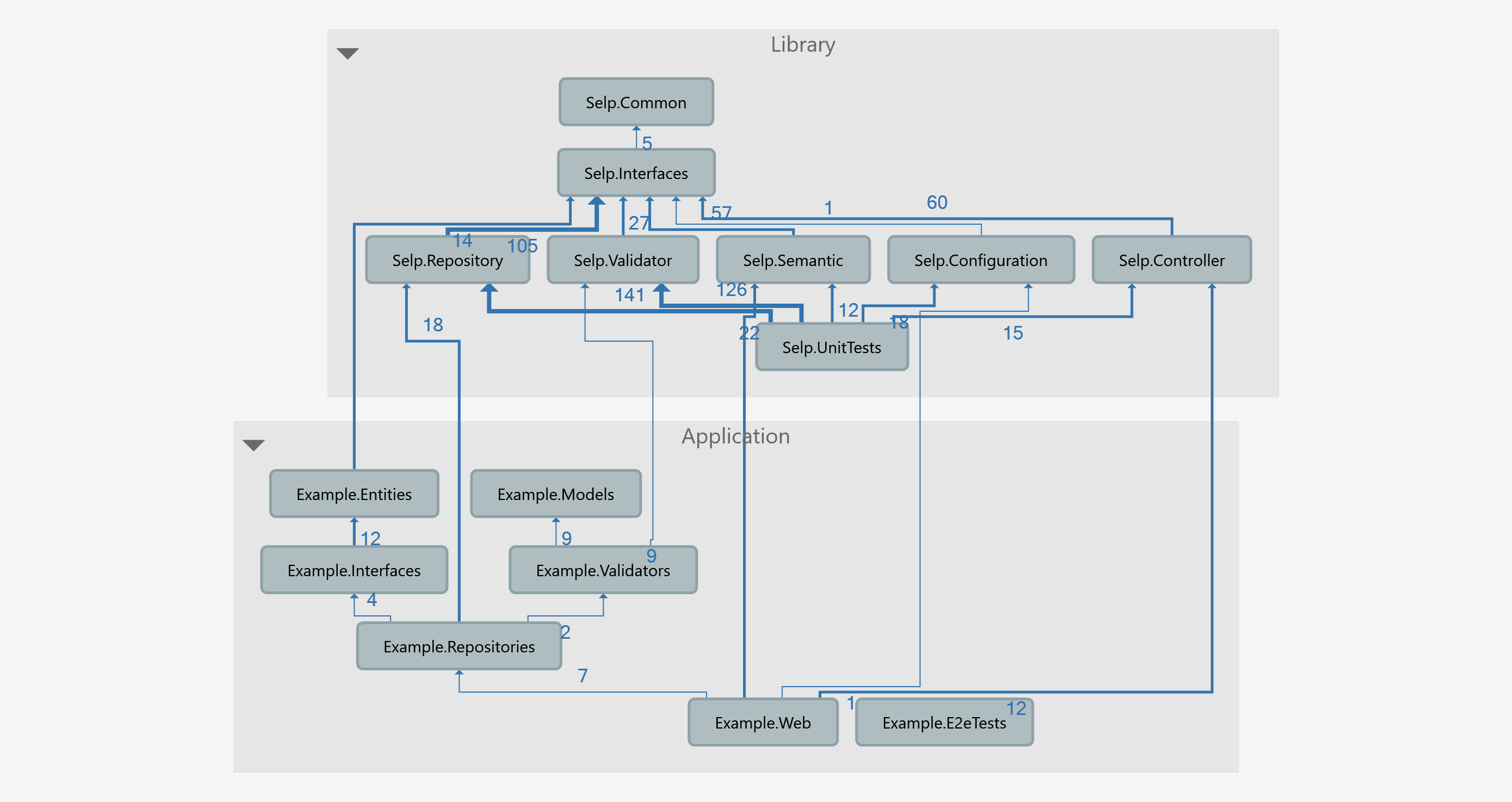


Рисунок 6 – Архитектура фреймворка Selp.

Каждая из этих библиотек зависима от модуля интерфейсов Selp.Interfaces. Это логично, так как все библиотеки должны знать об API других. Модуль в свою очередь содержит зависимость от Selp.Common. Данный модуль семантически модулем не является. Он содержит всего лишь набор сущностей, исключений, перечислений, аттрибутов, классов-помощников и прочих служебных вещей, которые используются по всему проекту.

Последним классом, который можно увидеть на диаграмме является Selp.UnitTests. Как можно понять из названия этот класс отвечает за юнит-тестирование. Также очевидно, что он должен зависеть ото всех модулей проекта, так как он тестирует их реализацию.

Данная архитектура полностью соответствует принципам SOLID и GRASP. Она жизнеспособна, относительно проста, расширяема и легко поддерживаема. Библиотека открыта для расширения, закрыта для случайной модификации, каждый ее модуль имеет одну единственную ответственность и при необходимости может быть заменен.

## Описание модуля Репозиторий

Предназначение модуля Репозиторий – управление данными в системе. Модуль основан на одноименном шаблоне проектирования. Шаблон проектирования Репозиторий ставит своей целью централизацию и унификацию доступа к данным. Данная цель достигается при помощи реализации CRUD-операций для каждой сущности. Иными словами, репозитории должны создавать, редактировать, удалять и получать сущности, которыми они управляют, из источника данных.

В решении, выполненном в рамках данной магистерской диссертации источником данных репозиториев является EntityFramework. На текущий момент EntityFramework умеет работать только с реляционными базами данных. Однако, поддерживаются базы данных всех типов, которые поддерживаются ADO.NET. В перспективе, при обновлении на следующую версию EntityFramework (которая в данным момент находится в разработке) станет возможно работать с любыми источниками данных в целом: тестовыми файлами, XML-документами, хранилищами в оперативной памяти, нереляционными базами данных и множеством других.

Основная идея EntityFramework в том, что он избавляет разработчика от самостоятельной разработки базы данных. Фреймворк создает и управляет базой данных с помощью генерации SQL-запросов. Генерация происходит на основе данных обычных классов-сущностей. От программиста требуется только создание этих классов и объявление их в объекте-наследнике специального класса DbContext.

Наследники DbContext используются для доступа к базе данных. Это обязательное требование EntityFramework. Класс состоит из полей обобщенных коллекций DbSet<>. Каждое из этих полей является таблицей в базе данных. При создании базы данных именно этот класс отвечает за связь классов и таблиц базы данных. При необходимости есть возможность, переопределив некоторые методы, изменить стандартное поведение EntityFramework и настроить самостоятельно такую функциональность как внешние и первичные ключи, первоначальную инициализацию данных, транзакции, хранимые процедуры, функции, названия самой базы данных, таблиц, полей, каскадное удаление, изменение. Иными словами, в этом классе мы имеем полный контроль над базой данных из C# кода.

Отдельно стоит отметить функциональность миграций в EntityFramework. Используя DbContext есть возможность сгенерировать скрипт для изменения базы данных без потери данных при изменении модели базы данных в коде.

В данной работе реализация шаблона проектирования Репозиторий выполнена в виде обобщенного абстрактного класса SelpController<TEntity, TKey>. Упрощенную диаграмму классов можно увидеть на рисунке 7. В частности, на нем изображен публичный API (интерфейс ISelpRepository< TEntity, TKey>), а также сам абстрактный класс. Несложно увидеть, что API представляет собой обобщенный набор CRUD операций.

В данном решении шаблон проектирования Репозиторий был несколько расширен. В него было добавлено несколько дополнительных задач, которые так или иначе влияют на поведение Репозитория и не могут быть использованы самостоятельно.

