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

**Библиотека для межсетевого связывания микросервисов для платформы .Net core**

БГУИР ДП 1-40 04 01 00 020 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

**РЕФЕРАТ**

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

дипломный проект / А. А. Дерид. – Минск: БГУИР, 2018, - п.з. – 51 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 производится анализ предметной области, ставится цель разработки, уточняются требования, производится анализ видов и типов лицензирования

Второй раздел содержит анализ существующих аналогов, выбор технологий разработки, программных компонентов с учётом требований к лицензированию и производительности

Третий раздел посвящен проектированию архитектуры приложения. Производится выбор инструментария разработки.

В четвертом разделе описан процесс тестирования продукта, структура тестов, обосновываются требования к производительности.

В пятом 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.

Министерство образования Республики Беларусь

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Учреждение образования БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | |
| Факультет | КС и С | | | Кафедра | | | информатики | | | | | | | | | | |
| Специальность | 1-40 04 01 | | | Специализация | | | | | | 00 | | | | | | | |
| УТВЕРЖДАЮ | | | | | | | | | | | | | | | | | |
|  | | | | | | | | |  | | | | | Н.А.Волорова | | | |
| « | | | | | | | | |  | | » | |  | | | 20 | г. |
| ЗАДАНИЕ | | | | | | | | | | | | | | | | | |
| по дипломному проекту студента | | | | | | | | | | | | | | | | | |
| **Дерида Артема Александровича** | | | | | | | | | | | | | | | | | |
| (фамилия, имя, отчество) | | | | | | | | | | | | | | | | | |
| 1. Тема проекта: | | Библиотека для межсетевого связывания микросервисов для | | | | | | | | | | | | | | | |
| **платформы .Net Core** | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | |
| утверждена приказом по университету от | | | | « | 18 | » | | апреля | | | | 2018 г. | | | № | 654-c | |
| 2 . Срок сдачи студентом законченной работы | | | | | 01 июня 2018 года | | | | | | | | | | | | |
| 3. Исходные данные к проекту | | | Тип операционной системы – Windows 10; | | | | | | | | | | | | | | |
| Язык программирования – C#; | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | |
| Назначение разработки: создание библиотеки для внедрения современных подходов | | | | | | | | | | | | | | | | | |
| разработки ПО для предприятий с существующими закрытыми сетями | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | |
| 4. Содержание пояснительной записки (перечень подлежащих разработке вопросов) | | | | | | | | | | | | | | | | | |
| Введение | | | | | | | | | | | | | | | | | |
| 1 Анализ предметной области | | | | | | | | | | | | | | | | | |
| 2 Анализ существующих аналогов | | | | | | | | | | | | | | | | | |
| 3 Архитектура программного продукта | | | | | | | | | | | | | | | | | |
| 4 Тестирование программного продукта | | | | | | | | | | | | | | | | | |
| 5 Технико-экономическое обоснование | | | | | | | | | | | | | | | | | |
| Заключение | | | | | | | | | | | | | | | | | |
| Список использованных источников | | | | | | | | | | | | | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| 5. Перечень графического материала (с точным указанием наименования) и обозначения | | | |
| вида и типа материала) | |  | |
| Организация системы документооборота. Плакат — формат А1, лист 1. | | | |
| Диаграмма UML программного средства. Плакат - формат А1, лист 1. | | | |
| Диаграмма классов программного средства. Плакат -формат А1, лист 1. | | | |
| Программа обработки данных. Схема программы - формат А1, лист 1. | | | |
| База данных интернет-магазина. Схема данных - формат А1, лист 1. | | | |
| Модуль шифрования. Схема программы - формат А1, лист 1. | | | |
|  | | | |
|  | | | |
| 6. Содержание задания по технико–экономическому обоснованию | | | |
| Расчет сметы затрат, цены, прибыли и экономического эффекта программного продук | | | |
| та у пользователя. | | | |
| Задание выдал |  | | / К.Р. Литвинович / |
|  | | | |
|  | | | |
|  | | | |
| Задание выдал: |  | |  |

**КАЛЕНДАРНЫЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапов дипломного проекта (работы) | Объём этапа в % | Срок выполнения этапа | Примечание |
| Выполнение экономической части | | 10 | 28.03 – 03.04 |  |
| Системное проектирование | | 10 | 04.04 – 09.04 |  |
| Функциональное проектирование | 20 | 10.04 – 24.04 |  |
| Разработка программных модулей | 20 | 15.02–06.03 |  |
| Оформление первых разделов | 10 | 01.05 – 08.05 |  |
| Написание программного модуля | 10 | 09.05 – 14.05 |  |
| Завершение оформления записки | 10 | 15.05 – 21.05 |  |
| Подготовка документов | 10 | 22.05 – 28.05 |  |
| Прохождение нормоконтроля |  | 18.04 – 31.05 | Согласно графику |
| Рабочая комиссиия |  | 04.06–08.06 | Согласно графику |
| Рецензирование дипломного проекта | |  | 11.06 – 14.06 | Согласно графику |
| Защита дипломного проекта | |  | 15.06 – 30.06 | Согласно графику |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Дата выдачи задания | 25 марта 2018 г. | | Руководитель | |  | /М. Г. Рогов/ |
| Задание принял к исполнению | |  | | / А. А. Дерид / | |

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 7](#_Toc515226744)

[1. Анализ предметной области 9](#_Toc515226745)

[1.1 Постановка цели разработки 9](#_Toc515226746)

[1.2 Уточнение требований разработки 15](#_Toc515226747)

[1.2.1 Типы лицензий ПО и их краткое описание 15](#_Toc515226748)

[1.2.2 Вывод после анализа требований, видов лицензирования и типов лицензирования. 18](#_Toc515226749)

[2. Анализ существующих аналогов 20](#_Toc515226750)

[2.1 Аналог архитектуры. 20](#_Toc515226751)

[2.2 Аналог платформы разработки. 20](#_Toc515226752)

[2.3 Выбор среды виртуализации. 21](#_Toc515226753)

[2.4 Выбор платформы передачи сообщений 21](#_Toc515226754)

[2.4.1 Amazon SQS 22](#_Toc515226755)

[2.4.2 RabbitMQ 22](#_Toc515226756)

[2.4.3 Сравнение производительности. 23](#_Toc515226757)

[2.5 Выбор фреймворка для удобной работы с RabbitMQ 27](#_Toc515226758)

[3 Архитектура программного продукта. 28](#_Toc515226759)

[3.1 Инструментарий разработки программного продукта. 28](#_Toc515226760)

[3.1.1 Visual Studio 2017 28](#_Toc515226761)

[3.1.2 ReSharper Ultimate 30](#_Toc515226762)

[3.1.3 Source Tree 30](#_Toc515226763)

[3.1.4 GitLab 31](#_Toc515226764)

[3.2 Обзор модульной системы библиотеки 33](#_Toc515226765)

[3.3 Описание алгоритмов реализации логики приложения 38](#_Toc515226766)

[3.3.1 Обобщенный алгоритм работы программы 38](#_Toc515226767)

[3.3.2 Описание генерации динамических сборок 40](#_Toc515226768)

[4 Тестирование программного продукта 44](#_Toc515226769)

[4.1 Модульные тесты 44](#_Toc515226770)

[4.2 Интеграционные тесты 45](#_Toc515226771)

[4.3. Нагрузочные тесты 46](#_Toc515226772)

[5. Технико-экономическое обоснование 47](#_Toc515226773)

[5.1. Введение и исходные данные 47](#_Toc515226774)

[5.2. Расчет сметы затрат и цены программного продукта 47](#_Toc515226775)

[5.3. Расчет нормативной трудоёмкости 50](#_Toc515226776)

[5.4. Расчет основной заработной платы исполнителей 52](#_Toc515226777)

[5.5. Расчет капитальных затрат 60](#_Toc515226778)

[5.6. Расчёт экономического эффекта 61](#_Toc515226779)

[5.7. Вывод по технико-экономическому обоснованию 64](#_Toc515226780)

[Заключение 65](#_Toc515226781)

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.

