План

0. Всякое говно
1. Описание предметной области, проблематики
    1. Описание трехзвенной архитектуры (Выделение основных проблем в подходе с API (+диаграммы))
    2. ☑ Функциональные и нефункциональные требования (Проблемы с типами, аргументами и в целом провалом в типизации SQL-C#)
    3. Стек??
2. Архитектура компонента (Fluda)
    1. ☑ Описание решение со спекой
    2. ☑ Схема размещения (деплой диаграма)
    3. ☑ Изложение основных сущностей (+Описание решения проблемы с типизацией)
    4. ☑ Описание работы системы на стороне клиента
    5. Описание работы системы на стороне сервера
3. Реализация
    1. Применение кодогенерации (Почему нам нужен код ген)
    2. ☑ SQL (Основные команды и их генерация (трансляция спеки в SQL-код))
    3. Roslyn (Рослин и его применение для кодгена)
    4. Непосредственно кодген (Разложение сущностей на компоненты, описание их генерации)
4. Демо?

> Написать где-то, что прокси может быть балансировщиком

Раздел 0. Теоретический этап

> Тут будут определения и все такое
> ЗАменить слово спека
> Написать, что такое СУБД!

.NET – программная платформа для разработки программ предназначенная для языков C# и Visual Basic.
Visual Basic (VB) – язык программирования.
Visual Studio - интегрированная среда разработки (IDE) предоставляемая Microsoft. Данная IDE разрабатывает во многом для фреймворка **.NET** и его языков (C++/CLI, C#, F#), но также имеет поддержку и тулзы для работы с SQL, Python, Javascript/Typescript, Java etc.
API - программный интерфейс приложения; описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.
Код-стайл - набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования.
Язык программирования - формальный язык, предназначенный для записи компьютерных программ
Компиляция – трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду

Раздел 1. Постановка проблемы

1.1 Описание трехзвенной архитектуры

> Трехзвенная архитектура, активити диаграммы

> Описание проблемы

Несмотря на распространенность клиент-серверных систем, не существует общепринятого решения проблемы взаимодействия клиента и базы данных на сервере. Во многом, это обусловлено требованиями к безопасности - нельзя давать возможность присылать со стороны клиента произвольный SQL код. Актуальность данной проблемы подчеркивается сложным процессом реализации такого обращения с использованием тривиального сценария с открытым API. Типичный сценарий добавления нового метода API для получения данных:

1. Написание SQL-процедуры для вытягивания данных, добавление ее в базу
2. Добавление в репозиторий (или заменяющий его слой абстракции) нового метода, который вызывает написанную процедуру
3. Добавление метода в API, который будет вызваться с клиента и обращаться к методу репозитория
4. Добавление на сторону клиента обращение к API

Почему нельзя отправлять уже готовый SQL-запрос с клиента? Это обусловлено требованиями к безопасности. Если разрешить выполнять произвольный SQL-скрипт, то это создаст уязвимость системы. К тому же, есть ряд ограничений к тому, какие данные может получить клиент. Например, пользователь имеет доступ только к его данным и любой SQL-скрипт должен дополнятся дополнительным условием проверки.

1.2 Функциональные и нефункциональные требования (Проблемы с типами, аргументами и в целом провалом в типизации SQL-C#)

Основное функциональное требование - возможность унифицировать код для сценария, когда есть больше одного способа получения данных из базы, а именно:

- обращение к локальной базе на стороне клиента
- запрос на сервер с целью получить какие-то данные с внешней базы данных без дополнительной логики
- отправка запроса через прокси сервис. Иными словами - реализация возможности полученный запрос прокидывать через несколько узлов.

Основное требование к безопасности в данной системе является причиной, почему существующие ORM нельзя применить для описанных выше сценариев - ORM генерирует SQL-запрос на стороне клиента, а выполнять произвольный SQL код на стороне сервера - не безопасно. Из этого следует, что нужно убедиться, что система безопасна от SQL-инъекций.

