ВВЕДЕНИЕ

Уже на данном этапе развития рынка IT услуг конечный пользователь программного продукта тяжело переносит сбои в работе программного обеспечения. Современный рынок информационных технологий наполнен разнообразием услуг, у каждого сервиса есть аналоги со своими плюсами и недостатками, из-за этого конечным клиентом не прощается регулярные сбои работы приложения или ответа сервера. Поэтому для бизнеса на данный момент времени выдвигаются на передний план такие показатели, как надежность и качество.

Одной из частей жизненного цикла любого приложения является разработка его архитектуры. От того как будет спроектирована его архитектура зависит безопасность, производительность, скорость добавления нового функционала а также возможность использования готовых функций текущего проекта в последующих. Архитектура - “фундамент” приложения, от которого зависит вся дальнейшая работа с ним, поэтому очень важно уделить достаточное внимание проработке всех нюансов на начальном этапе, так как рефакторинг и изменения архитектуры на последующих жизненных циклах будут стоить большого количества часов.

Также немаловажным является человеческий фактор, не стоит забывать, что после разработки архитектуры, со временем приложение будет дополнятся новым функционалом, то есть с написанным, готовым приложением будут работать люди, важно, чтобы структура проекта была понятна и не имела слишком высокого порога входа, дабы не увеличивать время разработки.

На данный момент есть несколько подходов к проектированию архитектур, самые популярные из них: “монолит” и микросервисы. При

этом микросервисная архитектура появилась совсем недавно (в середине 2010-х годов), что подтверждает актуальность данной проблемы, так как до сих пор с изменением рынка изменяются и способы проектирования информационных систем.

Целью данной работы является разработка микросервиса системы информеров для компании “Управление информационными проектами” как часть микросервисной архитектуры.

Для достижение цели поставлены следующие задачи:

1. Согласовать с компанией требования к системе информеров.
2. Разработать архитектуру приложения и выбрать технологии акцентируя внимание на расширяемость и конфигурируемость, слабую связность.
3. Реализовать микросервис.

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Разбор существующих архитектур

На данный момент существует два основных способа проектирования архитектуры приложения: “монолит” и микросервисный подход. Основой первого способа проектирования является идея создания приложения, в котором все компоненты будут неотделимо включены в него, отсюда и название способа. Во время работы система может взаимодействовать с другими службами или хранилищами данных, однако основа его поведения реализуется в собственном процессе, а все приложение обычно развертывается как один элемент. Для горизонтального масштабирования такое приложение обычно целиком дублируется на нескольких серверах или виртуальных машинах.

Из плюсов такого подхода к проектированию можно отметить:

1. Быструю скорость добавления нового функционала на ранних этапах развития приложения(т.к. в монолитном проекте проще отсмотреть связность кодовой базы и убедиться, что новая, только что добавленная функциональность работает корректно);
2. Быструю скорость выполнения операции(т.к. в монолитном приложении вызов определенной функции чаще всего не выходит за рамки этого же приложения, т.е. нет необходимости обращаться к сторонним приложениям через сеть, т.к. большая часть функциональности находится в самом приложении, время выполнения операции сокращается);

Из недостатков такого подхода к проектированию можно отметить:

1. Сложность масштабирования приложения из-за сильной связности и запутанности бизнес-логики, которая появляется в процессе эволюции приложения.

2. Сложность понимания бизнес логики из-за появления множества связей между компонентами.

Монолитная архитектура хорошо подходит для небольших проектов, так как стоимость разработки приложения с такой архитектурой гораздо ниже, к тому же на разработку такого приложения уйдет куда меньше времени в сравнении с приложением с микросервисной архитектурой.

Отличие микросервисного подхода от “монолитного” заключается в том, что в отличии от “монолитной” архитектуры, где каждый компонент являлся неразрывной частью самого приложения в микросервисной архитектуре всё приложение разбивается на микросервисы - отдельные компоненты программы, которые выполняют конкретный набор функций, связь между которыми осуществляется через определённый интерфейс посредством легковесных протоколов, например, HTTP. Ключевая идея микросервиса - независимость от окружающей среды, слабая с ней связность и высокий уровень повторного использования.