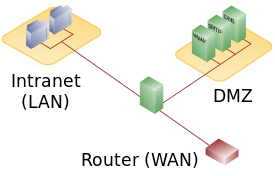
****

Рисунок 1 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#.

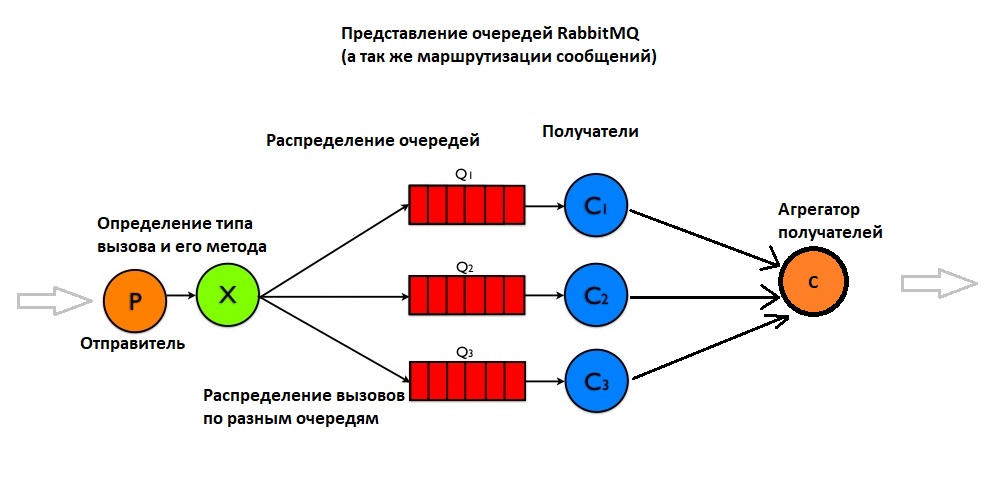


Рисунок 2 Представление очередей RabbitMQ

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

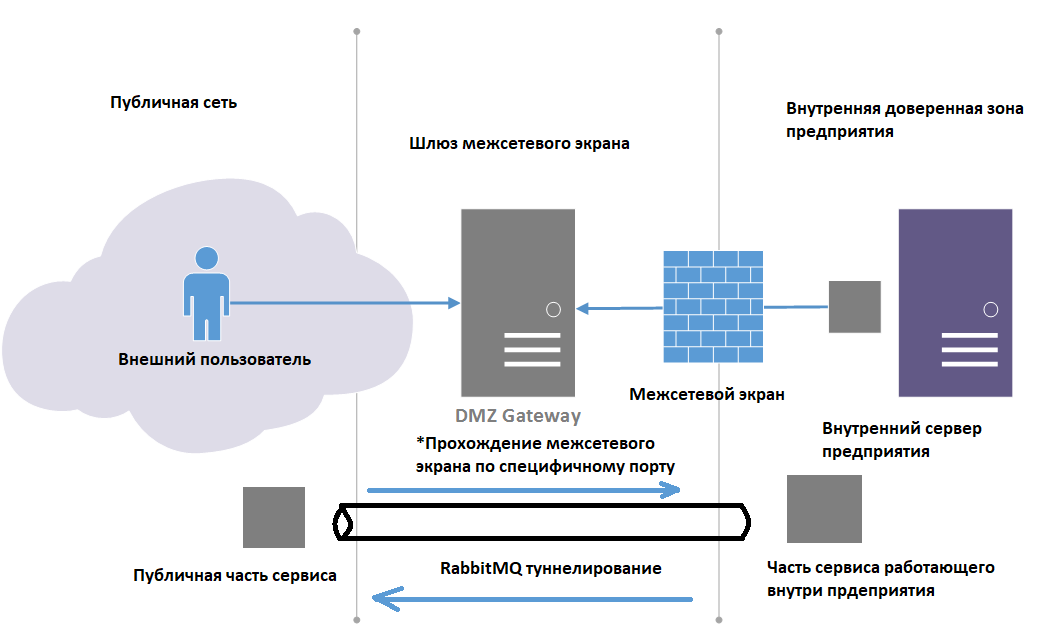


Рисунок 3 Обзорная схема работы системы с использованием библиотеки

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.

1.2 Уточнение требований разработки

Продукт разрабатывается с целью дальнейшего распространения и продажи, что накладывание некоторые ограничение на использования компонент сторонних разработчиков, а именно использование программных компонентов и фреймворков с определённым видом лицензий.

1.2.1 Типы лицензий ПО и их краткое описание

Большинство программных средств распространяются на основании двух видов лицензий: лицензии свободного использования (открытые лицензии и бесплатные виды лицензий) и коммерческие лицензии (лицензии несвободного использования). Так же существует ряд лицензий, которые по своим ограничениям или описаниям можно отнести к группе условно-бесплатных лицензий.

****

Рисунок 4 Типы лицензий и их иерархия

К коммерческим лицензиям относятся лицензии Commercial – главная цель лицензирования ПО такой лицензии это защита интеллектуальной собственности и заработок средств, и частично относятся лицензии из группы условно-бесплатных, такие как: Demo ware, Trial ware. Лицензии из последней группы зачастую так же призваны защищать интеллектуальную собственность, но их правила не так строги, и зачастую предполагают некоторый льготный период бесплатного использования для ознакомительных целей.

К лицензиям со свободным использованием относятся подгруппы бесплатная, с наиболее частыми представителями: Adware -- использования продукта на условиях просмотра рекламного контента компании и/или её партнёров, Freeware – полностью бесплатное ПО и GPL – так же лицензия обеспечивающая полностью бесплатный доступ к ПО, но имеющая ограничение, такое что, любое ПО которые используется на основе компонент с лицензией GLP, так же должно являться бесплатным.

К последней же группе относятся открытые лицензии – это лицензии, поставляемые вместе с ПО, которые имеет открытые источники кода, и которое может модифицироваться любым способом. Наиболее популярной лицензией для последнего вида ПО является OSL – лицензия для открытого ПО, которая продвигается организацией **Open Source Initiative.**

Создание ПО и ограничение его распространения с помощью лицензии это один из немногих путей для разработчика или компании по производству, поддержке или модернизации ПС по защите их интеллектуальной собственности. По каждому отдельному виду программного продукта применяются свои типы лицензирования.