> Проверить, что пример с сеть - валидный

Также, к функциональным требованием является возможность модификации запроса на стороне сервера. Основная цель данного механизма - принудительно фильтровать только те данные, которые принадлежат клиенту, который запрос отправляет. Например, в случае если это база данных

социальной сети, то запрос клиента на получение всех групп должен возвращать только те группы, которые открытые или к которым он имеет доступ.

Одной из проблем, которую должна решать данная система - навязывание строгой типизации при работе между C# и SQL. Обычно, если в проекте не используется ORM, то участки кода где передаются аргументы в запрос или читается ответ от сервера - являются потенциально багоопасными. Частые проблемы, которые нельзя выявить на этапе компиляции, только во время выполнения:

- аргумент в процедуру передается с неправильным именем. Довольно часто это может быть miss-spell или невнимательность ("objectId" и "object_id").
- нет проверки на то, какой тип ожидается как аргумент или какой тип вернулся в ответ на запрос.

Ввиду того, что внедрение подобного механизма при условии поддержания типизации в уже существующей системе - это процесс, которые занимает очень много времени, к списку требований добавлен пункт на автоматизацию внедрения. Автоматизация заключается в реализации кодогенерации всего нужного кода на основе уже существующей базы данных.

1.3 Стек??

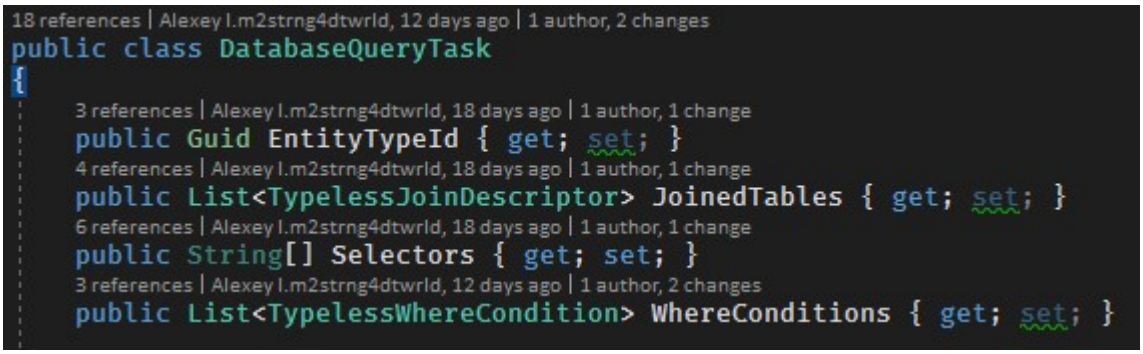
> ❤️ dotnet Что-то про "почему standard и кросплатформ" SQL server либа Roslyn.CSharp.Scripting Nuget

Раздел 2. Fluda

2.1 Описание решения со спекой

Для решения проблемы с передачей запроса от клиента к серверу или прокси, было внедрено промежуточное состояние запроса - сериализованное описание (спека). Идея заключается в том, чтобы предоставлять не готовый запрос, а описание для его генерации. Рассмотри минимально необходимый набор данных, который нужно хранить и передавать для построения запроса на получения данных. Указанные далее описания соответствуют базовым командам из SQL:

1. Идентификатор сущности, которую мы хотим получить. В упрощенной схеме - это просто таблица. Для этого можно использовать GUID. Соответствует указанию названия таблицы после `FROM`.
2. Для фильтрации сущностей нужно поддержать SQL-механизм WHERE. Учитывая, что в спеке уже лежит идентификатор таблицы, достаточно указать название столбца, нужное значение и способ сравнение (=, >, <, !=). Соответствует команде `WHERE`
3. Для реализации возможности указывать нужные столбцы таблицы, нужно в запросе также хранить их список. Соответствует перечислению столбцов после команды `SELECT`
4. Ввиду того, что почти всегда сущности декомпозируются для базы данных, логично предположить, что большая часть запросов будут обращаться к больше чем одной таблице. А значит, нужно поддерживать команду `JOIN` - хранить идентификатор таблицы которую нужно присоединить и пару столбцов, которые для этого будут использованы.