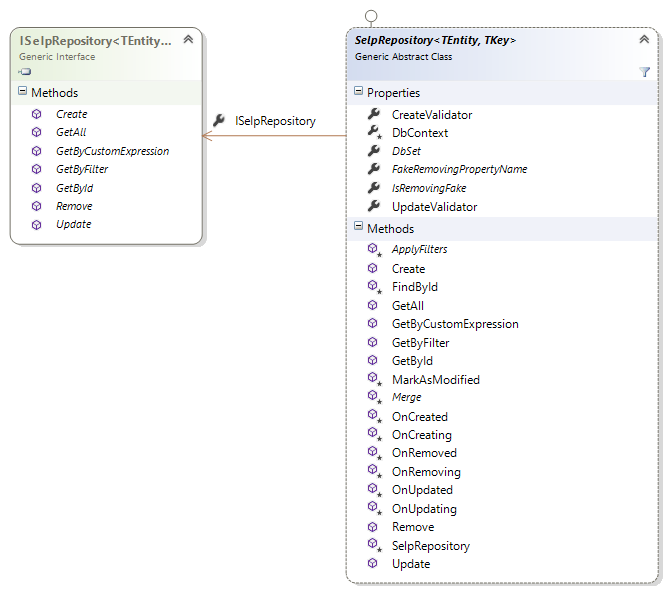


Рисунок 7 – Фрагмент диаграммы классов библиотеки Selp.Repository.

Первой такой задачей является фильтрация и паджинация данных. В крупных проектах стандатный CRUD-метода GetAll несет в себе потенциальные проблемы с производительностью, так как возвращает все записи вне зависимости от того, сколько их в источнике данных. Например, на рабочем сервере может храниться информация о десяти миллионах пользователей. Следовательно метод по контракту обязан вернуть их все. Итого получается, что на один клиентский запрос необходимо выкачивать данные обо всех пользователях, несмотря на то, что они потенциально могут быть не нужны. Чтобы избежать этой проблемы в Репозитории был реализованы дополнительные методы: GetByCustomExpression и GetByFilter. Первый метод является в большей мере служебным и позволяет выполнить запрос к базе по своему условию, которое может быть любым. Предполагается, что данный метод будет использоваться в случае необходимости создания какой-либо нестандартной логики. Метод GetByFilter позволяет отсортировать и выбрать именно те данные, которые нужны пользователю.

Рассмотрим этот метод детально. В качестве единственного параметра он принимает фильтр, который по-умолчанию поддерживает сортировку по возрастанию и убыванию по одному из полей. Дополнительно, для уменьшения количества данных реализована паджинация. Фильтр содержит два поля: номер страницы (Page) и количество элементов на странице (PageSize). Исходя из этих данных EntityFramework может построить запрос таким образом, чтобы из базы приходило сразу ограниченное количество записей. Для обеспечения расширяемости метода создан защищенный абстрактный метод ApplyFilters, который будет фильтровать записи по нестандартному условию. Параметры для такого условия можно передать, унаследовав класс BaseFilter и разместив аргументы внутри нового класса.

Вторая задача, которая была выполнена в данной реализации репозитория – создание механизма ложного удаления. Механизм предусматривает замену фактического удаления сущности на выставление ей флага IsDeleted = true. Механизм часто используется в коммерческих проектах для сохранения истории изменений и защиты от непреднамеренного удаления. Для включения механизма необходимо переопределить свойство IsRemovingFake и установить имя свойства, которое будет ответственно за распознавание ложно-удаленных сущностей.

Последняя задача, которая служит для увеличения расширяемости фрейворка – это инициация событий на изменение данных в источнике. При необходимости реализация репозитория может узнать, что сущность была создана, изменена или удалена. Данная возможность является заменой триггерам в базе данных и может быть использована в целях логирования, отладки, преобразования данных, а также в целях применения нестандартной логики.

## Описание модуля Валидатор

Изначально, модуль Валидатор спроектирован не был. Однако, при разработке библиотеки и ее практическом использовании появилась необходимость валидировать одни и те же данные в разных модулях. При создании кода валидаций возникло несколько неприятных моментов. Во-первых, валидации привели к увеличению количества кода. Данный факт был ожидаем, так как большую часть кода в проектах обычно составляет валидация. Во-вторых, увеличило связность системы, которой хотелось избежать с целью соответствия архитектуры задуманной.

Результатом анализа данной ситуации стало решение вынести валидационный код в отдельный модуль. Внешний интерфейс модуля (его можно увидеть на рисунке 8) представляет собой всего один метод – Validate, однако внутри он делает большой объем работы. Результаты работы использующие модули получают средствами свойств IsValid и Errors, которые сообщают о том, прошла ли валидация и какие именно ошибки произошли соответственно. Интерфейс модуля встроен в Selp.Repository и его реализации могут быть легко добавлены в него.

Валидатор может иметь вложенные валидаторы. Такая функциональность предоставляется по-умолчанию. При валидации сущностей валидатор выполняет проверку, которая предусмотрена в нем, а также запускает метод Validate у всех вложенных валидаторов. Вложенные валидаторы, в свою очередь также могут иметь вложенные валидаторы и при валидации запускают их. Если хотя бы в одном валидаторе произошла ошибка, то валидатор становится невалидным. Вне зависимости от результатов всегда выполняются все вложенные валидаторы. По результатам валидации можно получить весь список ошибок с конкретным указанием имен полей, в которых произошли ошибки.

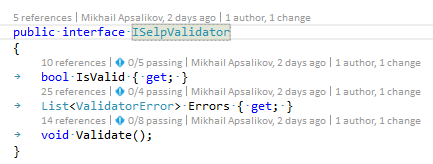


Рисунок 8 – Интерфейс валидационного модуля Selp.Validator.

## Описание модуля Контроллер

Рассмотрим детально Web 2.0 контроллер. Он представляет собой обычный REST-сервис с методами CRUD:

* Get([FromUri] BaseFilter query) – запрос сущностей по фильтру;
* Get(TKey id) – запрос на конкретную сущность по ID;
* Post([FromBody] TModel value) – создание сущности;
* Put(TKey id, [FromBody] TModel value) – изменение сущности;
* Delete(TKey id) – удаление сущности.

Контроллер представляет собой класс, унаследованный от ApiController, который входит в состав ASP.NET WebAPI. Это означает, что в приложении каждый из его методов будет доступен через серверный роутинг по соответствующему HTTP-глаголу.

Основная задача контроллера в данной системе – предоставление данных клиенту в виде REST-сервиса. Для успешной и безопасной реализации REST-сервиса необходимо изолировать данные, которыми обменивается клиент и сервер и данные из базы данных друг от друга. Для этого разделяются понятия сущности и модели. Разделение необходимо в связи с тем, что отдавать сущность в пользовательский код небезопасно, она автоматически связана кодом библиотеки EntityFramework с базой данных и при случайной ее модификации можно повредить целостность данных в самом источнике данных.

Попробуем дать определения сущности и модели. Сущность – это объект, семантически представляющий собой предмет или понятие предметной области, которым оперирует бизнес-логика приложения. Модель – это объект, семантически представляющий собой предмет или понятие предметной области, адаптированный под уровень представления. В контексте EntityFramework сущность также логически связана с базой данных, а также может эти данные фактически не содержать, а выдавать их по запросу. Такая техника называется Lazy loading (отложенная загрузка). Именно из-за отложенной загрузки нерационально отдавать клиенту данные в виде сущности. В контексте ASP.NET WebAPI модель представляет собой обычный класс с набором свойств, который гарантированно содержит в себе именно те данные, которыми он обменивается с клиентом. При конвертации сущность => модель данные из сущности загружаются вне зависимости от их фактического размещения и помещаются в модель. Это позволяет без проблем отдавать модель клиенту при дальнейшей обработке.