Полностью независимые микросервисные компоненты обеспечивают абсолютно автономное владение, что в результате дает следующие преимущества:

1. разработчики микросервиса в состоянии полностью понимать его кодовую базу. Они могут создавать, разворачивать и тестировать его независимо от других компонентов, используя намного более короткие итерационные циклы. Микросервис — всего лишь один из множества компонентов сети, поэтому для его разработки можно использовать наиболее подходящий для требуемой функциональности язык программирования или среду, а также оптимальный механизм

устойчивости. Данный подход позволяет существенно снизить объем кода и невероятно упрощает его поддержку. Это позволяет отдельным коллективам разработчиков оперативно применять новые технологии или модификации имеющихся технологий, не дожидаясь, пока другие фрагменты приложения будут готовы к такому переходу. Некоторые определения понятия подчеркивают, что микросервис должен быть настолько простым, чтобы при необходимости его можно было полностью переписать за одну итерацию.

2. команда разработчиков микросервиса может масштабировать компонент во время выполнения независимо от других микросервисов, тем самым обеспечивая эффективное использование ресурсов и оперативное реагирование на изменения нагрузки. Теоретически нагрузку компонента можно перенести на самую подходящую для конкретной задачи платформу. Благодаря такому независимому переносу можно получить преимущество за счет правильного размещения в сети. Микросервисы с хорошим исходным кодом предоставляют огромные возможности для масштабирования по требованию, что успешно подтверждается практикой. Кроме того, такие микросервисы способны реализовать преимущества эластичных облачных сред, обеспечивающих экономичный доступ к огромным ресурсам.

3. отдельные пространства выполнения немедленно обеспечивают устойчивость, которая не зависит от сбоев других компонентов. При правильном разделении на компоненты (в частности, отказ от синхронных зависимостей и использование шаблонов проектирования Circuit Breaker) каждый микросервис можно разрабатывать под определенные требования доступности, не затрагивая приложение целиком. Различные технологии, например контейнеры и упрощенные

среды выполнения, позволяют быстро отключать микросервисы в случае сбоев, без ущерба для других несвязанных функций. В то же время в реализациях микросервисов не применяется модель сохранения текущего состояния, что позволяет мгновенно перераспределить нагрузку и практически незамедлительно инициализировать новые среды выполнения.

Из минусов данной архитектуры стоит отметить:

1. время выполнения операции. Так как все приложение разбито на множество микросервисов связь между которыми осуществляется по сети, к времени выполнения самих вычислений нужно прибавить время выполнения транзакций.
2. сложность эксплуатации. Каждый микросервис - это небольшой компонент, который легко понять, однако по мере роста приложения количество микросервисов растёт и сложность понимания проекта проявляется в понимании взаимосвязи между сервисами.

Несмотря на более большое количество времени необходимое для выполнения одной операции, на текущий момент основной концепцией организации распределенных информационных систем считается микросервисный подход. Однако существуют различные способы ускорения выполнения операции.