2.2. Схема размещения (деплой диаграма)

Рассмотрим схему работы системы на транспортном уровне. В общем случае существует клиент, который отправляет спеку и сервер, который обрабатывает ее и возвращает ответ от базы данных.

> На это все можно активити сделать

На стороне сервера реализован обработчик запросов к базе. Этапы обработки:

1. Чтение сериализованого запроса, десериализация
2. Генерация SQL-кода по запросу
3. Отправка SQL-запроса в базу, чтение результата
4. Сериализация результата, отправка на клиент

Рассмотрим три сценария работы. Самый простой случай - работа с локальной базой. В этом кейсе запрос создается и выполняется на одной машине:

1. Генерируем спеку
2. Транслируем его в SQL-запрос и выполняем на локальной базе
3. Считываем и возвращаем ответ базы данных

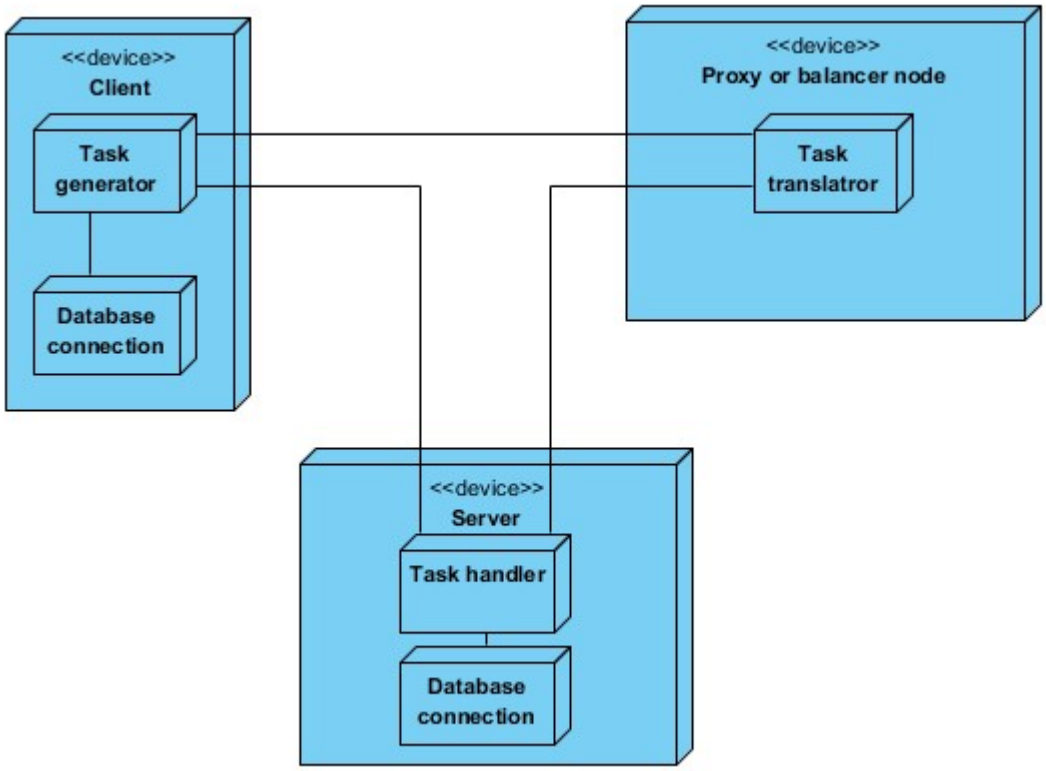
Более реальный кейс - это клиент-серверное общение:

1. Генерируем спеку
2. Сериализуем ее и отправляем запросом на сервер
3. На стороне сервера читываем запрос, выполняем дополнительную обработку, если нужно и выполняем на базе, которая находится на стороне сервера
4. Считываем, сериализуем и возвращаем ответ базы данных клиенту
5. Обрабатываем полученный результат на стороне сервера