Рисунок 5 Типы лицензирования

Опишем и разберём типы лицензирования и ситуации в которых их применяют:

1. Лицензирование с помощью подписки представляет из себя тип лицензирования регламентирующий использование ПО в ограниченный период времени согласно тарифному плану, предоставленному компании-продавцу. Данный тип предусматривает периодические платежи многоразово в соответствии с тарифным планом.
2. Предустановленное ПО это тип лицензирования, предусматривающий использования ПС которые поставляются вместе с некоторым иным продуктом, и доступен для использования только совместно с продуктом, с которым он эскортируется.
3. Полный комплект – тип лицензирования которые разрешает использование ПО одним или несколькими людьми на одном ЭВМ в один момент времени. Так же ПО можно перенести на другую ЭВМ, но данная операция предусматривает остановку использования данного ПО на первой машине.
4. Корпоративная лицензия – это тип лицензирования, как можно понять из названия, подходящий для крупного бизнеса, которому требуется большее количество лицензий, нежели другим. Данный вид лицензии схож с типом полного комплекта, но поставляется как множество продуктов и лицензий к ним на определенное количество установок.

Стоит учитывать, что помимо предоставленных выше видов, так же существуют менее популярные лицензии или их комбинации, например, корпоративная лицензия в качестве подписки – предоставление подписки на использовании большого количества продуктов и установки на множестве ЭВМ на определённый временной интервал. Такое количество типом лицензирования связано с желанием покупателей-клиентов и бизнес-покупателей оптимизировать свои расходы. Компании же, занимающиеся реализацией ПО подстраиваются под нужды пользователей и рынка.

1.2.2 Вывод после анализа требований, видов лицензирования и типов лицензирования.

Исходя из требований к разрабатываемому ПС и анализу видов и типов лицензирования, мы сможем сформулировать более чёткие цели и выводу по использованию и лицензированию нашего ПС:

В качестве модулей, фреймворков, библиотек и других реализуемых дополнительных компонент нашей системы мы обязаны использовать только компоненты, распространяемые по свободным и открытым лицензиям. Данное ограничение связано с дальнейшим коммерческим распространением продукта, что снимает возможность использования бесплатного ПО.

Возможными способами распространения и типами лицензирования моего ПС предпочтительно будут являться лицензирование по подписки и корпоративное лицензирование, а также их комбинации.

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.

Рисунок 6 Сравнительный показатели скорости работы

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

Рисунок 7 Время выполнения запроса

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 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.

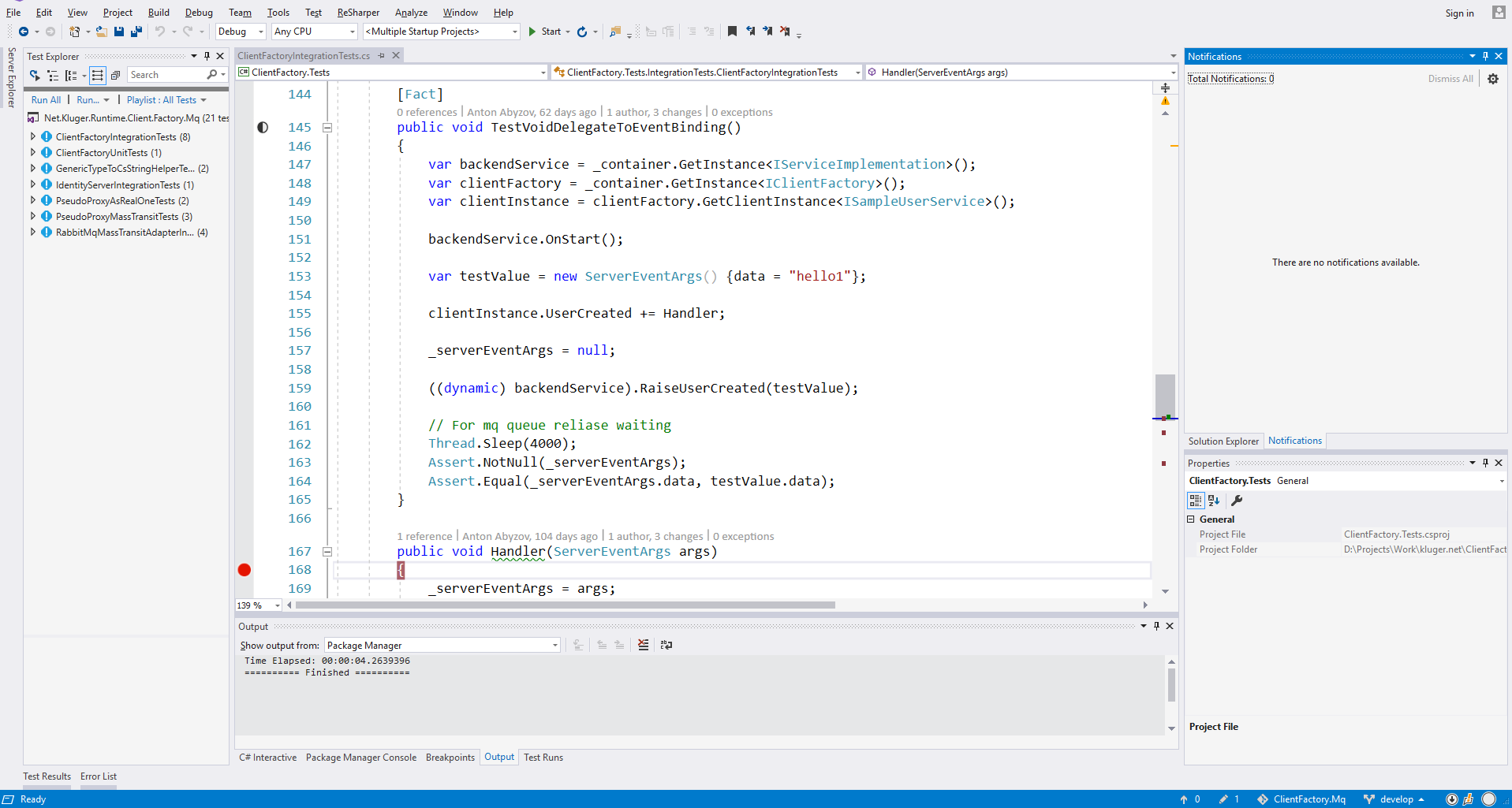


Рисунок 8 Экран рабочей среды Visual Studio

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.