В продолжение темы моделей также следует упомянуть, что в разработанном фреймворке используется два типа моделей: полная (TModel) и сокращенная (TShortModel). Предполагается, что полная модель содержит в себе все поля, которые когда-либо могут понадобиться клиенту (обычно набор полей совпадает с сущностью, но могут быть исключения). Такая модель может содержать в себе большие по размеру поля такие как файл или xml. В реализации фреймворка полная модель используется только для получения конкретной сущности по ID.

Сокращенная модель содержит в себе только те поля, которые однозначно характеризуют сущность. Для выбора списка сущностей используется именно эта модель, так как данных много. Если клиентскому приложению понадобится полная модель, он в любой момент сможет запросить ее отдельно по ID. Эта модель также возвращается в методах Post и Put. Теоретически, это может понадобиться для того, чтобы убедиться что запись создалась успешно и какие точно поля записались в источник данных. Возможен также вариант, что некоторые поля будут изменены в процессе создания/изменения сущности, однако это нерекомендуемая к использованию возможность. В этом случае они также будут отражены в возвращаемой сокращенной модели.

Помимо вышеизложенных задач, контроллер также ответственнен за обработку ошибок в серверном коде и их преобразование в ответы клиенту с соответствующими HTTP-коды ошибок. В данной работе контроллер распознает ошибки фреймворка и возвращает коды 404, 405 и 500 когда это требуется. Поведение можно изменить с помощью переопределения виртуального метода HandleException.

## Описание модуля Семантический контроллер

Аналогично, Web 2.0 контроллеру, Семантический контроллер представляет собой REST-сервис, унаследованный от класса ASP.NET WebAPI ApiController. Однако, семантический сервис реализует только два метода CRUD вместо пяти:

* Get() – запрос всех сущностей;
* Get(TKey id) – запрос на конкретную сущность по ID.

Рассмотрим эти методы детально. Метод Get() возращает RDF-данные обо всех сущностях, которые есть в источнике данных. Иными словами используется метод репозитория GetAll.

Аналогично обычному Контроллеру есть четкое разделение сущностей и моделей, а модели с свою очередь разделаются на полные и сокращенные. Полные используются при запросе конкретной сущности, а сокращенные при запросе набора сущностей.

Основным различием Контроллера от Семантического контроллера является способ предоставления данных пользователю. Обычный контроллер ASP.NET WebAPI поддерживает сериализацию данных в форматы XML или JSON в зависимости от настроек сервера и пожеланий клиента.

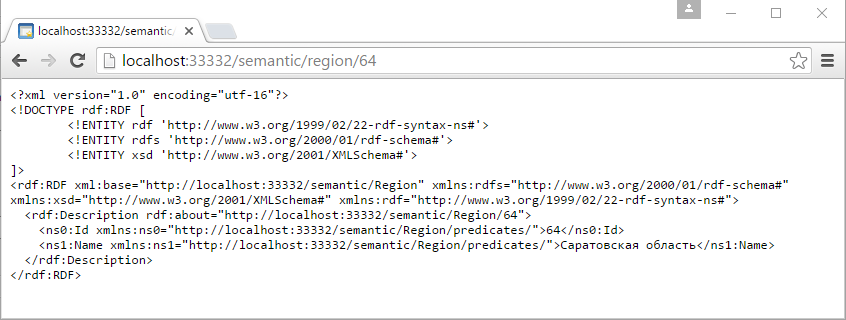
Семантический сервис обязан предоставлять данные в формате RDF. Несмотря на обилие функциональности ASP.NET не умеет работать с семантическими форматами. Это означает, что необходимо создать свое решение этой задачи. Такое решение было создано и представляет собой два связанных между собой алгоритма. Во-первых, перед тем как использовать данные нужно сгенерировать граф. Этим занимается специальный внутренний статический класс SemanticCore. Во-вторых, данные полученного графа необходимо сериализовать. Эта операция делегируется библиотеке dotNetRdf, так как она умеет работать с множеством различных сериализаций RDF и предоставляет соотвествующий интерфейс, который в свою очередь можно и рекомендуется использовать в реализации инверсии управления.

Класс SemanticCore представляет собой единственный публичный метод GenerateGraph, который принимает в качестве параметра модель или коллекцию моделей. Результатом его работы является объект типа IGraph, который содержит в себе RDF-граф.

Давайте рассмотрим алгоритм работы преобразования модель-граф:

* создание пустого графа;
* определение базового URI графа;
* инициализация графа;
* создание субъекта модели;
* запрос информации о типе модели (рефлексия);
* для каждого свойства модели вытащить информацию о свойстве и записать ее в предикат;
* для каждого свойства модели вытащить информацию содержащуюся в объекте и записать ее в объект триплета;
* добавить триплеты в граф;
* вернуть граф.

После выполнения алгоритма преобразования граф сериализуется с использованием объекта, реализующего интерфейс IRdfWriter, который входит в поставку dotNetRdf. Это означает, что мы можем как использовать стандартный сериализатор этой библиотеки, так и написать свой. Пример работы семантического сервиса можно увидеть на рисунке 9.

Рисунок 9 – Пример ответа за запрос к семантическому ресурсу.

## Инверсия зависимостей в библиотеке

Вернемся к рассмотрению архитектуры приложения. Как было упомянуто ранее в главе 4.2 архитектура приложения построена на использовании инверсии зависимостей. За саму реализацию инверсии зависимостей в клиентском коде будут отвечать разработчики конечных приложений. В библиотеке всего лишь сделана возможность ее реализации. Для реализации инверсии зависимостей можно использовать два шаблона проектирования: Service Locator и IoC-контейнер (Inversion of Control container). Они могут быть реализованы как вручную, так и с использованием библиотек, таких как UnityContainer от Microsoft или Ninject. Рассмотрим эти шаблоны проектирования подробнее.

Оба шаблона проектирования представляют собой объект, который настраивает связи «интерфейс-класс» для определенной точки входа в приложение. Класс имеет методы для настройки связей, с помощью которых происходит настройка, а также метод Resolve, который создает объект реализующий соответствующий интерфейс согласно настройкам и возвращает его.

Отличие этих шаблонов в том, что ServiceLocator представляет собой глобальный статический объект и доступ к нему есть во всем коде приложения. IoC-контейнер в свою очередь виден только в точке входа в приложение и автоматически подставляет модули в качестве параметров. Каждый из подходов имеет свои положительные и отрицательные стороны, однако решение в любом случае останется за разработчиками конечного приложения. Возможен даже такой вариант, что инверсия зависимостей использоваться не будет. В таком случае, разработчики берут на себя все риски связанные с расширением своей системы.

В демонстрационном приложении, речь о котором пойдет в главе 5 использует шаблон проектирования IoC-контейнер, предоставляемый с помощью библиотеки UnityContainer.

## Производительность

Одним из сложнейших вопросов в жизни любого разработчика программного обеспечения на любой платформе и языке программирования является проблема производительности разрабатываемых приложений. Фреймворк для генерации семантических данных, разрабатываемый в данной магистерской работе не является исключением.