Последний кейс, которые заложен в архитектуру приложения - общение Клиент <-> Прокси <-> Сервер. Сценарий обработки не сильно усложняется:

1. Генерируем спеку
2. Сериализуем ее и отправляем запросом на прокси
3. Прокси определяет, кто должен обработать данных запрос и отправляет соответствующему серверу.
4. На стороне сервера читываем запрос, выполняем дополнительную обработку, если нужно и выполняем на базе, которая находится на стороне сервера
5. Считываем, сериализуем и возвращаем ответ на прокси
6. Прокси пробрасывает ответ клиенту

7. Обрабатываем полученный результат на стороне сервера



2.3 Изложение основных сущностей

> Возможно, стоит добавить примеры

Очевидно, что для реализации заявленного функционала, нужно в первую очередь описать сущности базы данных в коде.

Самыми очевидными типами при работе с базой являются модели базы данных - классы, которые отображают в коде столбцы таблиц как переменные являясь дата-классами.

EntityAccessor

> Теперь тут еще и логика Selector'ов должна быть

Одной из базовых сущностей является таблица. Для описания каждой таблицы нужно создать класс, который опишет:

- Идентификатор таблицы - GUID значение, которое будет привязано к типу, а значит и каждой отдельно взятой таблице
- Описание таблица - название таблицы из базы данных, которое состоит из схемы и названия ("[dbo].[Users]")
- Описание столбцов таблицы - совокупность названия столбца из SQL, его тип

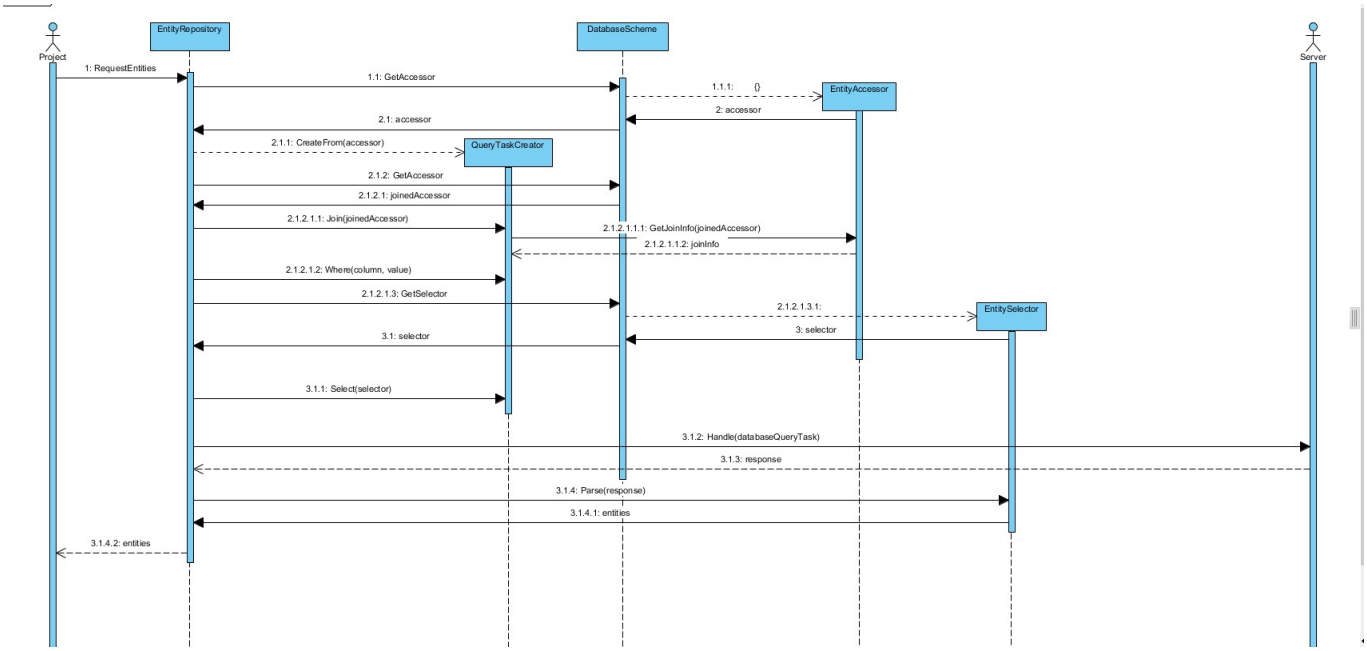
Помимо этого, для каждого такого класса описывается возможность объединить его с другим - возможность выполнить SQL-команду `JOIN` для этих двух таблиц. Для указания этого введет специальный интерфейс `IJoinable<OtherTable>`, который и указывает на то, что таблица может быть использована в команде `JOIN` с другой таблицей - `OtherTable`. Реализация данного интрфейса