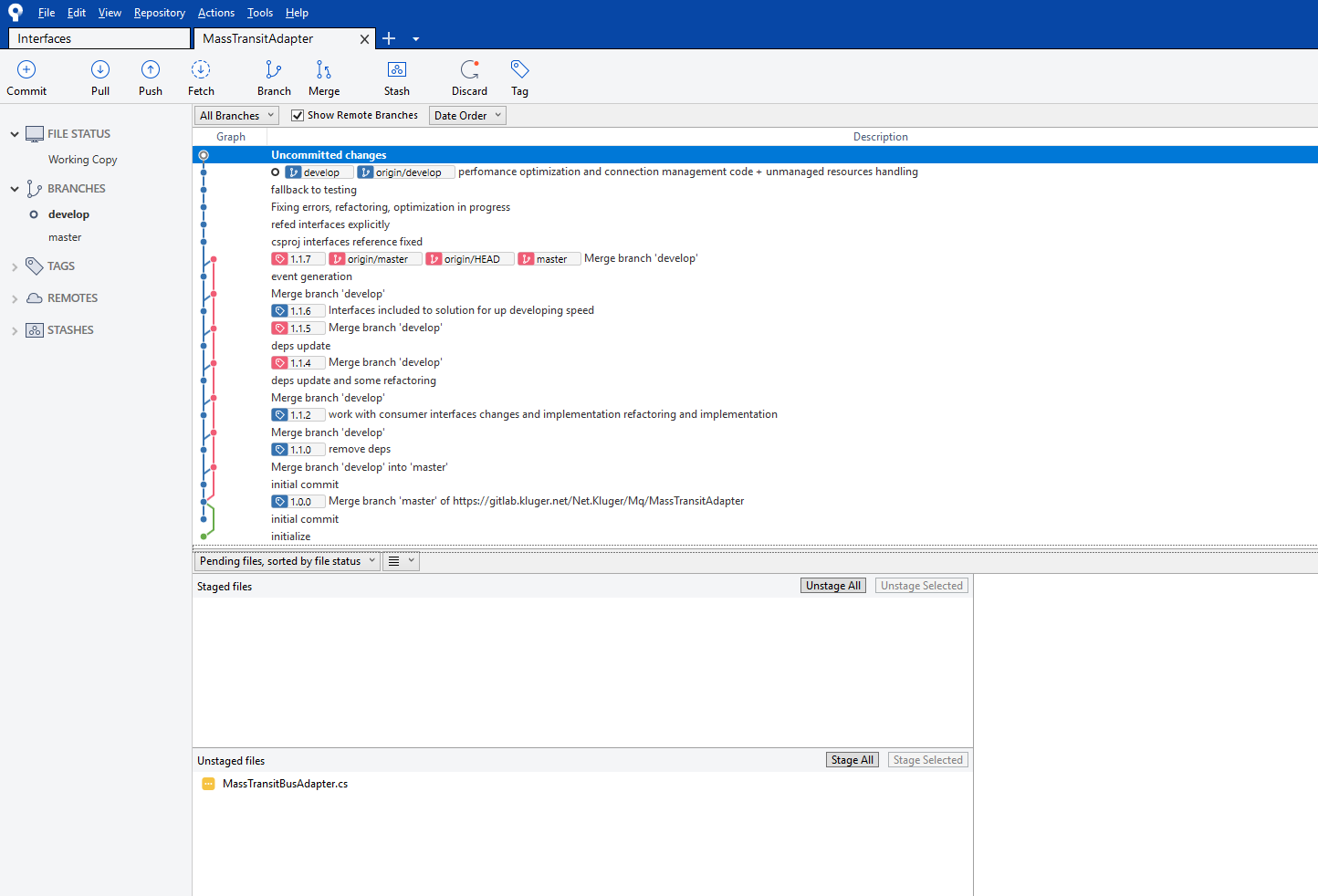


Рисунок 9 Демонстрация работы приложения SourceTree

3.1.4 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.

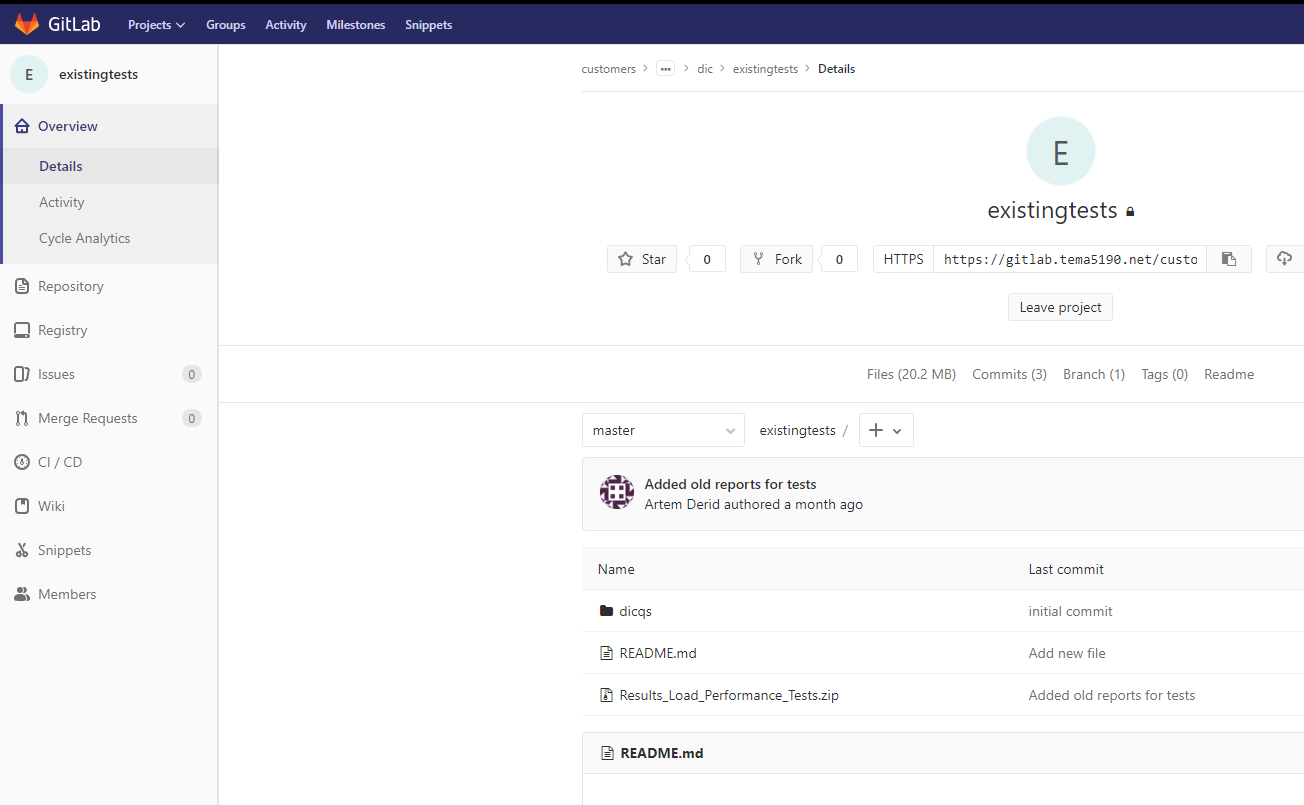


Рисунок 10 Демонстрация работы сайта GitLab

3.2 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. Самые основные интерфейсы, которые входят в этот модуль это IEventMessage, IBusConnector и IServiceImplementation.

