Министерство образования Республики Беларусь

Учреждение образования  
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Компьютерных систем и сетей

Кафедра Информатики

*К защите допустить:*

Заведующий кафедрой информатики

\_\_\_\_\_\_\_\_\_\_\_\_\_ Н.А. Волорова

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к дипломному проекту

на тему:

**АВТОМАТИЗИРОВАННАЯ СИСТЕМА РАСЧЁТА И СТРАХОВКИ ГРУЗОПЕРЕВОЗОК**

БГУИР ДП 1-10 04 01 00 029 ПЗ

Студент: А.А. Дерид

Руководитель: М.Г. Рогов

Консультанты:

*от кафедры информатики* М.Г. Рогов

*по экономической части* К.Р. Литвинович

Нормоконтролёр: Н.Н. Бабенко

Рецензент:

Минск 2018

Министерство образования Республики Беларусь

|  |
| --- |
| Учреждение образования БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ |

Факультет компьютерных систем и сетей Кафедра информатики

Специальность 1-40 01 03 Информатика и технологии программирования

УТВЕРЖДАЮ

Заведующая кафедрой ИиТП

Волорова Н.А.

20 февраля 2017 г.

**ЗАДАНИЕ**

**по дипломному проекту студента**

Дерида Артема Александровича

1 Тема проекта «Библиотека для межсетевого связывания микросервисов для платформы .Net core» утверждена приказом по университету от 17 февраля 2017г. №328-с

2 Срок сдачи студентом законченного проекта 26 мая 2017 г.

3 Исходные данные к проекту: языки программирования C#, программная платформа Roslyn, фреймворк RabbitMQ, MassTransit, среда разработки Visual Studio 2017.

Назначение разработки: библиотека для межсетевого связывания микросервисов для платформы .Net core

4 Содержание расчетно-пояснительной записки

Введение

1 Программное средство взаимодействия пользователей

1.1 Основные требования к программному обеспечению

1.2 Обзор существующих технических решений

1.3 Выводы и постановка задачи на дипломное проектирование

2 Эргономическое проектирование программного средства взаимодействия пользователей

2.1 Анализ функций и их распределение в проектируемой системе

2.2 Разработка алгоритмов работы пользователей программного средства взаимодействия пользователей

2.3 Разработка эргономических требований и их реализация при разработке сценария взаимодействия пользователей и техники

3 Разработка программного средства взаимодействия пользователей

3.1 Разработка структуры программного средства взаимодействия пользователей и структуры базы данных

3.2 Разработка программных модулей

3.3 Тестирование программных модулей

4 Технико-экономическое обоснование эффективности разработки и использования программного средства взаимодействия пользователей

4.1 Характеристика программного средства

4.2 Расчет затрат на разработку программного обеспечения

4.3 Оценка экономической эффективности программного обеспечения

5 Охрана труда. Обеспечение электробезопасности в офисе при создании программного средства взаимодействия пользователей

Заключение

Список использованных источников

Приложение А (обязательное). Листинг программы

5 Перечень графического материала

Блок-схема алгоритма работы пользователя (ПД) – формат А1, лист 1

Блок-схема алгоритма работы программы (ПД) –формат А1, лист 1

Структура базы данных (ПД) – формат А1, лист1

Диаграмма классов (ПД) – формат А1, лист 1

Схема структурная взаимодействия пользователей и среды на рабочем месте (ПЛ) – формат А1, лист 1

Эскизы рабочих окон программы (ПЛ) – формат А1, лист 1

6 Содержание задания по технико-экономическому обоснованию

Технико-экономическое обоснование эффективности разработки и использования программного средства взаимодействия пользователей

Задание выдал М.Г. Рогов

КАЛЕНДАРНЫЙ ПЛАН

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапов дипломного проекта | Объём этапа, % | Срок выполнения этапов проекта | Примечание |
| Получение задания на дипломный проект |  | 20.02.2017 |  |
| Обзор научно-технической литературы по теме дипломного проекта | 10 | 28.03.2017 |  |
| Разработка эргономического раздела | 20 | 03.04.2017 |  |
| Разработка экономического раздела | 10 | 10.04.2017 |  |
| Разработка раздела по охране труда | 10 | 14.04.2017 |  |
| Разработка программного модуля системы | 30 | 04.05.2017 |  |
| Оформление пояснительной записки | 10 | 08.05.2017 |  |
| Разработка графических материалов дипломного проекта | 10 | 15.05.2017 |  |
| Сдача дипломного проекта |  | 26.05.2017 |  |

Дата выдачи задания 20 февраля 2017 г. Руководитель Рогов М.Г.

Задание принял к исполнению 20 февраля 2017 г Дерид А.А.

**РЕФЕРАТ**

РАЗРАБОТКА БИБЛИОТЕКИ ДЛЯ МЕЖСЕТЕВОГО СВЯЗЫВАНИЯ МИКРОСЕРВИСОВ ДЛЯ ПЛАТФОРМЫ .NET CORE:

дипломный проект / А. А. Дерид. – Минск: БГУИР, 2018, - п.з. – хх с., чертежей (плакатов) – 6 л. формата А1.

Ключевые слова: БИБЛИОТЕКА, МИСКРОСЕРВИСЫ, СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ, МЕЖСЕТЕВОЙ ЭКРАН, DOCKER, КОНТЕЙНЕР, CSHARP, RABBITMQ.

Объектом исследования и разработки библиотека для связывания программных компонентов через сеть интернет.

Целью проекта является разработка программного комплекса, который является библиотекой для генерации связи через сеть интернет между компонентами системы.

При разработке и внедрении приложения использовался такой стек технологий как .Net Core, C#, Roslyn, RabbitMQ, MassTransit.

Данный программных продукт может быть полезен как в разработке новых программных продуктов для предприятий с уже построенной кодовой базой и настроенными бизнес процессами, так и для создания полностью новых продуктов с соблюдением современных методологий разработки сетевого ПО.

В первом разделе данной работы ставится задача дипломного проектирования, устанавливается предметная область разработки, выдвигаются общие требования к созданию приложения, а также определяются перспективы развития первой версии продукта.

Второй раздел содержит общие теоретические сведения о состоянии финансовой грамотности в Республике Беларусь, а также о видах технологий, применяемых при их разработке и внедрении приложения и их особенностях.

Третий раздел посвящен проектированию архитектуры приложения.

В четвертом разделе описана реализация программного комплекса.

Пятый раздел содержит эргономическую оценку информационного приложения.

В шестом разделе приведено технико-экономическое обоснование эффективности разработки и внедрения программного продукта.

В разделе заключение содержатся краткие выводы по дипломному проекту.