заключается в добавлении одного метода (`Join`), который возращает пару столбцов, которые будут использованы в SQL-команде `JOIN`.

Что касается столбцов, указание типов позволяет не только в удобной форме реализовать парсинг ответа сервера, но также реализовать типобезопасность при генерации WHERE-команд. Для класса `Column<TEntityAccessor, TValue>` переопределены операторы сравнения, которые вместо булевого значения возвращают описание этого сравнения, которое можно добавить с спеку. В случае применения оператора для столбца и значения с разными типами, компилятор выдаст ошибку компиляции.

2.4. Описание работы системы на стороне клиента

Систему условно можно разделить на две основные части - клиентская часть и серверная часть. Клиентская часть является промежуточным слоем между клиентской частью основного продукта и абстрактным end-point'ом на стороне сервера, который обрабатывает запросы.



Самым абстрактным уровнем, с которым и будет взаимодействовать основной проект — это репозитории. Основная их задача - сокрытие всей логики и предоставление набора уже реализованных методов, которые будут доступны к использованию. Для доступа к `EntityAccessor` используется класс `DatabaseScheme`, который отображает содержит в себе инстансы всех `EntityAccessor`.

QueryTaskCreator

Механизм создания запроса соответствует паттерну Builder - класс QueryTaskCreator содержит набор методов для добавления какой-то информации в спеку:

- From - создает новую команду принимая на вход `EntityAccessor`

> Вот тут с дескриптором что-то такое себе

- `Where<TFieldValue>(WhereCondition<T1, TFieldValue> whereCondition)` - добавляет новое условие для `WHERE`-команды, принимает на вход дескриптор для данного `EntityAccessor`
- `Join<T2>(T2 joinedEntity)` - добавляет информацию о том, что нужно выполнить `JOIN` с таблицей чей `EntityAccessor` был передан в качестве аргумента. На этапе компиляции

выполняется проверка, что `joinedEntity` реализует `IJoinable<>` для нужного `EntityAccessor`

> Возможно, стоит нормально написать, но для начала нужно писать про селекторы

- `Select(IEntitySelector selector)` - метод, который принимает описание нужны для получения определенной сущности базы данных столбцов

По итогу получаем функциональность генерации описанной выше спеки с проверкой типов и уменьшением количества потенциальных багов.

Раздел 2.5. Описание работы системы на стороне сервера

Вторая часть системы - обработчик запросов. Основная логика - создание SQL-запроса по спеке. Стоит отметить, что для разных баз данных нужно генерировать разные SQL-запросы т.к. синтаксис будет отличаться. Для прототипа была реализована SQL-генерация совместимая с SQL Server'ом. Алгоритм обработки спеки:

1. Получаем по id сущность, которую нужно подставить в `FROM`
2. Выполняем рекурсивный обход спеки и достаем все JOIN'ы и WHERE.
3. Собираем все колонки для SELECT
4. Отправляем запрос в базу
5. Считывает ответ и сериализуем его в `Dictionary`

В результате таких действий, по спеке можно получить результат выполнения и спарсить его на стороне клиента.