Интерфейсы группы IEventMessage призваны для надстройки над сообщениями RabbitMQ и нужны для маршрутизации сообщений. Часть из них имеет интерфейс, который не подразумевает входных аргументов и выходных значений. Они призваны для работы методов без параметров и событий. Часть же реализована с помощью встроенного механизма языка C#, который называется обобщение. Термин обобщение, по существу, означает параметризированный тип. Роль параметризированных типов состоит в том, что они позволяют создавать классы, интерфейсы, структуры, методы и делегаты, в которых содержащиеся и обрабатываемые данные указываются в виде входящего параметра. С помощью обобщений можно, например, создать единый интерфейс, который автоматически становится пригодным для обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется обобщенным, как, например, обобщенный класс или обобщенный интерфейс.

public interface IEventMessage

{

};

public interface IEventMessage<TEventArg> : IEventMessage

{

TEventArg EventMessageData { get; set; }

};

public class EventMessage<TEventArg> : IEventMessage<TEventArg>

{

public TEventArg EventMessageData { get; set; }

};

public class AltEventMessage

{

public object EventMessageData { get; set; }

public Type MessageDataType { get; set; }

public string EventName { get; set; }

}

Интерфейс IBusConnector является очень важным исходным блоком для построения связи с очередью RabbitMQ. Данный интерфейс обозначает все необходимые методы для разных способов отправки сообщения, приёма сообщения, а также описывает интерфейс метода подписки и позволяет узнать подключен ли класс к определённой шине сообщений. Так как шина сообщения является классом, который захватывает неуправляемые ресурсы данный интерфейс так же наследуется от интерфейса IDisposable. Этот специальный интерфейс позволяет во время уничтожения класса сборщиком мусора вызвать метод Dispose, в реализации которого мы можем программно попытаться освободить все неуправляемые ресурсы, в данном случае сокет соединения.

public interface IBusConnector : IDisposable

{

void Publish<T>(T message) where T : class;

void Publish(Type type, object message);

Task PublishAsync<T>(T message) where T : class;

Task PublishAsync(Type type, object message);

IDisposable Subscribe<T>(Action<T> onMessage, int? expiresAfterSeconds = null) where T : class;

IDisposable SubscribeAsync<T>(Func<T, Task> onMessage, int? expiresAfterSeconds = null) where T : class;

TResponse Request<TRequest, TResponse>(TRequest request)

where TRequest : Request

where TResponse : Response;

TResponse Request<TRequest, TResponse>(string topic, TRequest request)

where TRequest : Request

where TResponse : Response;

Task<TResponse> RequestAsync<TRequest, TResponse>(string topic, TRequest request)

where TRequest : Request

where TResponse : Response;

IDisposable Respond<TRequest, TResponse>(Func<TRequest, TResponse> responder)

where TRequest : Request

where TResponse : Response;

IDisposable RespondAsync<TRequest, TResponse>(Func<TRequest, Task<TResponse>> responder)

where TRequest : Request

where TResponse : Response;

IDisposable Respond<TRequest, TResponse>(string topic, Func<TRequest, TResponse> responder)

where TRequest : Request

where TResponse : Response;

IDisposable RespondAsync<TRequest, TResponse>(string topic, Func<TRequest, Task<TResponse>> responder)

where TRequest : Request

where TResponse : Response;

void Send<T>(string queue, T message) where T : class;

Task SendAsync<T>(string queue, T message) where T : class;

IDisposable Receive<T>(string queue, Action<T> onMessage) where T : class;

IDisposable ReceiveAsync<T>(string queue, Func<T, Task> onMessage) where T : class;

bool IsConnected { get; }

}

Интерфейс IServiceImplementation является также одним из важнейших, хоть и достаточно простым в описании, для работы приложения. Этот интерфейс необходимо реализовывать у класса-контейнера или класса-декоратора наших микросервисов.

public interface IServiceImplementation : IDisposable

{

Task OnStart();

Task OnSuspend();

}

1. MassTransit Adapter – модуль который содержит общую логику передачи сообщений с помощью фреймворка MassTransit и платформы RabbitMQ. Данный модуль реализует интерфейс IBusConnector. 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.
2. 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. Класс ProxyGenerator является разделяемым классом с описанием логики в нескольких физических файлах. Данное разбиение позволяет использовать только те части реализации класса, которые необходмы в конкретной ситуации. Так же класс реализует интерфейс IProxyGenerator, который позволяет поддерживать единый стиль написания и использовать систему внедрения зависимостей:

public interface IProxyGenerator

{

DynamicAssemblyContainer<T> GenerateClient<T>() where T : class;

}

Так же важнейшей частью модуля является интерфейс ITypeAnalyzer и его реализация

public interface ITypeAnalyzer

{

TypeStructure AnalyzeType(Type type);

}

1. 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.

public class ClientFactory: IClientFactory

{

private readonly ITypeAnalyzer \_typeAnalyzer;

private readonly IProxyGenerator \_proxyGenerator;

public ClientFactory(ITypeAnalyzer typeAnalyzer, IProxyGenerator proxyGenerator)

{

\_proxyGenerator = proxyGenerator;

\_typeAnalyzer = typeAnalyzer;

}

public T GetClientInstance<T>() where T : class

{

var generator = \_proxyGenerator;

var result = generator.GenerateClient<T>();

return result.Proxy;

}

}

1. 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.

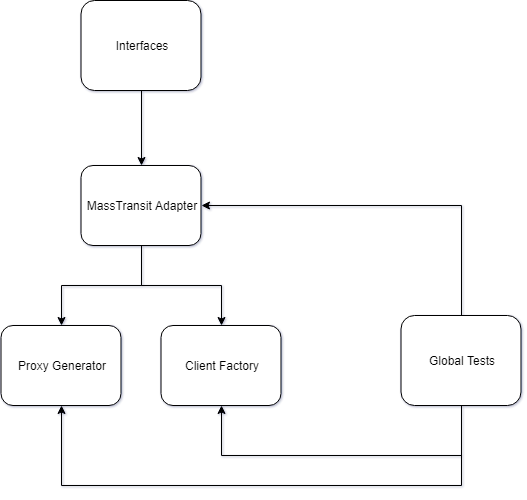


Рисунок 11 Схема зависимостей модулей

3.3 Описание алгоритмов реализации логики приложения

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 проектирования Одиночка. Данный паттерн проектирования 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:

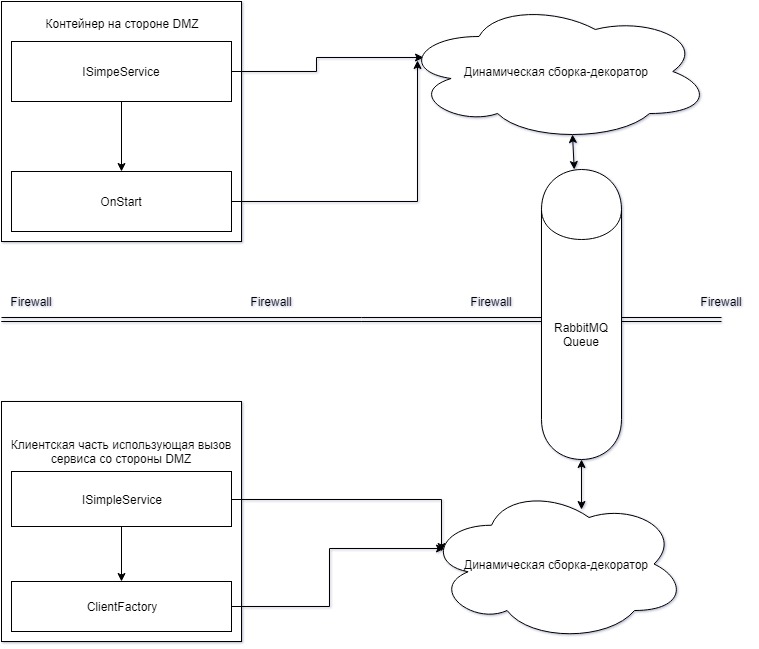


Рисунок 12 Схема связи внутренней и внешней части сервиса

3.3.2 Описание генерации динамических сборок

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:

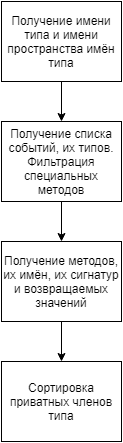


Рисунок 13 Схема работы анализатора типов

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.