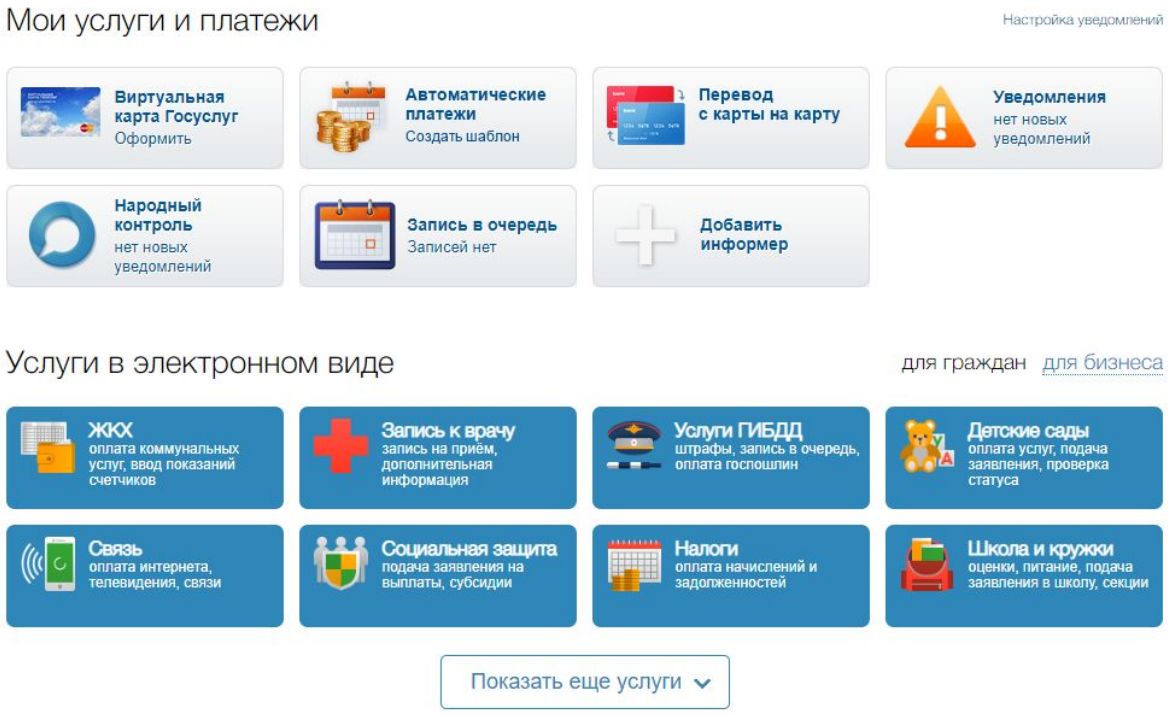
Так как одной из слабых точек микросервисов является скорость транзакций, то есть перед тем как операция завершится микросервис может обратиться к множеству других микросервисов, которые в свою очередь обратятся к еще определённому множеству. И дабы ускорить этот процесс в некоторых системах обращение к микросервисам осуществляется параллельно, таким образом время выполнения операции вычисляется не суммой времени ответов от микросервисов, а самым

долгим из них.

Так как микросервисы позволяют не привязываться к определенным инструментам, существует возможность выбирать максимально оптимальный способ реализации задачи.

1.2 СИСТЕМА ИНФОРМЕРОВ

Система информеров - система занимающаяся созданием, редактированием, обновлением, удалением информеров. Данная система используется для портала uslugi.tatarstan.ru.



Так как упомянутый выше портал сам по себе состоит из множества модулей микросервисная архитектура подходит для него лучше всего.

В основу любой системы лежат сущности, как базовые элементы над которыми происходят различные манипуляции, в данном случае система информеров не исключение. Ядром данной системы являются следующие сущности: услуги и информер.

Информер предназначен для того, чтобы уведомлять пользователя о какой-либо информации, тип информации определяется типом информера. Услуга предназначена для хранения данных, которые будут использоваться для получения информером информации из сторонних

сервисов. Набор данных определяется типом услуги. Также каждый тип информера относится к определённому типу услуги.