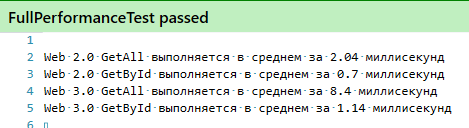
После реализации программного продукта были написаны четыре e2e-теста на производительность, которые пытаются получить одни и те же данные через разные контроллеры: обычный и семантический. Их результаты вы можете увидеть на рисунке 10. Тестирование производилось на простых данных (~90 записей, содержащих 2 поля). Однако этого достаточно, чтобы понять, что производительность фреймворка достаточно хорошая. Проблемы с производительностью обычно начинаются, если запрос начинает выполняться более 500-1000 миллисекунд.

Рисунок 10 – Результаты тестов производительности.

Однако стоит обратить внимание, что производительность семантической части ниже, чем производительность обычного REST-сервиса. По ходе тестирования производительности эта разница приблизительно 2-4 раза. Это связано с тем, что семантическому сервису дополнительно приходится конвертировать данные в граф и сериализовывать их. Более того, информация в формате RDF/XML, который использовался при тестировании имеет больший объем, чем информация в формате JSON.

Помимо данных, построения графов и сериализации производительность также теряется из-за использования рефлексии. В отличие от вышеизложенных проблем, проблему с рефлексией можно решить и это будет сделано в следующей версии фреймворка. Основная идея оптимизации производительности в том, чтобы собрать информацию о классе один раз, закешировать ее и автоматически сгенерировать высокопроизводительную функцию, которая будет создавать RDF-граф без использования рефлексии. Иными словами, при первом запросе к сущности запрос будет идти дольше, однако все последующие запросы пройдут намного быстрее. Данное решение, предположительно сможет уменьшить разницу между Web 2.0 и Web 3.0 контроллерами до 1.2-1.5 раз.

Другим путем увеличения производительности в конечном приложении может быть замена диалекта RDF с RDF/XML на более легковесный, например Notation3. В этом случае, на больших объемах данных возможен вариант, что семантический сервис станет работать быстрее обычного, так как обычный контроллер тоже сериализует данные, только в формат JSON, который требует меньше ресурсов, чем RDF/XML, но больше ресурсов, чем Notation3.

Следующим путем для увеличения производительности безусловно является поддержка асинхронности. В данный момент, в приложении она не поддерживается, однако ее планируется реализовать в одной из следующих версий. Технология асинхронности позволяет экономить потоки. Напрямую, на прирост производительности она не повлияет, однако при большом количестве запросов на сервер, когда закончатся все выделенные сервером на процесс потоки синхронное приложение вынуждено будет ждать, в то время как асинхронное приложение продолжит стабильно работать.

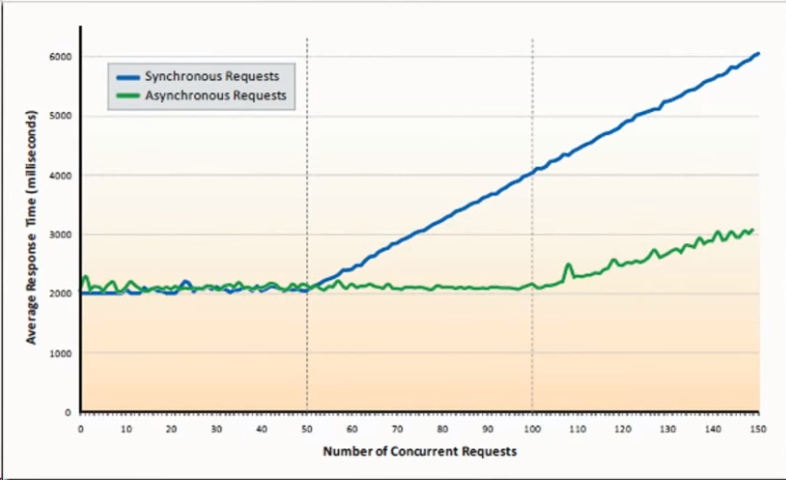


Рисунок 11 – Сравнение производительности синхронного и асинхронного подходов.

Сравнение времени отклика на запрос синхронного и асинхронного подходов можно увидеть на рисунке 11. Синей линией обозначен синхронный подход, зеленой – асинхронный. По достижению максимального количества потоков (50) синхронное приложение начинает значительно терять в производительности. В свою очередь асинхронное приложение в производительности по достижению максимального количества потоков не теряет.

## Тестирование

Согласно практикам методологии разработки Extreme Programming тестирование должно проводиться по технике TDD (Test-Driven Development). Общий процесс разработки выглядит так:

* добавление теста;
* запуск тестов (тесты должны показать отрицательный результат);
* написание кода;
* запуск тестов (тесты должны показать положительный результат);
* рефакторинг.

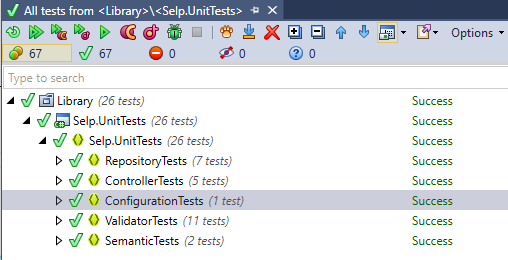


Рисунок 12 – Выполненные тесты фреймворка Selp.

Подход к разработке TDD увеличивает время на разработку на 35%, однако значительно уменьшает время на поддержку (до 90%, в зависимости от качества тестов). Однако, учитывая, что на поддержку кода обычно тратится больше времени, чем на его разработку, в конечном итоге – это оптимизация времени на разработку/поддержку программного продукта.

Главной метрикой качества автоматизированного тестирования является величина покрытия кода тестами в процентах. В данной работе величина покрытия тестами фреймворка составляет: 87%. Идеальная величина покрытия тестами, очевидно, равна 100%. Однако стоит понимать что, во-первых, это недостижимо на реальных проектах. Во-вторых, каждые последующие 10% покрытия тестами требуют все большего и большего количества временных затрат (они растут в геометической прогрессии). На реальных проектах, хорошей практикой считается покрывать код тестами на 80%. Покрытие кода тестами на 90% можно считать идеальным.

Как уже было сказано, величина покрытия тестами в фреймворке составляет 87%. На рисунке 12 можно увидеть сами выполненные тесты, а на рисунке 13 - как именно они распределены. Наиболее качественно покрыты тестами модули Валидатор и Конфигурация. Конфигурацию тестировать просто, она является небольшим модулем, поэтому у нее высокий процент. Валидатор полностью состоит из пользовательской логики, поэтому его тоже вполне реально покрыть тестами на 100%. Репозиторий также покрыт качественно, потому что логически он является самым важным модулем в системе и именно его нужно тестировать в первую очередь.

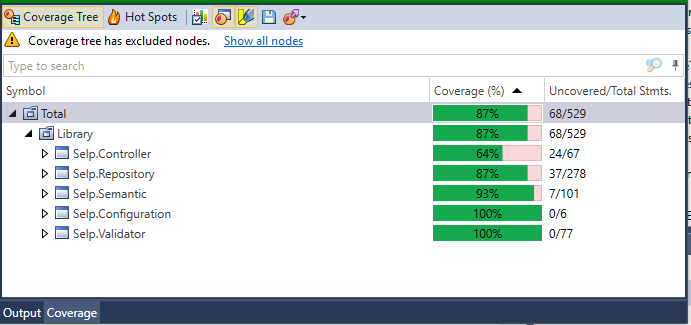


Рисунок 13 – Анализ величины покрытия тестами фреймворка Selp.