4 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.

4.1 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.

4.2 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.

Для проверки работы системы созданы модули и их заглушки начиная с самых простых проверок взаимодействия и самым большим количеством mock сервисов и данных, и заканчивая полной проверкой работы модуля и эмуляции как внутренней, так и внешней части сервиса внутри выполнения одного тестового сценария. Так же отдельно стоит выделить проверки доставки и получения сообщений с помощью ручной реализации отправки в определённую заданную очередь сообщений с помощью клиентской библиотеки для платформы .Net RabbitMQ.

Кроме этого производится одновременный запуск одного теста параллельно, чтобы проверить работу системы маршрутизации и убедится в корректности получения сообщения на внешней стороне клиента.

4.3. Нагрузочные тесты

В главе где я приводил сравнение с аналогами фреймворков и программных платформ я приводил результаты тестирования других платформ. Стоит упомянуть что данные тесты были адаптированы и используются в процессе ручного тестирования при каждом крупном изменении версии одного из пакетов. Во время цикла тестирования обычной работы библиотеки запускаются нагрузочные тесты чтобы собрать статистику и проверить работоспособность приложения при большой нагрузке на сервер, который занимается обработкой очереди RabbitMQ, проверке корректной работы маршрутизации, и производятся замеры производительности. Если замеры производительности оказались ниже допустимого уровня инициируется процесс оптимизации, так как система в будущем призвана работать на системах с большими нагрузками и имеет повышенные требования к производительности.

5. Технико-экономическое обоснование

5.1. Введение и исходные данные

Технологические подходы к созданию программных средств (ПС) постоянно обновляются и подстраиваются под современные нужды и тенденции рынка. Но не всегда то, что широко используется для решения множества задач подходит под узкоспециализированные задачи крупных предприятий с уже наработанной программной базой, техническими средствами, установленной сетевой коммуникацией. Модернизация всей инфраструктуры таких предприятий зачастую чрезмерна или требует слишком больших денежных затрат.

Задачей моего проекта является создания ПС – библиотеки для межсетевого связывания сервисов, которая поможет более эффективно внедрять современные подходы к разработке программного обеспечения, такие как микросервисная архитектура в уже существующие системы.

Библиотека может решить проблемы создания микросервисной архитектуры для сетей с закрытым доступом в рамках корпоративной сети предприятия без дополнительной закупки оборудования, содержания штаба разработчиков и технических специалистов и без больших изменений существующей кодовой базы компании. Но разрабатываемое ПС должно быть не только новым в техническом плане, но и целесообразным с точки зрения экономики.

Целью технико-экономического обоснования ПС является определение экономической эффективности создания данного продукта и дальнейшего его применения.

Для этого рассчитаем смету затрат, прибыль и цену ПС.

5.2. Расчет сметы затрат и цены программного продукта

В таблице 1 приведены исходные данные для расчета сметы затрат и цены программного продукта.

1. − Исходные данные для расчета сметы затрат и цены программного продукта.

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование показателей | Буквенные обозначения | Единицы измерения | Количество |
| Коэффициент новизны |  | единиц | 0,9 |
| Группа сложности |  | единиц | 1 |
| Дополнительный коэффициент сложности |  | единиц | 0,12 |
| Поправочный коэффициент, учитывающий использование типовых программ |  | единиц | 0,8 |
| Установленная плановая продолжительность разработки |  | лет | 1,25 |
| Продолжительность рабочего дня |  | ч | 8 |
| Тарифная ставка 1-го разряда |  | руб | 155 |
| Коэффициент премирования |  | единиц | 1,2 |
| Норматив дополнительной заработной платы исполнителей |  | % | 20 |

Продолжение таблицы 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Отчисления в фонд социальной защиты населения |  | % | 34 |
| Отчисления в Белгосстрах |  | % | 0,6 |
| Расходы на научные командировки |  | % | 30 |
| Прочие прямые расходы |  | % | 20 |

Программный комплекс относится к 1-й группе сложности. Категория новизны продукта – «Б». Для оценки экономической эффективности разработанного программного средства производится расчёт прибыли от продажи одной системы (программы). Расчёты выполнены на основе методического пособия [1].

Объем программного средства определяется путем подбора аналогов на основании классификации типов программного средства, каталога функций, которые постоянно обновляются и утверждаются в установленном порядке. На основании информации и функциях разрабатываемого программного средства по каталогу функций определяется объем функций. Объем программного средства определяется на основе нормативных данных, приведенных в таблице 2.

1. − Характеристика функций и их объем

|  |  |  |
| --- | --- | --- |
| Номер функции | Наименование (содержание) функций | Объем функций |
| 104 | Преобразование операторов входного языка и команды другого языка | 4700 |
| 106 | Преобразование входного языка в машинные команды (транслятор, препроцессор, макрогенератор) | 8010 |
| 107 | Синтаксический и семантический анализ входного языка и генерация кодов команд | 1640 |
| 208 | Организация соединения со сторонними сервисами | 11880 |
| 210 | Загрузка, обработка и консолидация данных | 1370 |
| 507 | Обеспечение интерфейса между компонентами | 2400 |
| Итого: |  | 30000 |

Общий объём программного продукта ( определяется по следующей формуле (4.1):

|  |  |
| --- | --- |
|  | (4.1) |

|  |  |
| --- | --- |
| где | − объем отдельной функции ПО;  − общее число функций. |

= 30000 (строк исходного кода).

5.3. Расчет нормативной трудоёмкости

Исходя из режима работы в реальном времени, а так же использования множества машин применяется коэффициент к объему ПО, который определяется по формуле (4.2):

|  |  |
| --- | --- |
|  | (4.2) |

|  |  |
| --- | --- |
| где | − коэффициент, соответствующий степени повышения сложности  за счет конкретной характеристики; |
|  | − количество учитываемых характеристик. |

С учетом дополнительного коэффициента сложности рассчитывается общая трудоемкость ПС по формуле (4.3):

|  |  |
| --- | --- |
| , | (4.3) |

|  |  |
| --- | --- |
| где | – общая трудоемкость; |
|  | – нормативная трудоемкость ПС; |
|  | – дополнительный коэффициент сложности ПС; |
|  | – поправочный коэффициент, учитывающий степень использования при разработке стандартных модулей; |
|  | – коэффициент, учитывающий степень новизны ПС. |

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

На основе уточненной трудоемкости разработки ПС и установленного периода разработки, общая плановая численность разработчиком рассчитывается по формуле (4.4):

|  |  |
| --- | --- |
|  | (4.4) |

|  |  |
| --- | --- |
| где | − эффективный фонд времени работы одного работника в течение  года (дн.);  − общая трудоемкость разработки проекта (чел./дн.);  − срок разработки проекта (лет). |

Эффективный фонд времени работы одного работника () рассчитывается по формуле (4.5):

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | где | − количество дней в году;  количество праздничных дней в году;  − количество выходных дней в году;  − количество дней отпуска. | | (4.5) |

Срок разработки установлен 15 месяцев ( = 1,25 г.):

Рассчитаем трудоемкость ПС и численность исполнителей по стадиям по таблице 3.