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

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc514506818)

[1. Анализ литературы в области разработки микросервисов. 6](#_Toc514506819)

[1.1. Основные понятия и определения. 6](#_Toc514506820)

[1.2. Анализ существующих аналогов 13](#_Toc514506821)

[**1.2.1.** Аналог архитектуры. 13](#_Toc514506822)

[**1.2.2.** Аналог платформы разработки. 13](#_Toc514506823)

[**1.2.3.** Выбор среды виртуализации. 14](#_Toc514506824)

[**1.2.4.** Выбор средства передачи сообщения 14](#_Toc514506825)

[2. Анализ требований к программному средству 15](#_Toc514506826)

ВВЕДЕНИЕ

Одну из самых доминирующих ролей в современном мире разработки программного обеспечения представляют приложения, сетевые сервисы и онлайн сайты в сети интернет.

Одну из самых распространённых топологий данных приложений можно описать как систему трёх взаимодействующих между собой компонентов: пользовательский интерфейс, база данных и серверная часть. Серверная часть обрабатывает поступающие к ней клиентские запросы на обработку, чтение или запись данных и взаимодействует с базой данных для хранения информации. Серверная часть чаще всего представляет собой место для обработки доменной логики и доменной модели.

Устоявшаяся модель разработки серверных приложений зачастую представлена в виде единого приложения, процесса, исполняемого на мощной рабочей станции. При изменении доменной логики, доменной модели, усовершенствовании или исправлении работы системы необходимо вносить изменения в программу, обрабатываемую на рабочей станции, что влечёт за собой компиляцию и пересборку всего приложения.

С ростом потребностей бизнеса может не хватать мощности одной единицы развёрнутой системы. Решение этой задачи является горизонтальное масштабирование, иначе говоря репликация приложения и запуск одного программного продукта на разных рабочих станциях, распределяя нагрузку между ними с помощью дополнительной системы.

С течением времени, становится труднее сохранять хорошую модульную структуру, изменения логики одного модуля имеют тенденцию влиять на код других модулей. Масштабировать приходится все приложение целиком, даже если это требуется только для одного модуля этого приложения. Так же не стоит забывать о дополнительном оборудовании и/или программных средствах для обеспечения распределения нагрузки равномерно между рабочими станциями. Развёртывание новых экземпляров требует дополнительных знаний и специалистов, тонкой настройки системы для того чтобы экземпляры работали в идентичной среде.

Ответом на все описанные проблемы является микросервисная архитектура.

Микросервисная архитектура - это очень легковесная и поддерживаемая практика разработки программного обеспечения. Каждый сервис является выделенным процессом, который отвечает принципам единственной ответственности. Сервисы представляют собой сепаратные модули, которые легко покрываются модульным и интеграционным тестирование.

Так же из бонусов данной архитектуры можно выделить более разделяемый процесс разработки каждого из сервисов. Следуя спецификации доменных процессов и моделей, разработкой могут заниматься одновременно разные разработчики и даже разные команды, используя разные языки программирования, платформы, библиотеки и многое другое.

Репликация таких сервисов тоже более удобна, особенно в связи с развитием облачных технологий и контейнеризации, которые позволяют оперативно разворачивать всё локальное окружение для процесса прямо на лету, и реплицируются автоматически в зависимости от нагрузки в текущий момент, что снижает издержки на содержание сетевого оборудования, аренды помещения, покупки серверов, количество персонала.

У данного подхода по мимо плюсов есть и недостатки. Чаще всего микросервисная архитектура используется для публичных точек доступа, что, зачастую, не подходит для специфичных требований к безопасности и устройству внутренней сети крупных предприятий.

Моё решение — это создание библиотеки для связывания микросервисов которые работаю во внутренней сети предприятия с их публичными частями, которые развёрнуты и исполняются в открытой части сети. Связь между публичной и приватной частью сервисов происходит с помощью декорирования сервисов и туннелирования функциональных вызовов между ними. А сама генерация туннелей и поддержка связи осуществляется с помощью современных возможностей платформы .Net Core, языка C#, а также нового открытого для сообщества компилятора от компании Microsoft, под псевдонимом Roslyn, который позволит мне создавать динамические связи между внутренними и внешними компонентами наших веб служб.

1. Анализ предметной области

1.1 Постановка цели разработки

**Микросервисы** – это технология разработки программного обеспечения, одна из разновидностей сервисно-ориентированной архитектуры, который структурирует приложение как совокупность слабосвязанных сервисов.

В разработке с использованием облачных технологий, микросервисов и контейнеризации есть некоторые недостатки. Микросервисная архитектура предполагает публичность, доступность любых зависимых компонент сервиса по отношению к другим, а также постоянный доступ сети интернет. В противном случае, отсутствие соединения по причини технических неполадок, неправильной настройки межсетевого экрана, или просто экранирования части функционала во внутренней сети предприятия приводит к отказам работы системы

Моя разработка призвана решить недостатки архитектурного подхода для построения современных, соответствующих тенденциям приложения, которые смогут обеспечить устойчивую архитектуру, модульную разработку, тестируемость и соответствовать лучшим практикам для разработки программного обеспечения.

Суть проекта в создании библиотеки для межсетевого связывания и декорирования функциональных блоков, которая сможет работать в условиях ограниченного доступа в сеть и будет обеспечивать должный уровень безопасности.

**Межсетевое связывание** – связывание программных служб, расположенных в сети таким образом, будто они исполняются на одной рабочей станции или вовсе эмуляция работы одного логического блока, но разделяя их на связанные между собой реализации.

**Декорирование** – это паттерн проектирования, который предполагает оборачивание некоторого функционала для того чтобы перехватить его вызов, произвести манипуляции над поступающими в логический блок данными, или для корректировки или изменения результата работы логического блока.

Сама библиотека будет разработана на языке C# и построена с учётом дальнейшего выполнения на платформе **.Net Core**.

**.Net Core** – кроссплатформенная (работает на рабочих станциях под управлением системами компаний Microsoft, Apple, а также на некоторых операционных системах семейства Linux) программная платформа, основной задачей которой является исполнение разных управляемых языков программирования, а также ряд функциональных библиотек для их поддержки, разработанная компанией Microsoft как аналог платформы .Net Framework.

Так же библиотека разрабатывается с возможностью исполнение внутри контейнера Docker. Это позволит разрабатывать сепаратные сервисы, которые смогут исполнятся как на единой машине, так и на разных стационарных или облачных хостингах.

**Docker** — программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на уровне операционной системы. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер.