Слабым местом в тестировании фреймворка является инфраструктура ASP.NET WebAPI. Модули Семантический Контроллер и Контроллер зависят от нее. Тестирование инфраструктуры ресурсозатратно и нецелесообразно, так как она уже протестирована специалистами Microsoft. Семантический контроллер потерял 7% от уровня покрытия тестами, однако это приемлемо. Web 2.0 в свою очередь потерял 36%, так как он активно использует инфраструктуру ASP.NET. Именно на такие варианты развития событий закладывается 20% потерь покрытия тестами.

Суммируя, имеем 87% покрытие тестами фреймворка, что является отличным результатом.

# Разработка демонстрационного приложения

Для того, чтобы продемонстрировать возможности фреймворка было разработано простое приложение из четырех связанных между собой сущностей. Также тестовое приложение демонстрирует реализацию вложения сущностей друг в друга. В данной главе рассматриваются особенности разработанного приложения и способы его взаимодействия с основным фреймворком.

## Использованные программные и языковые средства

Для разработки демонстрационного приложения был использован язык C# и платформа .NET. Это логично, так как фреймворк зависит именно от них. Детальное описание данного стека технологий можно найти в главе 4.1. Рассмотрим программные продукты, которые были использованы для разработки самого приложения:

* IDE Microsoft Visual Studio 2015 Enterprise[28];
* JetBrains ReSharper 10 Ultimate[29];
* MS SQL Management Studio 2014;
* библиотека dotNetRdf для работы с семантическими данными[30];
* ASP.NET WebAPI 2;
* Entity Framework 6.0;
* Разработанный фреймворк Selp;
* Unity;
* Angular-material;
* Materialize.css;
* Linq.js;
* jQuery.

Часть продуктов уже описана в части 4.1 данной магистерской работы. Рассмотрим остальные продукты. Библиотека Unity используется для реализации шаблона проектирования IoC-контейнер. Библиотека разработана компанией Microsoft и является одной из самых популярных в данных целях библиотекой. Остальные дополнительные библиотеки, не рассмотренные в главе 4.1 являются клиентскими.

Основой клиентской части приложения является фреймворк Angular-material. Он основан на известном в мире JavaScript-разработки фреймворке Angular.js и расширяет его функциональность. Фреймворк позволяет значительно упростить разработку приложения, упорядочить клиентский код и повысить его поддерживаемость. В дополнение к Angular-material клиентский код использует jQuery – библиотеку для поиска элементов по HTML и Linq.js – библиотеку для работы с коллекциями на стороне клиента.

Materialize.css – это CSS-библиотека, которая является набором стилей для material-дизайна, который используется в проекте.

## Описание архитектуры

Архитектура приложения представлена в виде ASP.NET WebAPI Web-приложения, с вынесенными из нее модулями интерфейсов репозиториев, валидаторов, моделей и сущностей. Архитектуру приложения можно увидеть в главе 4.2 на рисунке 6. За приложение отвечает блок «Application» на этой диаграмме модулей.

Модуль репозиториев содержит в себе классы-репозитории соответствующие каждой сущности. Каждый из репозиториев наследуется от SelpRepository. Эти классы хранят в себе фактическую логику работы с соответствующими сущностями. Помимо репозиториев в модуле репозиториев также хранится наследник класса DbContext - ExampleDbContext и TestDataIntializer. ExampleDbContext содержит список сущностей, соответствующих таблицам в базе данных. TestDataIntializer инициализирует тестовые данные в таблице.

Модуль интерфейсов содержит в себе интерфейсы для всех контроллеров, семантических контроллеров, репозиториев и прочих классов для удобства их замены через инверсию зависимостей.

Модуль валидаторов содержит в себе валидаторы для всех сущностей системы. Модули моделей и сущностей хранят наборы моделей и сущностей соответственно.

Само Web-приложение состоит из двух логических частей: Front-End (клиентская часть) и Back-End (серверная часть). Описание Front-End части можно увидеть в главе 5.6 данной магистерской диссертации. Серверная часть содержит контроллеры и семантические контроллеры, которые отвечают за ответ на запросы к Web 2.0 и Web 3.0 частей приложения соответственно. Однако серверная часть содержит в себе не только контроллеры. Здесь можно увидеть также конфигурационный код приложения, а именно конфигурацию инверсии зависимостей, конфигурацию серверного роутинга, инициализацию семантической части приложения и многое другое.

## Описание логики

Демонстрационное приложение представляет собой простейший пример системы для создания страховых полисов в страховой компании. Приложение поддерживает базовую авторизацию и аутентификацию, в том числе клиентскую. Как было сказано ранее, приложение построено на четырех сущностях. Основная сущность – Страховой Полис(Policy). Она связана с сущностями Пользователь (User) и Регион (Region) связями один ко многим, а также связью многие ко многим с сущностью Контрагент (Party). Достаточно взглянуть на даталогическую схему базы данных, чтобы понять как работают связи между сущностями. Схему можно увидеть на рисунке 14.

Несмотря на то, что главная демонстрационная сущность – Страховой Полис содержит связь многие ко многим с сущностью Контрагент, в клиентской части эта связь не реализована в связи с тем, чтобы снизить порог входа в понимание фреймворка. Данную связь разработчики клиентских приложений могут реализовать самостоятельно.

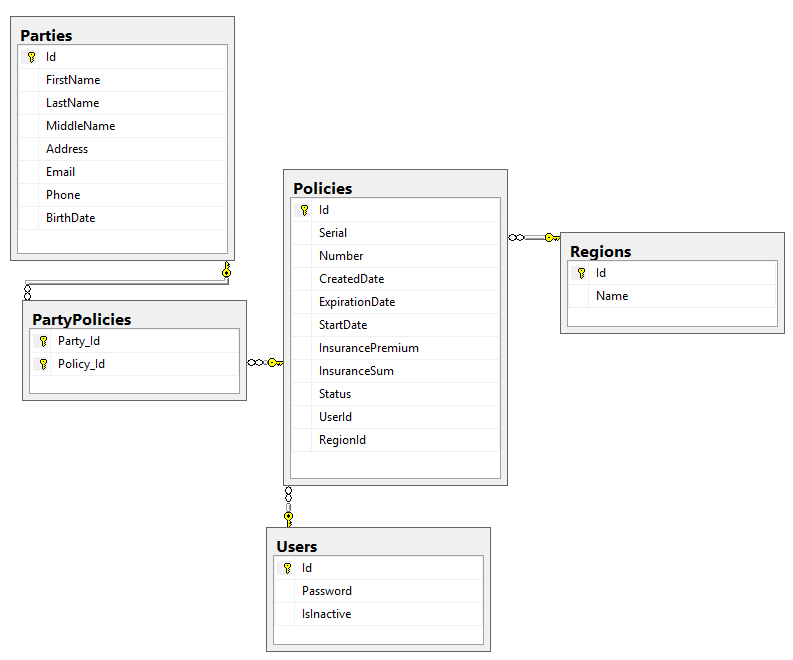


Рисунок 14 – Даталогическая схема базы данных демонстрационного приложения.

Здесь и далее (главы 5.3–5.5) будет рассматриваться процесс создания демострационной сущности и ее представление в демонстрационном приложении по шагам. При первой загрузке сайта мы видим главную страницу, оформленную по шаблону Call-To-Action (призыв к действию). Функционально, это обычная страница авторизации и регистрации. На этой странице ничего интересного с точки зрения семантического Web нет. Скриншот страницы находится на рисунке 15 данной магистерской диссертации.