1. − Расчет утонченной трудоемкости ПС и численности исполнителей по стадиям

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Показатели | Стадии | | | | | Итого |
| ТЗ | ЭП | ТП | РП | ВН |
| Коэффициенты удельных весов трудоемкости стадии разработки ПО (d) | 0,09 | 0,07 | 0,07 | 0,61 | 0,16 | 1,0 |
| Распределение нормативной трудоёмкости ПО( по стадиям, чел./дн. | 69 | 55 | 55 | 472 | 124 |  |
| Коэффициент сложности ПО (Kc) | 1,12 | 1,12 | 1,12 | 1,12 | 1,12 |  |
| Коэффициент, учитывающий использование стандартных модулей |  |  |  | 0,8 |  |  |
| Коэффициент, учитывающий новизну ПО (Кн) | 0,9 | 0,9 | 0,9 | 0,9 | 0,9 |  |
| Численность исполнителей, чел. (Чi) | 2 | 2 | 2 | 2 | 2 | 2 |
| Сроки разработки, лет |  |  |  |  |  | 1,25 |

5.4. Расчет основной заработной платы исполнителей

Реализацией проекта занимались 2 человека. В соответствии с численностью и выполняемым функциями устанавливается штатное расписание группы специалистов-разработчиков.

Расчет основной заработной платы осуществляется в следующей последовательности. Определим месячные (Тм) и часовые (Тч) тарифные ставки начальника отдела (тарифный разряд – 14; тарифный коэффициент – 5,31;), инженера-программиста 1-й категории (тарифный разряд – 12; тарифный коэффициент – 4,39). Месячный тарифный оклад (Тмо) определяется путем умножения действующей месячной тарифной ставки 1-го разряда (Тм1) на тарифный коэффициент (Ткi), соответствующий установленному тарифному разряду специалиста (формула (4.6)):

|  |  |
| --- | --- |
| . | (4.6) |

Часовая тарифная ставка рассчитывается путем деления месячной тарифной ставки на установленный при сорокачасовой рабочей неделе в восьмичасовом рабочем дне фонд рабочего времени – 176 часов – приведена в формуле (4.7)

|  |  |  |  |
| --- | --- | --- | --- |
| |  |  | | --- | --- | | где | – часовая тарифная ставка (руб);  – месячная тарифная ставка (руб). | | (4.7) |

Принимаем тарифную ставку 1-го разряда равной 155 руб. Месячная и часовая тарифные ставки начальника отдела:

Месячная и часовая тарифные ставки инженера-программиста 1-й категории равны соответственно:

Расчет месячных и почасовых тарифных ставок сведен в таблицу 4.

1. − Штатное расписание группы разработчиков

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Должность | Количество ставок | Тариф-ный разряд | Тарифный коэффициент | Месячная тарифная ставка (руб) | Часовая тарифная ставка  (руб) |
| Начальник отдела (ведущий инженер программист) | 1,00 | 14 | 5,31 | 823,05 | 4,68 |
| Инженер-программист 1-й категории | 1,00 | 12 | 4,39 | 680,45 | 3,87 |
| Итого: |  |  |  | 1503,5 | 8,55 |

Основная заработная плата исполнителей на конкретное ПО рассчитывается по формуле (4.8):

|  |  |
| --- | --- |
|  | (4.8) |

|  |  |
| --- | --- |
| где | − количество исполнителей, занятых разработкой конкретного ПО;  – часовая тарифная ставка i-го исполнителя;  – плановый фонд рабочего времени i-го исполнителя;  – количество часов работы в день;  – коэффициент премирования; |

.

3,87 \* 8 \* 312 \* 1,2 + 4,68 \* 8 \* 312 \* 1.2.

Дополнительная заработная плата исполнителя или исполнителей проекта определяется по формуле (4.9):

|  |  |
| --- | --- |
|  | (4.9) |

Отчисления в фонд социальной защиты населения и на обязательное страхование (ЗС) определяются в соответствии с действующими законодательными актами по формуле (4.10):

|  |  |
| --- | --- |
|  | (4.10) |

|  |  |
| --- | --- |
| где | НСЗ – норматив отчислений в фонд социальной защиты населения и на обязательное страхование (34 +0,6%). |

Расходы по статье «Машинное время» (РМ) включают оплату машинного времени, необходимого для разработки и отладки программного средства. Они определяются в машино-часах по нормативам на 100 строк исходного кода машинного времени в зависимости от характера решаемых задач и типа ПО, и определяются по формуле (4.11):

|  |  |
| --- | --- |
|  | (4.11) |

Затраты по статье «Накладные расходы» (РН), связанные с необходимостью содержания аппарата управления, вспомогательных хозяйств и опытных (экспериментальных) производств, а также с расходами на общехозяйственные нужды (РН), определяются по формуле (4.12):

|  |  |
| --- | --- |
|  | (4.12) |

Полная себестоимость высчитывается по формуле (4.13):

|  |  |
| --- | --- |
|  | (4.13) |

Прогнозируемая прибыль Ппс рассчитывается по формуле (4.14):

|  |  |
| --- | --- |
|  | (4.14) |

Прогнозируемая отпускная цена вычисляется по формуле (4.15):

|  |  |
| --- | --- |
|  | (4.15) |

Налог на добавленную стоимость (НДСi) считается по формуле (4.16):

|  |  |
| --- | --- |
|  | (4.16) |

|  |  |
| --- | --- |
| где | Hдc − норматив НДС (%). |

Для прогнозируемой отпускной цены (Цo) существует формула (4.17):

|  |  |
| --- | --- |
|  | (4.17) |

Кроме того, организация-разработчик осуществляет затраты на сопровождение Рс, которые определяются по нормативу (Нс), где Нс – норматив расходов на сопровождение и адаптацию (20%) – формула (4.18).

|  |  |
| --- | --- |
|  | (4.18) |

В следующей таблице 5 приведены исходные данные для расчета экономического эффекта.

Таблица 5 − Исходные данные для расчета экономического эффекта.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Наименование показателей | Обозначения | Единицы измерения | Значение | | Наименование источника информации |
| в базовом варианте | в новом варианте |
| Капитальные вложения, включая затраты пользователя на приобретение | Kпp | руб. |  |  | Договор заказчика с разработчи-ком |
| Затраты на сопровождение ПО | Кс | руб. |  |  | Договор заказчика с разработчи-ком |
| Время простоя сервиса, обусловленное ПО, в день | П1,П2 | мин | 200 | 20 | Расчетные данные пользователя и паспорт ПО |
| Стоимость одного часа простоя | Сп | руб. | 100 | 100 | Расчетные данные пользователя и паспорт ПО |

Продолжение таблицы 5.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Среднемесячная ЗП одного программиста | Зсм | руб. | 680,45 | 680,45 | Расчетные  данные пользователя |
| Коэффициент начислений на зарплату | Kнз |  | 1,2 | 1,2 | Рассчитывает-ся по данным пользователя |
| Среднемесячное количество рабочих дней | Др | день | 22 | 22 | Принято для расчета |
| Количество типовых задач, решаемых за год | Зт1 ,Зт2 | задача | 1500 | 1500 | План пользователя |
| Объем выполняемых работ | А1, А2 | задача | 1000 | 1000 | План пользователя |
| Средняя трудоемкость работ на задачу | Tc1,  Тс2 | Челове-ко-часов | 15 | 1.3 | Рассчитывает-ся по данным пользователя |
| Количество часов работы в день | Tч | ч | 8 | 8 | Принято для расчета |
| Ставка налога на прибыль | Hп | % |  | 18 | Выдержка из законов РБ |

5.5. Расчет капитальных затрат

Общие капитальные вложение (Kо) заказчика (потребителя), связанный с приобретением, внедрением и использованием программного средства, включают в себя затраты на приобретение программного средства, освоение программного средства, а также затраты на доукомплектацию техническими средствами в связи с внедрением нового программного средства, а также затраты на пополнение оборотных средств, в связи с использованием нового программного средства.