**Хостинг** — это комплекс услуг по предоставлению [ресурсов](https://ru.wikipedia.org/w/index.php?title=%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%80%D0%B5%D1%81%D1%83%D1%80%D1%81%D1%8B&action=edit&redlink=1) для размещения [информации](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F) специальном сетевом оборудовании (сервере) которое подключено к широкополосному интернету, постоянно находится в сети и обеспечивает бесперебойный доступ клиентов.

**Виртуальный выделенный сервер (VPS**) — это комплекс услуг по предоставлению пользователю изолированных ресурсов мощного физического сервера. Виртуальные серверы функционируют независимо друг от друга, так что при размещении на физическом сервере служб они не влияют на производительность других служб и имеют доступ только к выделенным для них ресурсам (Как аппаратным, так и программным).

Технология контейнеризации Docker так же примечательно удобством доставки конечного программного продукта на удалённый облачный сервер. Каждый образ настроен таким образом, чтобы содержать все необходимые для работы программного продукта элементы его окружения, что позволит сократить расходы на настройку оборудования и администрирования, а также позволит удобно доставлять конечный продукт, готовый к развёртыванию для заказчика.

Сама виртуализация контейнеров Docker является так же новшеством, в развитии программного обеспечения. Аналоги от разнообразных компаний, которые разрабатывают продукты для создания и работы виртуальных машин, такие как Oracle **- VM VirtualBox**, **VMware**, Vagrant производят полную программную, а зачастую и аппаратную виртуализацию гостевой операционной системы, на базе операционной системе – хосте. Данных подход очень трудозатратен в плане разворачивания операционной системы, а также очень трудоёмкий в плане обеспечения производительности работы системы.

В противовес этому контейнеры Docker производят виртуализацию на уровне операционной системе. По сути своей они являются лишь программной оболочкой для изоляции сепаратных процессов, сервисов и услуг на базе ядра системы-хоста, которые обеспечивают уникальное окружения и среду исполнения для программного продукта, а также систему виртуализации сети.

**Виртуализация на уровне операционной системы** — метод виртуализации, при котором ядро операционной системы поддерживает несколько изолированных экземпляров пространства пользователя, вместо одного. Эти экземпляры (часто называемые *контейнерами* или *зонами*) с точки зрения пользователя полностью идентичны реальному серверу.

Одно из условий исполнения программного продукта является создание возможности использования одного сервиса, как единой логический блок с доступом внутри и с наружи закрытой сети крупного предприятия, с ограничениями на доступ к сети, настроенными межсетевыми экранами, специфичной маршрутизацией, частью заблокированных портов и повышенными требованиями безопасности, устойчивости к вредоносным атакам. Стоит учитывать, что сами внутренние сети предприятия могут и вовсе работать по принципу **DMZ**.

**DMZ** (от англ. *Demilitarized Zone* — демилитаризованная зона, ДМЗ) — сегмент сети, содержащий общедоступные сервисы и отделяющий их от частных. В качестве общедоступного может выступать, например, веб-сервис: обеспечивающий его сервер, который физически размещён в локальной сети (Интранет), должен отвечать на любые запросы из внешней сети (Интернет), при этом другие локальные ресурсы (например, файловые серверы, рабочие станции) необходимо изолировать от внешнего доступа частично или полностью.

**Цель ДМЗ** — добавить дополнительный уровень безопасности в локальной сети, позволяющий минимизировать ущерб в случае атаки на один из общедоступных сервисов: внешний злоумышленник имеет прямой доступ только к оборудованию в ДМЗ.

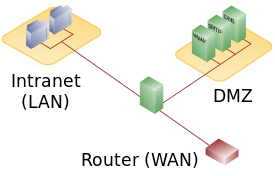
****

Рисунок 1.1 Структура сети с использованием DMZ

DMZ создаётся с помощью межсетевого экрана, который обеспечивает разграничение ресурсов и возможность установления доступности ресурсов из разных участков сети.

**Межсетевой экран**, **сетевой экран** — программный или программно-аппаратный элемент компьютерной сети, осуществляющий контроль и фильтрацию проходящего через него сетевого трафика в соответствии с заданными правилами.

**Маршрутизация** — процесс определения маршрута следования данных в сетях связи.

Для исполнения всех требований мною было продумано решение, которое соответствовало бы всем необходимым условиям. Подход заключался в создании между двумя или более разделёнными частями одного сервиса туннеля, или моста, построенном на платформе обмена сообщениями RabbitMQ и фреймворка MassTransit для более удобного обращения к вышеназванной платформе, который работал бы на специфичных, выделенных ему сетевых портах, а также на основе стандарта AMQP.

**Туннелирование** в компьютерных сетях — процесс, в ходе которого создается защищенное логическое соединение между двумя конечными точками посредством инкапсуляции различных протоколов.

**RabbitMQ** — платформа, реализующая систему обмена сообщениями между компонентами программной системы на основе стандарта **Advanced Message Queuing Protocol**.

MassTransit — это бесплатный фреймворк с открытым исходным кодом для создания легкой шины сообщений и созданный для разработки распределённых приложений построенных на платформе .Net Framework

**AMQP**— открытый протокол для передачи сообщений между компонентами системы. Основная идея состоит в том, что отдельные подсистемы (или независимые приложения) могут обмениваться произвольным образом сообщениями через AMQP-брокер, который осуществляет маршрутизацию, возможно гарантирует доставку, распределение потоков данных, подписку на нужные типы сообщений.

Сама система обмена сообщениями RabbitMQ построена идентично структуре данных – очередь. Каждое сообщение, которое посылается от отправителя до получателя хранится в очереди, пока одна из сторон либо не получит сообщение, либо не пошлёт токен возврата, или токен отказа, что удалит сообщение из очереди.

**Токен** возврата в сети – сообщение об ошибке, после получения которого, система рекомбинирует сообщения внутри одной очереди и помечает сообщение, надлежащее к извлечению из очереди как устаревшее. Когда очередь выдачи сообщения дойдёт до помеченного сообщения оно автоматически будет отброшено и удалено.

Для каждого функционального вызова мною было предусмотрено создание выделенной очереди. Это позволило уменьшить издержки на фильтрацию и перенос сообщений, которые адресованы не активному вызову в конец очереди и её чрезмерной сортировке.

Маршрутизация же вызовов происходит самой RabbitMQ, на основе полученных имён системных вызовов, или на основе статических типов сообщений, что необходимо для вызова делегатов и событий платформы языка C#.