Перейдем непосредственно к демонстрации создания сущности. На рисунке 16 можно увидеть страницу создания страхового полиса. На странице можно увидеть множество полей, такие как серия страхового полиса, номер страхового полиса, дата начала его действия, дата окончания действия, страховая сумма и страховая премия, статус, дата создания и многие другие.

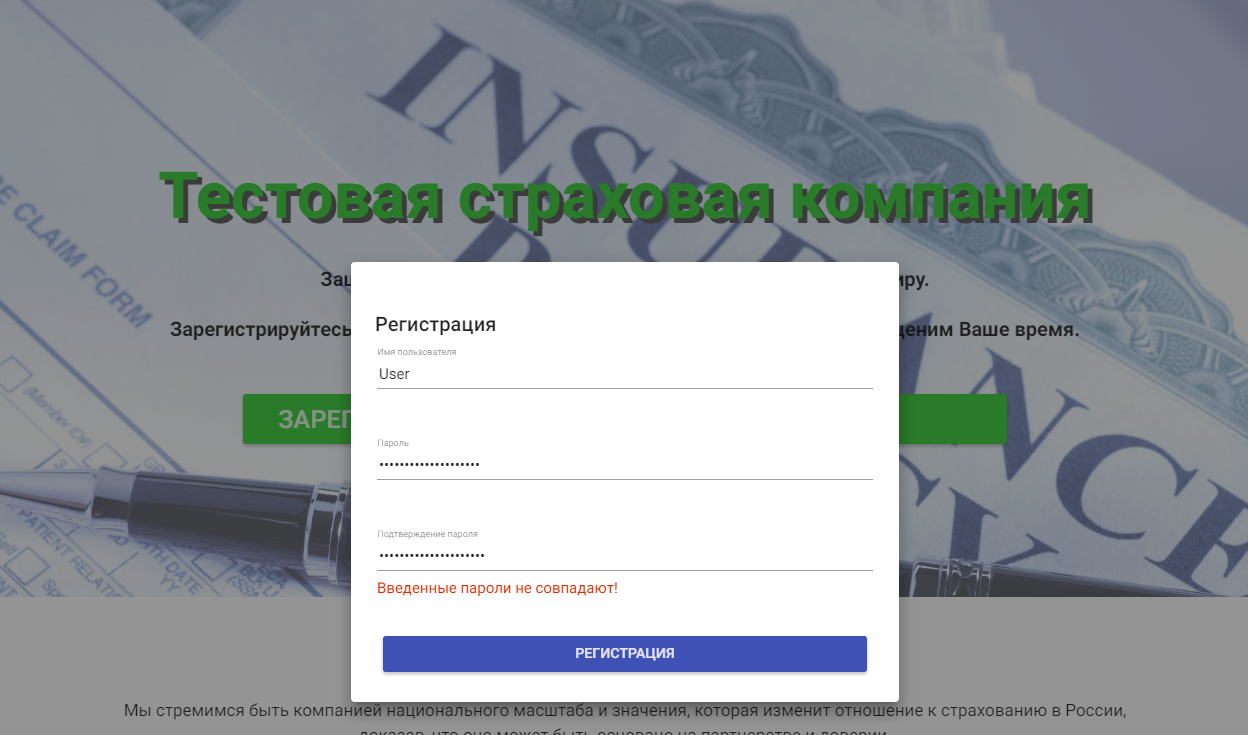


Рисунок 15 – Главная страница демонстрационного приложения.

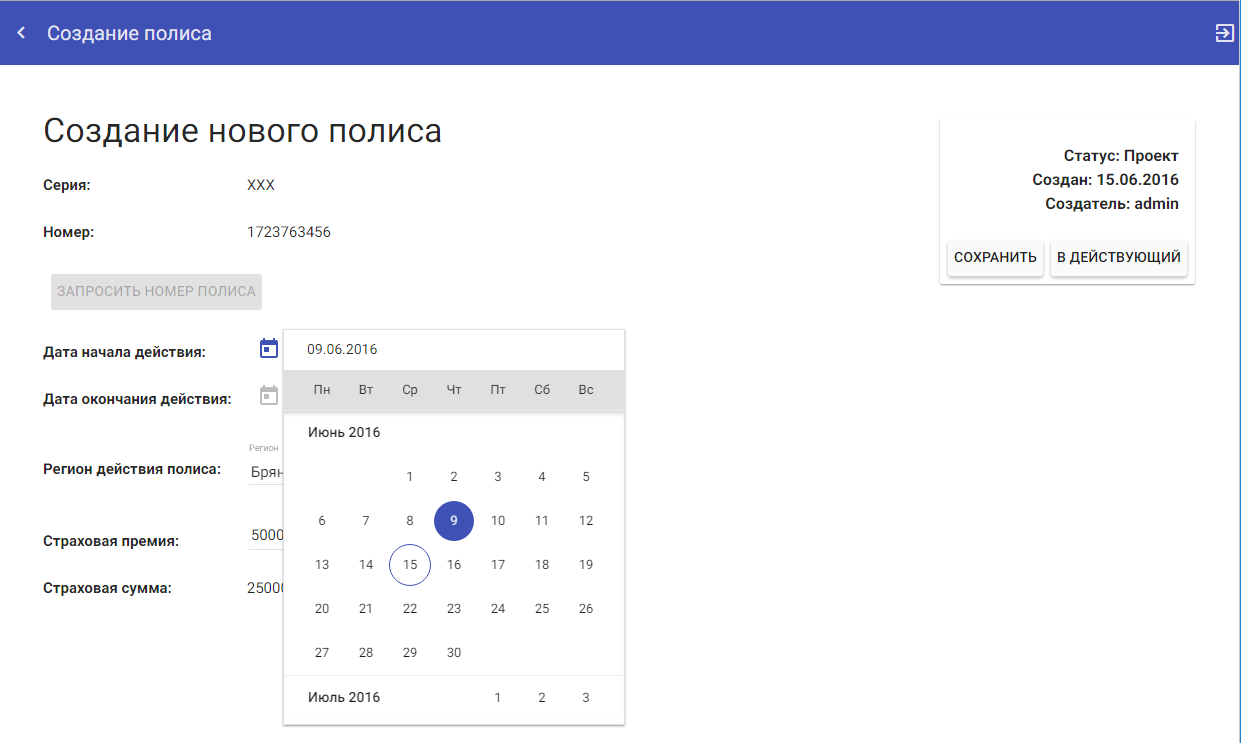


Рисунок 16 – Страница создания страхового полиса.

Каждое из полей имеет свою логику. Они могут как генерироваться автоматически, генерироваться по запросу на сервер, вычисляться как на клиентской стороне, так и на серверной. Отдельно стоит отметить вложенное поле Регион, которое может быть отредактировано и использует контроллер, отличный от контроллера сущности Страховой Полис.

Создадим сущность и нажмем кнопку сохранить. При сохранении на серверную часть приложения уходит POST-запрос на контроллер страховых полисов для создания новой сущности. С созданной сущностью можно работать дальше, однако суть фреймворка заключается в технических деталях, о которых будет рассказано в следующих двух подразделах.

## Описание контроллеров

В то время, когда создавалась сущность страхового полиса в главе 5.3 через клиентское JavaScript-приложение, внутри использовался наследник модуля Контроллер фреймворка, разрабатываемого в данной магистерской диссертации. Результатом создания полиса является то, что при попытке получить этот полис GET-запросом к соответствующему контроллеру, должны выдаваться данные в формате JSON. Данные можно увидеть на рисунке 17.