1.3 ТРЕБОВАНИЯ К СИСТЕМЕ ИНФОРМЕРОВ

Компания “Управление информационными проектами” выдвинула свои требования к системе информеров:

- реализовать надежную и отказоустойчивую архитектуру, под надежностью понимается то, как часто происходят сбои, под отказоустойчивостью - то, как реагирует система на сбои.
- система должна позволять наращивать мощность посредством горизонтального и вертикального масштабирования
- система должна предоставлять доступ к своей функциональности через API
- необходимо реализовать возможность обновления информеров, где для каждого информера была возможность задавать алгоритм обновления
- реализовать взаимодействие с системой информеров по протоколу HTTP с использованием REST API системы отчетов

Перечень основных методов REST API системы отчетов:

- обновление информера
- получение данных информера
- получение данных услуги
- создание информера
- создание услуги
- получение данных типа информера
- получение данных типа услуги
- создание типа информера
- создание типа услуги
- получение списка информеров по идентификатору услуги
- получения списка услуг
- получение списка типов информеров

- получение списка типов услуг

2. ПРОЕКТИРОВАНИЕ СИСТЕМЫ

2.1. ИСПОЛЬЗУЕМЫЕ ТЕХНОЛОГИИ

Для данного проекта было решено использовать язык программирования php версии 7.2, фреймворк symfony 4.2, брокер сообщений rabbitMQ, СУБД PosgreSQL.

Так как данный приложение реализуется по микросервисной архитектуре, не имеет значения для других микросервисов какие технологии использованы в конкретном микросервисе. Поэтому выбор технологии обуславливается лишь задачей.

В настоящее время можно столкнуться с огромным множеством языков программирования, каждый из которых имеет свои преимущества в своей области разработки. PHP входит в десятку популярных, эволюционируя с каждым разом с большими темпами, по сравнению с другими языками или платформами последние 5 лет [10]. PHP — язык программирования с открытым исходным кодом, разработанный для написания веб-разработок, исполняющийся на серверной стороне (Web-сервере) [9]. Можно выделить несколько особенностей и плюсов PHP для создания веб-приложений:

1. Бюджетность и экономичность — открытость исходного кода языка позволяет быстро создавать прототипы и готовые сайты, большое количество документаций, руководств и инструкций в свободном доступе, шаблонов и скриптов. Язык быстро интегрируется со многими бесплатными и популярными CMS. Все это делает стоимость написания программ на PHP ниже, по сравнению с другими языками.

2. Быстрые сроки реализации приложений — большое количество фреймворков позволяет создавать масштабируемые проекты в кратчайшие сроки.
3. . Возможность простого расширения функциональности путем добавления библиотек и расширений, распространенных в свободном доступе.
4. Кроссплатформенность — совместимость PHP с большим количеством операционных систем и популярных серверов.
5. Средства для автоматического управления памятью — удобство для разработчиков, так как нет необходимости думать об освобождении и распределении памяти после работы программы.

В компании “Управление информационными проектами” работают php разработчики, также в компании используется фреймворк симфони, поэтому поэтому дабы уменьшить порог вхождения были выбраны именно symfony и php. По той же причине была выбрана СУБД PosgreSQL.