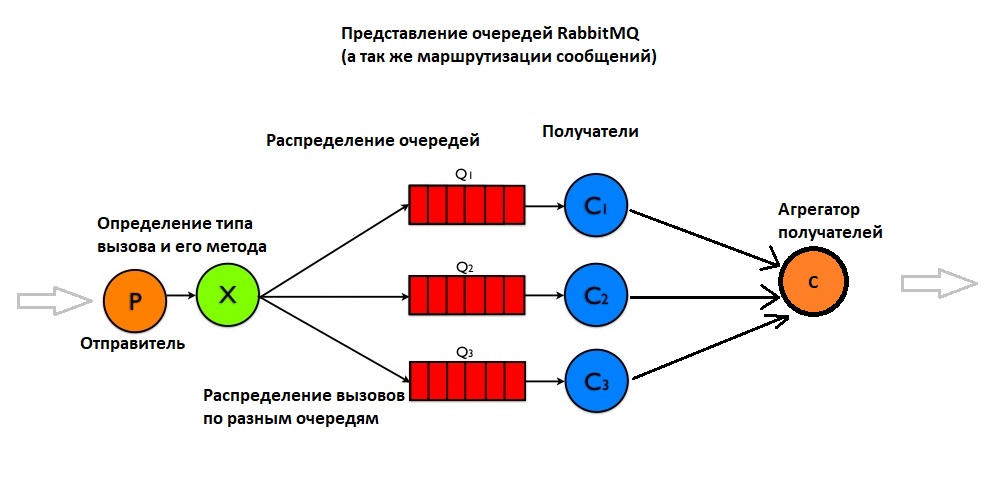


Рисунок 1.2 Представление очередей RabbitMQ

Схема, иллюстрирующая работу библиотеки и сервисов с учётом закрытости сети предприятия

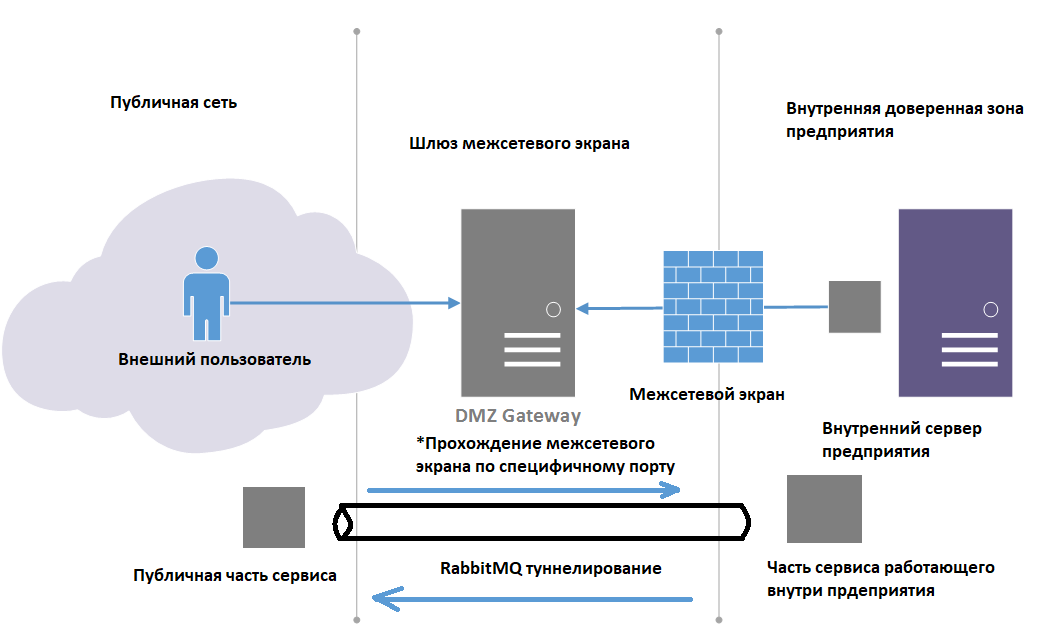


Рисунок 1.3 Обзорная схема работы системы с использованием библиотеки

Сама же библиотека будет оборачивать любые определённые клиентами программистами микросервисы после подключения библиотеки и инкапсулирования необходимых для работы функциональных методов. Декорирование же будет происходить посредством использования **.NET Compiler Platform**, более известной под кодовым названием **Roslyn**.

**Roslyn** – это набор открытых для сообщества разработчиков компиляторов и статических анализаторов, разработанных для языков VS.NET и C# компанией Microsoft. Компиляторы доступны как, традиционно, с помощью командной строки, но и с помощью нового API, который доступен внутри кода. Второй подход примечателен тем, что предоставляет модули для синтаксического (лексического) анализа кода, семантического анализа, динамической компиляции на CIL и эмиссии кода.

Используя возможности Roslyn, мы можем создавать динамические сборки в памяти, которые и будут реализовывать паттерн декоратор и перехватывать функциональные вызовы обёрнутых сервисов, и туннелировать их по сети с помощью RabbitMQ.

Подводя итоги описанных целей и реализаций можно дать характеристику моему программному продукту:

Я разрабатываю библиотеку для декорирования легковесных сервисов на платформе .Net Core, с возможностью исполнения их на рабочих станциях с ограниченным доступом в сеть Интернет и разделением их частей для внутренних сетей предприятий и для публичных серверов. Библиотека построена на принципе декорирования, инкапсулированию функциональных вызовов и эмуляции их вызовов на противоположной части. Туннелирование между частями осуществляется по протоколу AMQP и с помощью платформы RabbitMQ, к которой я обращаюсь с помощью фреймворка MassTransit, а само декорирование производится с помощью Roslyn и создание динамических сборок в памяти, а также их исполнению.

2. Анализ существующих аналогов

С учётом специфичных требований, комплексных систем предоставляющих полного аналога для работы с платформой .Net не существует. Следует рассматривать каждую из частей библиотеки и искать её аналоги.

2.1. Аналог архитектуры.

Как было описано ранее аналогом микросервисной архитектуры является использование монолитной архитектуры. В главах введение и предыдущей подглаве, в ходе построения задачи, были описаны недостатки такой архитектуры, так что не имеет смысла их описывать вновь.

2.2. Аналог платформы разработки.

Выбор ключевой платформы был выбран с учётом возможностей языка, с учётом его поддержки, количество доступных для использования библиотек.

Я считаю язык C# отличным выбором для данных целей. C# является высокоуровневым языком, который имеет огромной спектр возможностей для разработки ПО разного назначения, и поддерживается компанией Microsoft. Этот язык достаточно быстро развивается, не смотря на устоявшуюся кодовую базу. Он отвечает всем современным требованиям разработки. Его устройство и самый популярные инструменты для разработки являются отличным подспорьем для разработчика, и позволяют увеличить скорость написать проектов, но и сохранить возможность отслеживания ошибок, благодаря метаданным сборок, статической типизации, лексических и статических анализаторов.