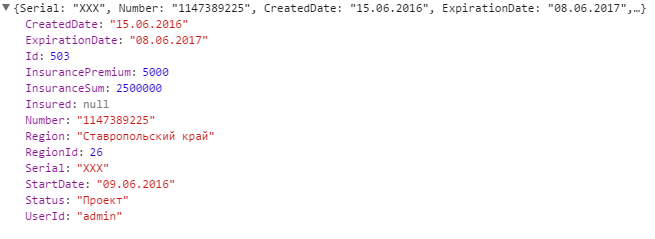


Рисунок 17 – Результат создания страхового полиса в Web 2.0 контроллере.

На рисунке можно увидеть все поля из клиенской части приложения, которыми она может воспользоваться для последующей обработки и отображения пользователю. Все эти данные выдаются по запросу из базы данных и являются актуальными. Однако, как было сказано ранее, эти данные не могут быть использованы другими приложениями без специально запрограммированной для этого API клиентского приложения.

## Описание семантических контроллеров

Семантические контроллеры в данном демонстрационном приложении выдают те же самые данные, что и Web 2.0 контроллеры, добавляя в них семантические данные. Именно они являются ключевой особенностью разрабатываемого фреймворка. Давайте рассмотрим результат обращения к семантическому контроллеру. Его можно найти на рисунке 18.

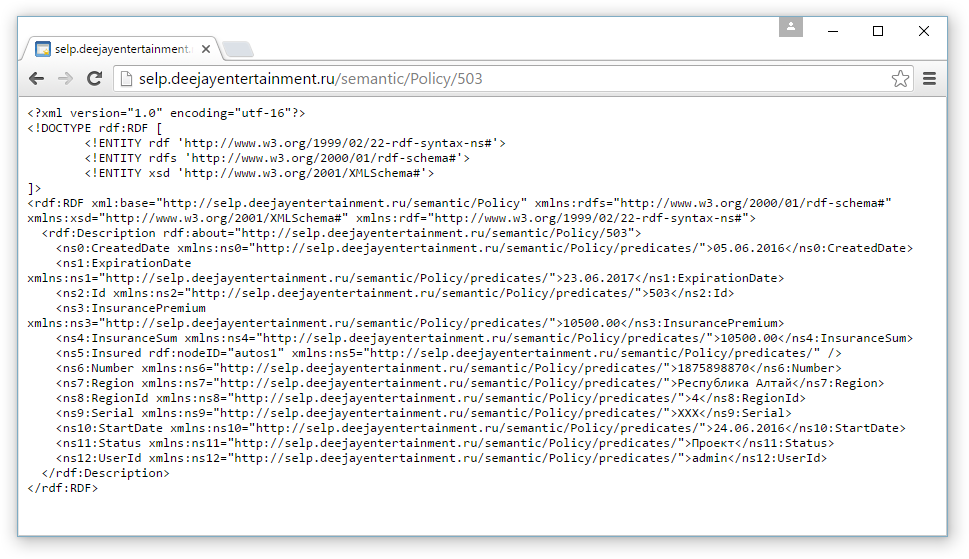


Рисунок 18 – Страховой полис в формате RDF/XML.

На рисунке видно, что данные те же самые, что и в Web 2.0 контроллере, однако находятся в одном из диалектов языка RDF – RDF/XML. Для каждого из свойств объекта генерируется специальный предикат.

## Краткое описание клиентской части приложения

Основой клиентской части приложения является фреймворк Angular.js. Приложение состоит из семи Angular-контроллеров, четырех сервисов, клиентского роутинга и конфигурации.

Контроллеры отвечают за определенный логически связный участок HTML разметки. Они следят за тем, чтобы на этом участке разметки всегда отображались актуальные данные. Также в зону ответственности контроллеров входит непосредственная обработка событий пользователя, таких как щелчок мышью, нажатие определенной клавиши и другие. В данном приложении четыре контроллера отвечают за страницы (создание и изменение полиса – обслуживаются одной разметкой, но разными контроллерами). Два контроллера обслуживают модальные окна регистрации и авторизации. Седьмой контроллер обслуживает все приложение целиком.

Сервисы в Angular служат для хранения логики приложения. В демонстрационном приложении два сервиса из четырех отвечают за работу с серверной частью. Они дублируют методы Web 2.0 API и доставляют данные соответствующим контроллерам по запросу. Третий сервис urlService хранит данные о конкретных адресах серверных методов. Последний сервис dateService является служебным и содержит в себе нестандартные функции работы с датами.

Клиентский роутинг реализован с помощью дополнительного модуля ui-router, который встраивается в Angular.js как один из его компонентов. Его настройка состоит из списка доступных страниц и соответствующих им HTML шаблонов и контроллеров. В конфигурации указаны необходимые форматы дат и выставлены настройки локализации приложения.

# Заключение

В ходе выполнения данной магистерской диссертации были изучены основные понятия и подходы технологии Семантического Веб, выполнен анализ рынка аналогичных программных продуктов, определены требования к фреймворку, выбраны программные и языковые средства для разработки, организован процесс разработки. И в конечном итоге разработан, отлажен, протестирован и оптимизирован фреймворк для автоматической генерации семантических данных в формате RDF средствами рефлексии. В дополнение к фреймворку было разработано, тестовое приложение для демонстрации работы генерации семантических данных, которое также было протестировано, отлажено и оптимизировано.

В процессе выполнения работы были выявлены, обозначены и успешно решены следующие проблемы:

* При разработке связей Контроллер-Репозиторий и Семантический контроллер-Репозиторий было обнаружено, что сущности из базы невозможно полноценно использовать при выдаче клиенту в связи с тем, что клиенту не всегда нужны все поля из сущности. Проблема была решена введением моделей (подробности в главе 4.5).
* Также возникла проблема распознавания поля по имени для ложного удаления и сортировки. Проблема была решена средствами рефлексии в модуле Репозиторий.
* В процессе разработки также встала проблема поддержки различных сериализаций RDF. Проблема была решена использованием интерфейса IRdfWriter, который входит в состав dotNetRdf.
* Для написания юнит-тестов потребовалось эмулировать данные, без обращений к реальной базе данных. Проблема разрешилась использованием библиотеки Moq для создания объектов-заглушек, которые эмулировали обращения к базе данных.

В данной работе было принято решение использовать одну из гибких методологий разработки программного обеспечения – XP (Extreme Programming). Иными словами, фактически, разработка не была поделена на фазы: анализ требований, проектирование, реализация, тестирование и друге. Каждый из этих этапов является непрерывным и требования как к функционалу, так и к архитектуре могут измениться в любой момент. Однако, в целях избежания беспорядка методология разработки XP регламентирует процессы, в том числе и процессы добавления и изменения требований. Подробнее об этой методологии можно узнать в главе 2.2.

По результатам данной магистерской диссертации можно сделать следующие выводы:

* идея Семантического Веб является гениальным решением проблемы стандартизации API, однако ее распространенность, особенно в коммерческом программировании, оставляет желать лучшего;
* для разработки проекта наиболее подходящей платформой является .NET как с технической точки зрения, так и с точки зрения бизнеса;
* для реализации доступа к данным самым предпочительным программным средством является фреймворк объектно-реляционного маппинга EntityFramework;
* для реализации Web 2.0 API самым предпочтительным вариантом платформы является ASP.NET WebAPI 2;
* для реализации Семантического API единственным вариантом является библиотека dotNetRdf;
* при разработке такого типа приложений в заданных условиях наиболее подходит методология разработки Extreme Programming;
* при создании таких расширяемых приложений, как разработанное в данной магистерской диссертации рекомендуется использовать модульную архитектуру, основанную на инверсии зависимостей;
* одним из главных вопросов при разработке расширяемого фреймворка является тестируемость. Она была обеспечена средствами техники написания тестов Test-Driven Development (TDD);
* другим, не менее важным вопросом является производительность. На текущий момент семантическая генерация выполнена с помощью рефлексии, что отрицательно сказывается на производительности. Проблема не является критичной, ее можно решить при помощи кеширования, что и будет сделано в одной из следующей версий.