Таблица 6 – Расчёт капитальных затрат.

|  |  |  |
| --- | --- | --- |
| Наименование | Методика расчета | Значение (руб.) |
| Затраты пользователя на приобретение ПС по отпускной цене разработчика | Кпр |  |
| Затраты на освоение ПС | Кос | 0 |
| Затраты на сопровождение ПС | Кс |  |
| Затраты на пополнение оборотных средств в связи с использованием нового ПС | Коб | 0 |
| Общие капитальные вложения | Ко = Кпр + Кос + Кс + Коб | 111.136,42 |
| Экономия затрат на заработную плату в расчёте на 1 задачу | Сзе = Зсм \* (Tc1 – Tc2)/(Тч \* Др) | 52,96 |
| Экономия на заработную плату при использовании нового ПС | Сз = Сзе \* А2 | 52.966,84 |
| Экономия с учётом начисления на заработную плату | Сн = Сз \* Кнз | 63.560,21 |
| Экономия за счёт сокращения простоя сервиса | Сс = (П1 – П2) \* Сп \* Дрг / 60 |  |

Продолжение таблицы 6.

|  |  |  |
| --- | --- | --- |
| Общая годовая экономия текущих затрат связанных с использованием нового ПС | Со = Сн + Сс | 70.160,21 |

Внедрение нового ПС позволит сэкономить на текущих затратах 70.160,21 рублей.

5.6. Расчёт экономического эффекта

Для пользователя в качестве экономического эффекта выступает лишь чистая прибыль – дополнительная прибыли, которая остаётся в его распоряжении после внедрения нового программного средства, которая определяется по формуле (4.24):

|  |  |
| --- | --- |
|  | (4.24) |

В процессе использования нового ПО чистая прибыль в конечном итоге возмещает капитальные затраты, однако полученные при этом суммы результатов (прибыли) и затрат (капитальных вложений) по годам приводят к единому времени − расчетному году (за расчетный год принят 2018-й год) путем умножения результатов и затрат за каждый год на коэффициент дисконтирования α. В данном примере используются коэффициенты:

2018 г. – 1, 2019-й – 0,8696, 2020-й – 0,7561, 2021 г. – 0,6575.

Все рассчитанные данные экономического эффекта сводятся в таблицу 7.

Таблица 7 – Расчет экономического эффекта от использования нового программного средства.

| Показатели | Едини-цы  измере-ния | Годы | | | |
| --- | --- | --- | --- | --- | --- |
| 2018 | 2019 | 2020 | 2021 |
| Прирост прибыли за счет экономии затрат (Пч) | руб. | - |  |  |  |
| То же с учетом фактора времени | руб. | - | 50.029,28 | 43.499,47 | 37.826,87 |
| Приобретение ПО (Кпр) | руб. |  | - | - | - |
| Сопровожде-ние (Кс) | руб. |  | - | - | - |
| Всего затрат | руб. | 111.136,42 | - | - | - |
| То же с учетом фактора времени | руб. | 111.136,42 | - | - | - |
| Экономический эффект |  |  |  |  |  |
| Превышение результата над затратами | руб. | -111.136,42 | 50.029,28 | 43.499,47 | 37.826,87 |

Продолжение таблицы 7.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| То же с нарастающим итогом | руб. | -111.136,42 | -61.107,14 | -17.607,67 | 20.219,20 |
| Коэффициент приведения | единицы | 1 | 0,8696 | 0,7561 | 0,6575 |

5.7. Вывод по технико-экономическому обоснованию

Исходя из всех произведённых расчётов, и конечного результата из таблицы расчета экономического эффекта можно доказать, что использование заказчиком (потребителем) библиотеки для межсетевого связывания микросервисов является экономически целесообразным и выгодным. Данный подход позволяет существенно сократить расходы в нашей предметной области и приведёт к автоматизации процессов, требующих меньшее количество ресурсов и времени. Положительный экономический эффект заключается в уменьшении трудоёмкости работ на задачу, в существенном снижении времени простоя. Продукт является экономически выгодным исходя из того факта, что его окупаемость достигается по истечении трёх лет использования. В конце этого периода чистая прибыль составит 20.219,20 рублей.

Заключение

В ходе выполнения проекта были изучены такие технологии как RabbitMQ, MassTransit, Docker, а также дополнены знания по поводу сетевого взаимодействия, устройства крупных сетей, настройки межсетевых экранов.

Подводя итоги можно сказать что данная библиотека, разработанная для внедрения новых подходов к разработке ПО, таких как контейнеризация и использования микросервисов и модульности является готовым продуктом для доставки конечным потребителям и внедрения в реальных системах. Библиотека захватывает участки рынка в данном технологическом стеке, и у неё нет аналогов в своём классе. Современные инструменты, библиотеки, фреймворки и версия языка C# являются хорошим выбором и обеспечат актуальность продукта на долгое время. Сильная нагрузка на тестирование функциональности, нагрузочное тестирование и тестирование каждого отдельного модуля позволяет исключить возможность сбоя готового продукта, а также позволят вести поддержку библиотеки при дальнейших модификациях, расширении функционала и доработках. Продукт является совершенным технически, и, создавая его, я применил огромное количество знаний по разработке ПО, тестированию ПО, процессах оптимизации, а также правилах и концепциях по разработке качественного ПО, полученных в ходе обучения.

В будущем приложение будет развиваться посредством увеличения и улучшения качества модульности системы для более удобной сборки и поставки конечному потребителю. Также будет вестись разработка в сторону унификации системы доставки сообщений, для возможности использования других систем, если таковые уже будут в наличии у конечного потребителя. В следующих системных изменениях библиотеки будут использоваться более специфичные нагрузочные тесты, для нахождения слабых мест в производительности и их устранения.

С экономической точки зрения проект является выгодным решением, позволяющим уменьшить время простоя, расширять возможности существующего ПО, работающего на основе существующих на предприятиях сетей, без необходимости модернизации их, и без набора дополнительного персонала для их обслуживания и доработки.

Для запуска продукта необходим сервер обработки сообщений RabbitMQ, готовая закрытая сетевая структура с внутренними сервисами для расширений их функционала с помощью моего продукта.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Amazon SQS vs RabbitMQ – Блог компании EPAM на ресурсе Habr [Электронный ресурс]. – Электронные данные. – Режим доступа: https://habr.com/company/epam\_systems/blog/161787/
2. RabbitMQ API Documentation – Документация платформы RabbitMQ [Электронный ресурс]. – Электронные данные. – Режим доступа: <http://rabbitmq.github.io/rabbitmq-dotnet-client/api/RabbitMQ.Client.html>
3. RabbitMQ API Documentation – Документация платформы RabbitMQ [Электронный ресурс]. – Электронные данные. – Режим доступа: <http://rabbitmq.github.io/rabbitmq-dotnet-client/api/RabbitMQ.Client.html>