Не смотря на то что язык C# является управляемым языком, он позволяет и работать с низкоуровневыми вызовами, ссылками, управлению памяти и другими подходами, которые, хоть и признаны небезопасными, но порой являются необходимыми для специфичных проектов, или для связи с неуправляемыми ресурсами и языками программирования.

Платформа .Net Core является современной, усовершенствованной и отвечающей современным требованиям платформой для исполнения проектов написанных на языке C#. Платформа является кроссплатформенной (т.е. может исполнятся в разных средах), что открывается возможности к разработке для разных операционных систем, и возможности разработки для разворачивания и работы программного продукта внутри контейнера Docker

2.3. Выбор среды виртуализации.

Я ранее описывал другие способы виртуализации и основные продукты на рынке виртуализации. Одним из самых распространённых аналогов, а также продуктов, с которым чаще всего сравнивают является программный продукт Vagrant от компании HashiCorp.

В общем и целом, под Docker и Vagrant имеют в виду просто средства виртуализации с разной степенью этой виртуализации, Docker - как средство виртуализации на уровне ядра операционной системы – хоста.

Vagrant - как полную виртуализацию (машина виртуализуется полностью, и ей предоставляются виртуальные оперативные память, процессор, и прочие устройства). ВМ думает, что общается с аппаратными средствами, которые виртуализируются программно. Второй способ виртуализации является более функциональным, но также является гораздо более трудозатратным.

2.4. Выбор платформы передачи сообщений

Одним из самых распространённых средств передачи сообщений который противопоставляют RabbitMQ является Amazon SQS. Amazon SQS – это сервис, позволяющий создавать и работать с очередями сообщений. ASQS является очень близким аналогом к RabbitMQ, но является платным решением, которое не обеспечивают достаточную производительность. Ниже будет происходить сравнение данных платформ и рассмотрю схему развёртывания и работы с очередью сообщений Amazon SQS:

2.4.1. Amazon SQS

Схема развёртывания данной платформы очень нетривиальна – необходимо только оплатить услугу и настроить работу очереди. Так же доступны настройки межсетевого экрана для обеспечения безопасности.

Стандартные шаги работы с готовой очередью SQS такой:

1. Отправитель для отправки сообщения Получателю в очередь должен знать её URL. Затем, используя команду отправки сообщения, Отправитель добавляет сообщение в очередь, соответствующую указанному URL.
2. Получатель отслеживает доступность какого-либо сообщения, которое предназначалось ему в очереди, и в случае нахождения вызывает метод получения приоритетного сообщения. Приоритет же определяется датой отправления сообщения.
3. Как только сообщения получено, оно блокируется на некоторое время чтобы избежать повторного получения.
4. После удачной обработки сообщения Получатель использует команду удаления сообщения из очереди. Если же во время обработки была вызвана ошибка и не была вызвана команда удаления, то по истечению установленного лимита времени сообщение будет разблокировано и отправится обратно в очередь.

Таким образом, в среднем для отправки и обработки одного сообщения необходимо 3 вызова API. Amazon SQS является платным решением. Используя SQS, вы платите за количество вызовов API, а также за трафик между регионами.

2.4.2. RabbitMQ

Процедура развёртывания платформы RabbitMQ заключается в настройке выделенного сервера. Платформа RabbitMQ работает на языке программирования Erlang так что нам так же необходимо будет установить исполняемую среду Erlang OTP. Этот язык имеет официальную поддержку и может исполнятся на всех самых популярных операционных системах, так что мы не имеем проблем и ограничений в выборе операционной системе для хостинга данной части.

В качестве минусов стоит лишь выделить тот факт, что в случае отказа сервера обработки компонента RabbitMQ все Отправители будут накапливать сообщения из-за недоступности очереди, а также в случае дальнейшей доступности этого компонента могут быть перегружены Получатели из-за перегруженного потока сообщений.

2.4.3. Сравнение производительности.

В качестве следующего пункта для сравнение будет использован синтетический нагрузочный тест по отправке сообщений.

Тестирование производилось в нескольких конфигурациях, и проверка велась в некоторых тестовых сценариях:

1. Отправка сообщений разных размеров:
   1. Сообщения размером 16 байт
   2. Сообщения размером 1 килобайт
   3. Сообщения размером 64 килобайта (Максимальный размер сообщения для платформы Amazon SQS).
2. Использование аутентификации при отправке сообщения:

2.1. Единовременная аутентификация и дальнейшая отправка сообщений.

2.2. Прохождение процесса аутентификации перед процессом отправки каждого сообщения.

1. Проверка отправки сообщений в среде без подключения или с подключением с возможными перебоями до сервера обработки очереди.
2. Одновременная отправка и получение сообщений из одной очереди.
3. Одновременная отправка и получение сообщений из нескольких очередей.
4. Отправка сообщений с нескольких Отправителей для единственного Получателя.

Результаты тестирования:

1. Сравнение скорость обработки запроса, обобщенные результаты:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тест | Сообщений в секунду | Время выполнения запроса | | | |
| Среднее, мс | Минимальное, мс | Максимальное, мс | 90%, мс |
| Amazon SQS Отправка | 318 | 16 | 10 | 769 | 20 |
| RabbitMQ Отправка | 1894 | 2 | 0 | 11273 | 0 |
| Amazon SQS Получение | 178 | 25 | 17 | 721 | 28 |
| RabbitMQ Получение | 1217 | 3 | 0 | 3817 | 19 |

Из данной таблицы можно установить, что RabbitMQ имеет куда большую производительность нежели Amazon SQS, но также хорошо бросается в глаза максимальное время выполнения работы. В случае работы ASQS максимальное время работы в 48 раз больше чем среднее и в 38 чем большая часть запросов была выполнена. В случае же RabbitMQ, не смотря на очень низкое среднее время выполнения можно заметить огромные провалы в производительности и увеличение времени простоя в 5000 раз. Так же сравнение среднего времени и времени выполнения 90% можно установить, что RabbitMQ работает скачкообразно и не так стабильно в сравнении с ASQS, но в подавляющем количестве тестов значительно быстрее.

1. Сравнение скорости обработки запроса в зависимости от размера файла:

Исходя из результатов тестирования можно аргументировать выбор определённой платформы. Мной был сделан выбор в пользу в RabbitMQ. Не смотря на возможные ситуации простоя системы, общая скорость работы, а также скорость работы с большими объёмами данных значительно выше чем у аналога от компании Amazon. К тому же RabbitMQ является абсолютно бесплатным решением.