В данный момент программный продукт выпущен в версии 1.0. Однако, его разработка активно продолжается. В данный момент в продукт планируется добавить такие возможности как:

* поддержка асинхронности;
* автоматическая генерация модели по-умолчанию;
* добавление интерфейса или обертки для DbContext;
* добавление контроллеров, семантических контроллеров и репозиториев из коробки для стандатных часто используемыхся сущностей, таких как Пользователь;
* оптимизация генерации семантических данных на больших объемах данных и больших количествах запросов с помощью кеширования;
* автоматическая генерация Angular-сервисов для клиентской части на основе публичного API.

Результаты работы докладывались и опубликованы в научной литературе, а именно в вестнике СГТУ за 2015 год[3] и в журнале «Наука, техника и образование»[4].

Таким образом, цель работы достигнута: создан фреймворк, который умеет генерировать семантические данные из данных обычного Web 2.0 приложения в автоматическом режиме. Созданные в автоматическом режиме RDF данные проходят валидацию в валидаторе W3C.

# Список использованных источников

1. Вагарина, Н.С., Мельникова, Н.И. Адаптация зарубежного опыта применения технологий семантического Веб в образовании//Материалы конференции: XXIV международная заочная научно-практическая конференция «Технические науки – от теории к практике» . − Новосибирск, 7 августа 2013 г. − С.21-27
2. Kashyap V. Real World Semantic Web Applications / V. Kashyap, L. Shklar. – IOS Press, 2002. – 197 c.
3. Сытник А.А., Мельникова Н.И., Вагарина Н.С., Шульга Т.Э., Апсаликов М.Ю. О создании инструментов изучения семантического веба / Вестник СГТУ, 2015 3(80) − С. 145-150
4. Вагарина, Н.С., Апсаликов, М.Ю. Разработка библиотеки для генерации семантических данных//Научно-методический журнал: Наука, техника и образование №10(16) . − Москва 2015, ноябрь 2015 г. − С.31-35
5. Microsoft Development Network: LINQ to SQL [Электронный ресурс] / MSDN. Режим доступа - https://msdn.microsoft.com/en-us/library/bb386976(v=vs.110).aspx (дата обращения 24.05.2016).
6. Microsoft Development Network: Expression Trees [Электронный ресурс] / MSDN. Режим доступа - https://msdn.microsoft.com/en-us/library/bb397951.aspx (дата обращения 24.05.2016).
7. Официальный репозиторий EntityFramework [Электронный ресурс] / Codeplex. Режим доступа - https://entityframework.codeplex.com (дата обращения 24.05.2016).
8. ASP.NET Usage Statistics [Электронный ресурс] / Build with. Режим доступа - http://trends.builtwith.com/framework/ASP.NET (дата обращения 24.05.2016).
9. Официальный сайт библиотеки RDFMapper [Электронный ресурс]. – Режим доступа: https://rubygems.org/gems/rdf-mapper/versions/0.0.3 (дата обращения 24.05.2016)
10. RubyDoc: RDFMapper – Object-relation mapping for RDF data [Электронный ресурс] / RubyDoc. Режим доступа - http://www.rubydoc.info/gems/rdf-mapper/0.0.3 (дата обращения 24.05.2016).
11. Официальный сайт библиотеки SuRF [Электронный ресурс]. – Режим доступа: https://code.google.com/p/surfrdf/ (дата обращения 13.05.2016)
12. Официальный сайт библиотеки RDFBeans [Электронный ресурс]. – Режим доступа: http://rdfbeans.sourceforge.net (дата обращения 13.05.2016)
13. Aghaei, S. Evolution of the World Wide Web: from Web 1.0 to Web 4.0 / S. Aghaei, M. Nematbakhsh, H.K. Farsani. - International Journal of Web & Semantic Technology (IJWesT) Vol.3, No.1, January 2012. – 10 c.
14. What Is Web 2.0 [Электронный ресурс]. Режим доступа: http://oreilly.com/web2/archive/what-is-web-20.html (дата обращения 24.05.2016).
15. Тим Бернерс-Ли, Джеймс Хендлер и Ора Лассила. Семантическая Сеть [Электронный ресурс]. Режим доступа: http://www.cs.man.ac.uk/~ezolin/logic/semantic\_web\_rus.html (дата обращения 24.05.2016).
16. Berners-Lee, T. The Semantic Web. A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities [Электронный ресурс] / T. Berners-Lee, J. Hendler, O. Lassila. Режим доступа: http://jeckle.de/files/tblSW.pdf (дата обращения 24.05.2016).
17. Hebeler, J. Semantic Web Programming / J. Hebeler, M. Fisher, R. Blace. - John Wiley & Sons, 2011. - 648 с.
18. RDF 1.1 XML Syntax [Электронный ресурс]/W3C Recommendation. – Режим доступа: http://www.w3.org/TR/rdf-syntax-grammar/ (дата обращения 24.05.2016)
19. Beckett, D. Turtle - Terse RDF Triple Language [Электронный ресурс] / D. Beckett, T. Berners-Lee. - Режим доступа: http://www.w3.org/TeamSubmission/turtle/ (дата обращения 24.05.2016).
20. Berners-Lee, T. Notation 3 Logic. [Электронный ресурс] / T. Berners-Lee. -Режим доступа: http://www.w3.org/DesignIssues/Notation3.html (дата обращения 24.05.2016).
21. Методологии разработки программного обеспечения [Электронный ресурс]. Режим доступа: https://habrahabr.ru/sandbox/43802/ (дата обращения 10.05.2016).
22. Agile-манифест [Электронный ресурс]. – Режим доступа: http://www.agilemanifesto.org/iso/en/ (дата обращения 10.05.2016).
23. The Official Scrum Rulebook [Электронный ресурс]. – Режим доступа: https://www.scrum.org (дата обращения 10.05.2016).
24. Don Wells. Extreme Programming: A gentle introduction [Электронный ресурс] / D. Wells. Режим доступа: http://www.extremeprogramming.org (дата обращения 10.05.2016).
25. Rational Software. Rational Unified Process. Best Practices for Software Development Teams [Электронный ресурс] / Rational Software. Режим доступа: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251\_bestpractices\_TP026B.pdf (дата обращения 10.05.2016).
26. World Wide Web Technology Surveys [Электронный ресурс] / W3Techs. – Режим доступа: http://w3techs.com/technologies/overview/programming\_language/all/ (дата обращения 11.07.2015)
27. Microsoft Development Network: The Repository Pattern [Электронный ресурс] / MSDN. Режим доступа - https://msdn.microsoft.com/en-us/library/ff649690.aspx (дата обращения 11.07.2015).
28. Официальный сайт IDE Microsoft Visual Studio 2015 [Электронный ресурс]. – Режим доступа: http://www.visualstudio.com (дата обращения 10.05.2016).
29. Официальный сайт JetBrains [Электронный ресурс]. – Режим доступа: https://www.jetbrains.com (дата обращения 10.05.2016).
30. Официальный сайт библиотеки dotNetRdf [Электронный ресурс]. – Режим доступа: http://www.dotnetrdf.org (дата обращения 10.05.2016)