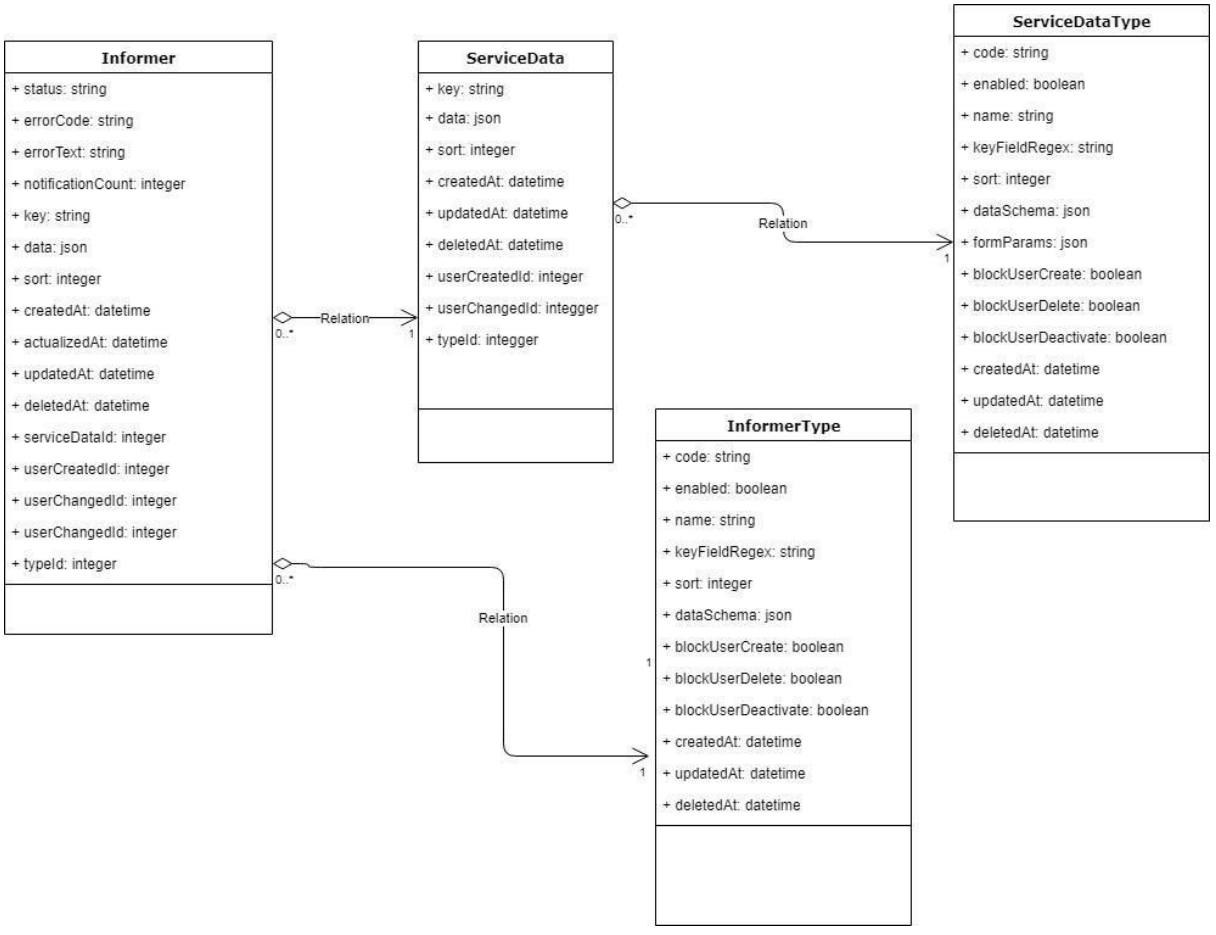
Из брокеров был выбран один самых популярных и простых в использовании - RabbitMQ.

2.2 СУЩНОСТИ

Так как работа системы информеров заключается в манипуляцией над сущностью “Информер” были выделены следующие сущности:

- Informer
- ServiceData
- InformerType
- ServiceData

Связь между которыми происходит следующим образом:



ServiceData - выступает в роли хранилища для данных, используемых в последствии информерами (Informer), тип которых (InformerType) относится к типу услуги (ServiceDataType).

Для хранения и работы с базой данных используется ORM библиотека Doctrine. Для каждой сущности написан репозиторий, через который можно получить необходимые данные и объекты связанные с этой сущностью.

3. РАЗРАБОТКА СИСТЕМЫ

3.1 Валидация

Так как данные услуги хранят в себе информацию, необходимую для информеров, может возникнуть ситуация, когда данных недостаточно, они не тех типов или же они просто невалидны по каким-то другим причинам.

Было решено добавить поля для таких сущностей как тип информера (InformerType) и тип услуги (ServiceDataType) поля хранящие json - схему содержащие данные о полях, которые в последующем будут использованы для валидации:

- данных информера по типу информера
- данных услуги по типу информера
- данных услуги по типу услуги

В данном случае обычных аннотаций фреймворка symfony недостаточно, из-за отсутствия валидации по json-схеме, однако можно заметить, что валидация проходит по похожей схеме: берутся данные, которые необходимо провалидировать, из сущности и json-схема по которой будет проходить валидация из типа информера или типа услуги, необходимо лишь передать валидатору необходимые данные.