2.5. Выбор фреймворка для удобной работы с RabbitMQ

Следующим критерием для рассмотрения аналогов является выбор фреймворка для удобной работы с платформой RabbitMQ. Стоит сказать, что RabbitMQ имеет готовую клиентскую библиотеку, но она предоставляет лишь базовые низкоуровневые вызовы что вызвало бы трудность связанные с увеличением времени разработки программного продукта. Выбор производился среди более высокоуровневых надстроек над данной платформой. Изучая предметную область было найдено 2 самых популярных и отвечающих требованием решения: фреймворки MassTransit и EasyNetQ. Оба фреймворка являются проектами с открытым исходным кодом, оба имеют приблизительно одинаковую популярность. Но стоит отметить что MassTransit имеет более подробную документацию, более активное сообщество, чаще обновляется, а также имеет полностью свободную лицензию в отличии от EasyNetQ. Так как мой продукт имеет ограничения на лицензию для дальнейшего беспрепятственного распространения, а также принимая во внимания вышеперечисленные факторы был сделан однозначный выбор в пользу фреймворка MassTransit.

3. Архитектура программного продукта.

В данной главе рассматриваются этапы проектирование программного продукта, а также уточняются требования при разработке целевого программного продукта.

* 1. Инструментарий, использовавшийся при разработки программного продукта.

В данной подглаве будут перечислены и аргументированы инструменты разработки, использовавшиеся в процессе создания программного продукта.

3.1.1. Visual Studio 2017

Так как в качестве языка разработки был выбран язык C# от компании Microsoft, было бы целесообразно использовать среду разработки от той же компании. Visual Studio 2017 является отличным продуктом по написанию программного кода для платформы .Net, для работы с неуправляемыми ресурсами и языком C++, а так же для разработки веб-приложений с использованием таких языков как JavaScript, Typescript и таблиц стилей CSS и SCSS. В продукт встроено большое количество модулей, которые позволяют производить статический анализ кода, отладку приложения, его удобную компиляцию. Так же представлены средства по улучшению качества кода, оптимизации зависимостей. Visual Studio позволяет выполнять модульное и интеграционное тестирование программных средств автоматически в многопоточном режиме после каждой компиляции приложения, имеет приятный и удобный интерфейс по выводу основной информации о процессах компиляции, ошибках компиляции и ошибках выполнения. Немаловажным фактом будет то, что расширенная версия данного продукта, которая носит приставку Professional позволяет так же производить полный мониторинг использования системных ресурсов, сборки мусора, сетевых обращений и других необходимых возможностей.

Новейшая версия данной среды разработки позволяет удобно использовать и комбинировать контейнеры Docker, автоматически производя компиляцию, публикацию и упаковку всех необходимых программных продуктов в контейнер докер, построенный на основе нескольких образов.

3.1.2. ReSharper Ultimate

ReSharper – дополнительный компонент (плагин) для интегрированной среды разработки Visual Studio который создан компанией JetBrains для повышения разработчиков, использующих языки C# и VB.NET, статического анализа кода, генерации кода, рефакторинга, оптимизации процессов разработки. Так же данный компонент позволяет просматривать декомпилированный код для более подробного изучения интерфейсов используемых компонентов, упрощает юнит тестирование и интеграционное тестирование при использовании сред MSTest и NUnit. Позволяет следит за сценариями сборки MS Build и NAnt.

3.1.3. Source Tree

Source Tree от компании Atlassian – это бесплатный графический интерфейс для работы с системами управления версиями Git и Mercurial. Данный клиент является кроссплатформенным, поддерживает работу с тегами и версиями продукта, что является очень важным инструментарием для нашей задачи, поскольку версионность модулей библиотеки очень важное требование, которое связано с автоматическим обновлением пакетов в Nuget.

* + 1. GitLab

GitLab – это система управлениями программными репозиториями Git а также одноимённый сайт по управлению этой системой. В качестве дополнительных возможностей GitLab предоставляет возможность конфигурирования процессов сборки и обновления версий продукта с помощью специальных файлов конфигурации с расширением yaml.

* 1. Обзор модульной системы библиотеки

Модульный подход является очень распространённым подходом при создании программных продуктов. Этот подход представляет программу как совокупность небольших независимых или слабо зависимых блоков, которые называют модулями. Структура и поведение модулей подчиняется определённым правилам. Данный подход обеспечивает очень хорошие возможности для модульного тестирования, снижает возможность появления критических ошибок из-за которых придётся заниматься переделыванием больших частей программы, позволяет разрабатывать их отдельно разными разработчиками или группами разработчиков и позволяет пере использовать модули в разных библиотеках независимо друг от друга.

В моём программном продукте была продумана и построена такая модульная система, основанная на разбиение каждого модуля как отдельного проекта для возможного дальнейшего их распространения и внедрения в качестве зависимостей в рабочем проекте. Данный подход позволяет добавлять зависимости и загружать только те модули, которые необходимы для конкретной задачи.

Список и расшифровка модулей:

1. Interfaces – модуль для сохранения всех интерфейсов и общих моделей, которые используется в других модулях. Данные сущности вынесены в отдельный модуль потому что являются зависимыми во всех остальных модулях.
2. MassTransit Adapter – модуль который содержит общую логику передачи сообщений с помощью фреймворка MassTransit и платформы RabbitMQ. В данном модуле реализованы все необходимые классы и методы для корректной работы MassTransit, а также написана логика по логгированию работы очереди и система оповещения об ошибках. Так же не маловажной частью модуля является анализ типов, которые в дальнейшем будут сериализироваться и передаваться с помощью платформы RabbitMQ и маршрутизация очередей по типу сервиса, который она обслуживает и по типу класса – сообщения, который будет передаваться в очередь.
3. Proxy Generator – данный модуль содержит логику по генерации динамических прокси классов для конкретного выбранного сервиса. Модуль является одним из самых важных и приоритетных в работе конечного программного продукта. Кроме логики по декорированию сервисов, модуль так же содержит некоторые необходимые модели для корректного связывания и загрузки необходимых для работы с MassTransit зависимостей в динамическую сборку в памяти.
4. Client Factory – модуль содержащий логику по генерации объектов, которые являются фактически реализациями интерфейса того сервиса, который мы декорировали с помощью модуля Proxy Generator и регистрации его в качестве Получателя. Модуль так же содержит реализации инструментов для статического анализа кода. Данные реализации позволяют узнать все необходимые типы, вычленить зависимости, определить методы, события, свойства и поля целевого класса-сервиса.
5. Global Tests – глобальный модуль содержащие модульные и интеграционные тесты для каждой части системы. Иерархия модуля разбита на подмодули каждый из которых содержит тесты для определённого компонента система. Часть из тестов представляют собой эмуляцию рабочих процессов, генерации и привязки необходимых компонентов и интеграционное тестирование их взаимодействия. Модуль настроен таким образом, чтобы запускаться либо после каждой перекомпиляции одной из вышеперечисленных библиотек, чтобы проверить их работоспособность. Второй способ запуска тестов – отслеживание изменения версии пакетов. Так как все модули будут поставляться в качестве пакетов, и часть из модулей содержит зависимости на Nuget пакеты MassTransit и RabbitMQ, данный подход позволяет проверить работоспособность системы при обновлении части модулей после изменений, совершенных другими разработчиками, или при обновлении версий главных библиотек.