Решено было писать собственную аннотацию и передавать ей выражение, через которое валидатор сможет определить откуда брать необходимые ему для валидации данные. Выражения были написаны используя синтаксис symfony expression language.А для валидации использовалась библиотека justinrainbow/json-schema валидирующая согласно стандарту предосталвенной “json-schema.org”.

Аннотация принимает два обязательных параметра:

- schema

- data

так же принимает такие параметры как exception для определения случаев, когда данная сущность не должна валидироваться. Исключение так же пишется, как выражение.

Сама валидация происходит после конвертации входящих данных в объект.

3.2. ОБНОВЛЕНИЕ ИНФОРМЕРА

Одной из задач, которую должна решать система информеров - это обновлять информацию информера, то есть в зависимости от типа информера получить эту информацию у стороннего сервиса или вычислить самостоятельно.

Для каждого способа обновления пишется отдельный класс наследующий **AbstractInformerUpdater**, в котором переопределяется метод update(Informer). Класс **AbstractInformerUpdater** в свою очередь имплементирует интерфейс **InformerUpdaterInterface**.
Выделение логики обновления в отдельный наследник **AbstractInformerUpdater-a** позволяет придерживаться одного из принципов ООП - полиморфизма, а наследование от абстрактного класса, а не от интерфейса напрямую решает проблему с дублированием кода.

С учетом того, что услугами системы информеров может пользоваться большое число пользователей, могут возникнуть ситуации, когда большое множество клиентов будут совершать запрос на обновление информера, что создаст большую нагрузку на сервер, что в свою очередь может вызвать нестабильную работу сервера. Довольно важно, чтобы сервер не страдал от высокой нагрузки из-за резкого повышения числа пользователей.

В случае системы информеров не принципиально будет ли информер обновлён моментально после отправки запроса на сервер или вычисление будет отложено на короткий срок. Исходя из этого было решено производить отложенные вычисления, а именно класть запрос на обновление информера в очередь, из которой в будущем определенный процесс уже заберёт сообщение и запустит обновление.

Для данного проекта было решено использовать очередь RabbitMQ и бандл **php-amqplib/RabbitMqBundle** для работы с ней.

Очередь RabbitMQ использует протокол AMQP, как и многие другие брокеры, через него она принимает и отправляет сообщения. Соответственно существуют стороны, которые отправляют и забирают оные, этим занимается поставщик (Producer) и подписчик (Consumer) соответственно.

В системе информеров в роли поставщика выступает класс **RabbitService**, который выделен как отдельный сервис и обладает двумя основными методами **sendMessageToUpdateInformer(Informer)** и **updateInformer(AMQPMessage),** которые отправляют сообщение на обновление и обрабатывают сообщение, полученное из очереди. Внутри сервиса при его инициализации в конструктор поступает ссылка на объект класса Producer, который позволяет отправлять сообщения в нужную очередь (конфигурация прописывается в services.yaml), и при необходимости можно добавить другие очереди, работать с которыми позволит этот сервис.

Метод **updateInformer** вызывается из класса **UpdateInformerConsumer,** экземпляры которого в будущем будут висеть в памяти, последовательно забирать сообщения из очереди и запускать их обновления. В упомянутом выше методе происходит чтение сообщения, получение, сохранение обновляемого информера из БД по информации полученной из **AMQPMessage,** вызов метода **updateInformer(Informer)** класса **InformerUpdaterService,** в котором через паттерн проектирования - фабрика создаётся наследник **AbstractInformerUpdater-a,** и запускается непосредственное обновление**.**

Таким образом, разделив между сервисами обязанности, сохраняется слабая связность и при необходимости изменения бизнес логики необходимо будет внести незначительные изменения в методы классов или добавить необходимую информацию в конфигурацию приложения.

3.3 API

Как уже было указано ранее, каждый микросервис связывается с другими через легковесные протоколы, например http. Было решено реализовывать REST API и документацию к ней.