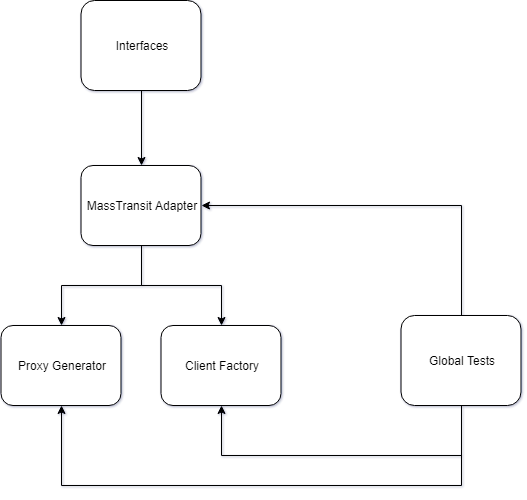


Рисунок 2.1 Схема зависимостей модулей

* 1. ОПИСАНИЕ АЛГОРИТМОВ РЕАЛИЗУЮЩИХ БИЗНЕС-ЛОГИКУ

3.3.1 Обобщенный алгоритм работы программы

Для использования библиотеки для конкретного микросервиса или нескольких микросервисов нам необходимо пройти 2 этапа:

1. Реализовать класс-контейнер для наших сервисов, для проекта, который находится внутри DMZ, у которого будет реализован интерфейс IServiceImplementation. Включение этого сервиса вынудит нас реализовать 2 метода: OnStart и OnSuspend для каждого из сервисов, которые включены в контейнер. Первый метод OnStart - подразумевает вызов статического метода у статического класса ProxyBridge.SubscribeMethodsToRespond и вызова второго метода ProxyBridge.ConnectEventsToProxy. Оба этих метода создают динамическую сборку, привязанную к конкретному сервису и очереди, которая будет декорировать и перехватывать вызов всех методов или событий наших сервисов и передавать их в очередь RabbitMQ. Второй же метод - OnSuspend призван для освобождения всех неуправляемых ресурсов, в том числе соединений с шиной RabbitMQ.
2. На стороне приложения, которая должна находится за пределами DMZ нам необходимо либо использую DI контейнер, либо реализовав все зависимости вручную создать экземпляр класс СlientFactory. После создания экземпляра мы можем вызывать у него метод GetClientInstance который будет создавать клиентскую сторону для любого микросервиса, реализованного на стороне DMZ.

Стоит отметить что клиенту-программисту стоит позаботится о том, чтобы обе части данного сервиса реализовывали паттерн синглтон. Данный паттерн подразумевает что экземпляры нашего класса должны быть доступны на всём жизненном цикле приложения и не должны обрабатывать сборщиком мусора. Данная конструктивная особенность связана с тем, что каждый экземпляр определённого класса-сервиса привязывается к определённой очереди. В случае недоступности одной из сторон очереди могут активно заполнятся и быть переполнены. А сама операция создания, генерации динамической сборки и соединения её с определённой очередью RabbitMQ является очень трудоёмкой операцией. Идентификация и маршрутизация сервисов происходит посредством анализа их интерфейса, а также анализом типов возвращаемых значений. Для каждого метода или события которое будет найдено компонентом Type Analyzer без создана отдельная очередь.

Ниже представлена диаграмма связи части сервиса на стороне DMZ и на публичной стороне:

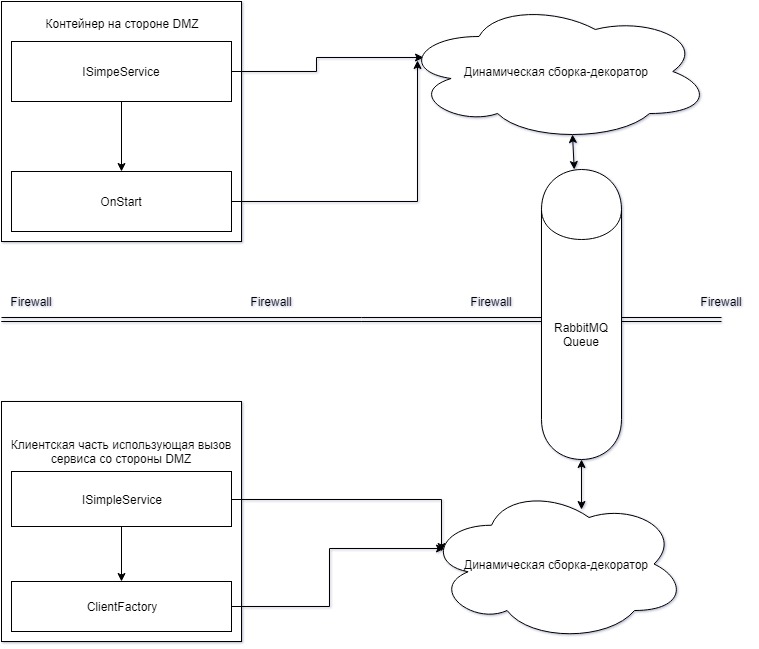


Рисунок 3.2 Схема связи внутренней и внешней части сервиса

3.3.2 Описание генерации динамических сборок для определённого сервиса.

Каждый сервис, для которого мы хотим создать клиентскую часть за пределами DMZ должен иметь собственный интерфейс. Это ограничение или особенность связаны с работой анализатора кода, который будет сообщать компоненту создания декоратора все известные ему методы, поля и свойства. Также это позволит сделать публичными только те методы, которые необходимо. Например, если у нас есть сервис UserService который имеет методы GetUser, UpdateUser, DeleteUser, MadeUserAdmin и этот сервис реализует интерфейс IPublicUserService который реализует только первые два из этих методов, то на публичной стороне будут доступны только они, и только служебные программы, работающие внутри внутренней сети предприятия, будут иметь доступ к удалению пользователя, либо к наделению его новыми возможности и ролями.

Так же анализатор кода делает обёртку только над публичными методами класса-сервиса. Это так же связано с обеспечением большей безопасности и скрытием внутренней реализации части сервиса, работающей внутри закрытой сети.