Причины по которым был выбран REST API просты. Его аналог SOAP не подходит по причине того, что он обязует отправлять сообщения в формате XML. В свою очередь REST API позволяет использовать любые форматы, а также использовать методы http как каркас взаимодействия с нашим сервером.

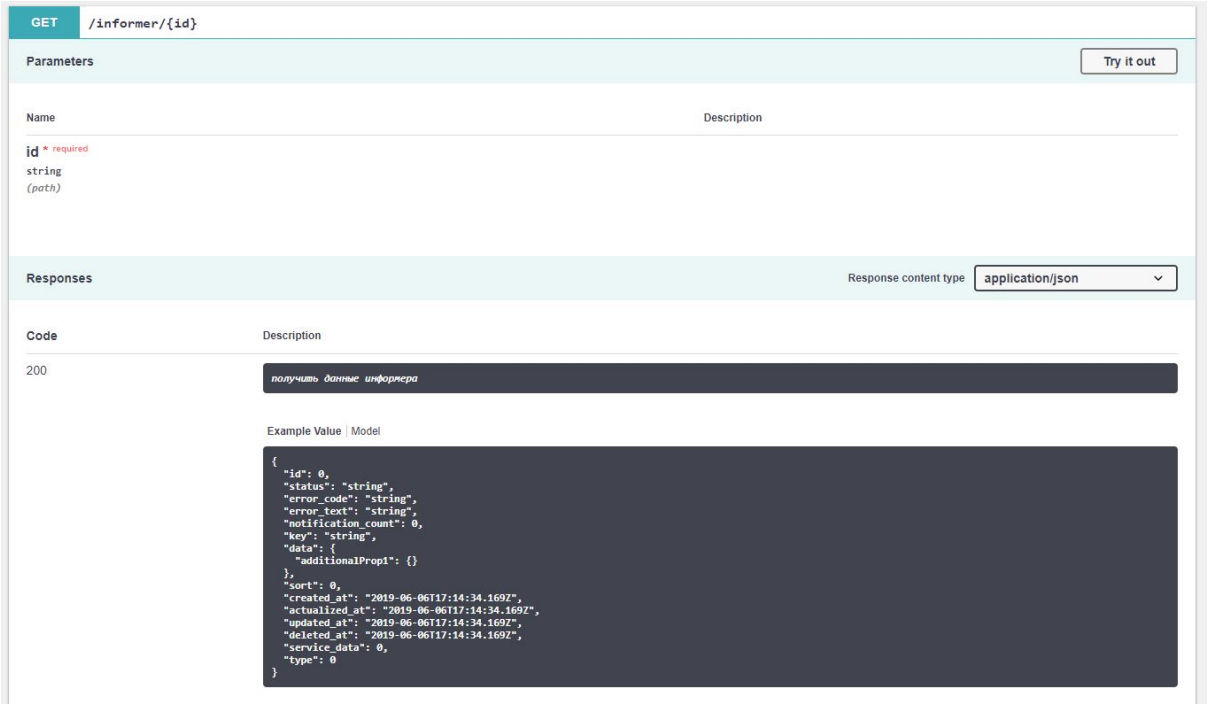
Были реализованы следующие методы:

- обновление информера
- получение информера
- сохранение информера
- получение списка услуг
- сохранение услуги
- получение услуги
- получение списка типа услуг
- получение типа услуги
- сохранение типа услуги
- получение списка типов информера
- получение типа информера
- создание типа информера