После того как анализатор типов проводит анализ интерфейса, начинает работать MassTransitAdapter. Данный тип анализирует полученное имя интерфейса, полученные методы и их возвращаемые значения. Эти параметры будут иметь ключевые роли в процессе маршрутизации и определённая необходимой очереди для каждого сервиса. Данный подход позволит нам иметь множество методов с одной сигнатурой, но разными возвращаемыми значениями, что даст возможность использовать принцип объектно-ориентированного программирования, которое называется инкапсуляция.

Ниже представлена схема работы анализатора типов:

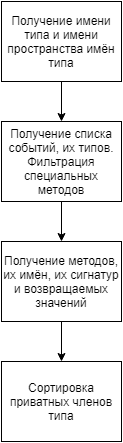


Рисунок 3.3 Схема работы анализатора типов

После этого шага начинается процесс привязки клиентской части к нашей очереди. Вызов метода у экземпляра вышеописанного класса ClientFactory.GetClientInstance будет так же вызывать работу типа Type Analyzer. После шагов анализа кода и определения маршрутизации очереди в работу вступает компонент Proxy Generator. Данный компонент на основе информации о всех типах и именах переменных, интерфейсов и методов сервиса начинает поэтапно производить генерацию кода с помощью компилятора Roslyn. Данные этапы можно расположить в следующем порядке:

1. Создание динамического генератора кода Roslyn для заданного языка – C#.
2. Соотношение типов методов и создание имён типов-сообщений для каждого из методов.
3. Создания списка NamespaceImportDeclaration для определения типов из внешних зависимостей и их дальнейшей загрузки в нашу динамическую сборку.
4. Перечисления всех методов исходного интерфейса, для последующей генерации методов-декораторов для работы с очередью.
5. Получение списка событий, для дальнейшей обработки их делегатов, возвращаемых значений и генерация методов-декораторов для системных скрытых методов RaiseEvent, AddEventListener, RemoveEventListener для работы с очередью.
6. Создания конструктора, в котором будет происходить инициализация шины сообщения для дальнейшей работы с RabbitMQ.
7. Определение списка переменных и событий и инициализация их значениями по умолчанию.
8. Генерация типа SyntaxNode который в дальнейшем после компиляции и будет представлять наш готовый тип.
9. Добавление списка внешних сборок как доверенных сборок, чтобы избежать возможных проблем с компиляцией.
10. Создание экземпляра типа CSharpCompilation который будет производить компиляцию нашего типа.
11. Логгирование промежуточного результата типа. Данный шаг является не обязательным и его можно пропустить если выставить определённые настройки в конфигурационном файле. Но для начинающих разработчиков клиентов, а также для меня лично этот этап был необходим чтобы получить специальные текстовые файла, один из которых содержит форматированный код C# и представляет результат нашей генерации, а второй содержит рекомендации и ошибки от компилятора Roslyn отформатированные сходим образом, как это делает Visual Studio 2017 во время обычной сборки какого-либо кода на языке C#. Поскольку наш тип в дальнейшем будет представлять из себя динамическую сборку в памяти это один из немногих способов проверить его генерацию и найти возможные логические или синтаксические ошибки.
12. После этапа логгирования происходит создание dll потока в памяти ЭВМ, компиляция нашей сборки и загрузка её в память, а также возвращается ссылка на готовый тип из этой сборки для дальнейшего использования в программе.

Та ссылка на объект в динамической сборке которую мы получили после выполнения метода GetClientInstance и является нашим клиентом для сервиса и который может находится и работать в публичной части нашего приложения.

1. Тестирование программного продукта

В ходе разработки моей библиотеки я следовал принципу TDD (c англ. test-driven development) **–** который можно перевести и раскрыть суть на русский язык как разработка через тестирование. Данных подход базируется на небольших этапах разработки: с начала пишется тест, которые покрывает поведение будущего компонента, затем производится создание прототипа, которые позволит пройти этот тест и предыдущие если они были, и после этого производится дальнейший рефакторинг и улучшение качества кода. Данный подход позволяет создавать очень хорошие программные продукты, отвечающие всем стандартам качества работы и качества написания кода. В моём случае этот выбор был критичен, так как любое изменение в модулях генерации динамической сборки, или модулях связи с платформой RabbitMQ приводили к нарушениям работы этих модулей. Если бы я не выбрал изначально данную методологию – затраты на поиск и исправление ошибки с ростом проекта достигли бы катастрофических масштабов и сорвали бы все сроки по его разработке. Так же такой подход является отличной документацией для других членов команды. Если будущие разработчик захочет узнать больше о поведение и пограничных значениях работы какого-то компонента, или даже его метода он может открыть список тестов и узнать всё об их поведении.

Также, как я описывал выше, данный подход, а также использование интегрированной среды разработки Microsoft Visual Studio 2017 позволил мне проводить тестирование во время каждой сборки приложения или части его компонентов, что позволило оперативно реагировать на изменение поведения модулей.

* 1. Модульные тесты

В ходе написание основных модулей программы были написаны тесты для проверки работы их ключевых методов, я хочу выделить тесты для двух наиболее на мой взгляд важный компонентах программы:

1. Для типа СlientFactory был разработан список тестов с данными-заглушками и специальным конфигурированием. Данный тип является очень важным, но в то же время результат его работы очень тяжело поддаётся тестированию. Как я описывал выше на этапе работы этого типа и генерации динамической сборки было продумано логгирование работы этого типа в два независимых файла. Первый файл и содержал форматированное представление нашего динамического класса-декоратора. Во время тестирования были разработаны специальные типы и конфигурации, во время генерации прокси-типов для которых производились сами же типы. Данный метод позволил сравнивать ожидаемый и полученный результат генерирования.
2. Для типа TypeAnalyzer были предусмотрены большие наборы разнообразных типов с заранее известными членами типа и их публичными и не специальными методами, а также простые генерируемые случайным образом типы. В ходе изменения поведения TypeAnalyzer и выполнения тестов можно было легко заметить отличие поведения от задуманного с помощью технологии Reflection (отражения) от компании Microsoft и сравнения результатов анализа этих двух модулей.
   1. Интеграционные тесты

Хоть разрабатываемая библиотека и разрабатывалась с модульным подходом в реализации, всё же не стоит забывать, что данный продукт — это комплексное решение, где связи между модулями очень сложны, а некоторые и вовсе происходят через работу сторонних сетевых компонентов и очередь RabbitMQ.

* 1. Нагрузочные тесты

В главе где я приводил сравнение с аналогами фреймворков и программных платформ я приводил тесты