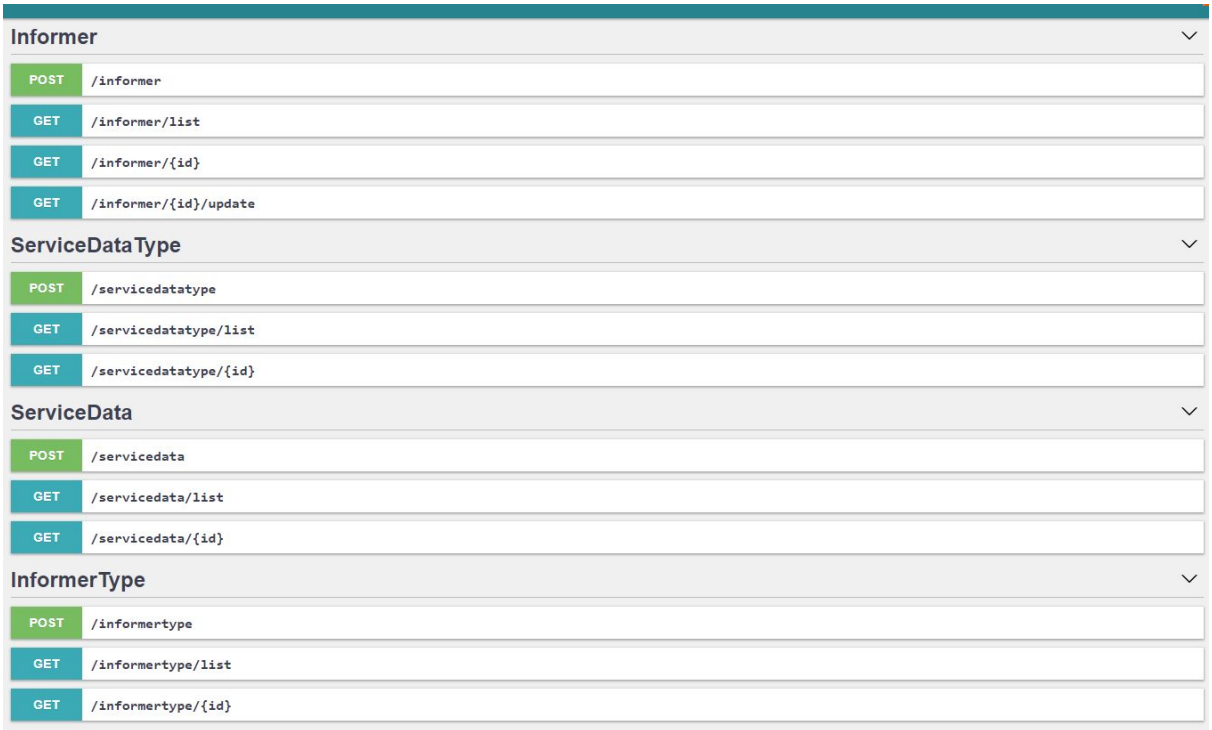
Для реализации api использовался fosRestBundle.

Для реализации документации использовался nelmioApiDocBundle.

Пример описания метода в документации:



Пример списка методов апи:



Заключение

Так как невозможно максимально точно оценить правильность выбора архитектуры приложения или же выбора тех или иных решений при реализации той или иной бизнес-логики, поэтому необходимо реализовывать приложения делая акцент на слабую связность, предполагая, что изменения в код будут вноситься часто.

В результате выполнения выпускной квалификационной работы был разработан микросервис системы информеров. Кодовая база получилась гибкой, конфигурируемой и читаемой, что позволит добавлять или изменять функционал затрачивая меньше ценных человеко-часов. По возможности максимально были использованы паттерны порождающие и поведенческие проектирования. Использованы современные технологии WEB-разработки.

В ходе работы были решены следующие задачи:

- Согласовать с компанией требования к системе информеров
- Разработать архитектуру приложения акцентируя внимание на расширяемость и конфигурируемость, выбрать технологии для реализации.
- Реализовать приложение.
- Разработать документацию к апи.

Компания «Управление информационными проектами» приняла разработанный микросервис системы информеров, как элемент микросервисной архитектуры.

В дальнейшем планируется добавить:

- модуль администрирования
- авторизацию через OAuth и JWT-token

- сервисы по обновлению и информера, имплементирующие созданный в рамках данной работы интерфейс

1.