Раздел 3

Раздел 3.1. Применение кодогенерации

Одним из нефункциональным требованием является кодогенерация нужных для использования Fluda компонентов. Довольно тривиальной проблемой при работе с базой данных является то, что табличные сущности (модели) нужно описывать дважды: в виде SQL-кода создания таблиц и на языке, который используется для бекенда, логика парсинга ответа от базы данных. Обычно, имеется два подхода:

1. Code first - подход, который закулючается в том, чтобы описывать сущности на языке бекенда, а потом использовать различные инструменты для создания базы данных на их основе.

> TODO: написать про EF

2. Db first - подход обратный, которые заключается в том, что сначала описывается база данных, таблицы, а потом генерируются шаблонные модели.

> TODO: какой-то переход к Флюде

Ввиду того, что основной задачей Fluda является реализация нового подхода, замена существующего функционала бекенда, ориентиром является именно существующая база, которая не будет изменятся.

3.2. SQL (Основные команды и их генерация (трансляция спеки в SQL-код))

> Написать про SQL особенности генерации запросов. Пояснить, почему это только для Sql Server и почему прикрутить что-то другое - не сложно

Основой для кодогенерации является база данных, ее схема. Если точнее - информация о том, какие есть сущности, что они из себя представляют (список столбцов) и какие между ними связи. Разные СУБД баз данных используют разный синтаксис для описания, минорные отличия в SQL. Рассмотрим самый популярные из них:

- Sql Server
- MySql
- PostgreSql

> Сравнение синтаксиса T-SQL и всякого мусора а-ля MySql PostgreSlq Написать как получить скрипт накатывания?

Входе сравнения выяснили, что отличия хоть и есть, но являются минорными и взаимозаменяемыи. Но ввиду того, что Sql Server является самым распространенной СУБД, именно он будет использоваться как основной. Рассмотрим основные SQL-конструкции, которые будут использованы для кодогенерации.

В первую очередь выполняется парсинг SQL скрипта создания базы для Sql Server. Выполняется парсинг на токены с помощью библиотеки TSQL.Parse. После этого выполняется деление на отдельные запросы по ключевому слову GO. Каждый запрос классифицируем в одну из категорий:

- CREATE TABLE

> Ну... View пока не готовы, это оч большая фича

- CREATE VIEW
- FOREIGN KEY
- Другое

Типичный запрос на создание состоит из:

- `CREATE TABLE [_1_].[_2_]`, где *1* - это схема, а *2* - это название таблицы, которое должно быть использовано как название сущностей/моделей
- Список описаний столбцов формата `[ModifiedDate] [datetime] NOT NULL`, где `ModifiedDate` - название столбца, которое должно соответствовать названиям полей класса, `[datetime]` - тип поля, которое должно быть переведено в тип используемого ЯП, `NOT NULL` - указание того, может ли поле не иметь значение, что соответственно должно отобразиться на полях моделей.
- `CONSTRAINT [PK_Person_BusinessEntityID] PRIMARY KEY` - описание ключей, которое не влияет никак на модели и кодогенерацию.

Помимо таблиц, есть также `View`, которые имеют очень похожую структуру скрипта создания за исключением того, что явно не указываются типы полей - они берутся из внешних таблиц.

Также, к схеме базы можно отнести `Foreign key` - их можно воспринимать как описание связей между двумя таблицами, явное указание, что эти таблицы можно джоинить. Шаблон такого скрипта:

- `ALTER TABLE [Sales].[SalesPerson]` - указание одной из таблиц, котороя будет джоиниться
- `WITH CHECK ADD CONSTRAINT [FK_SalesPerson_SalesTerritory_TerritoryID]` - SQL-синтаксис для указания, что это FK, не имеет смысловой нагрузки в данном контексте.
- `FOREIGN KEY([TerritoryID])` - указание столбца первой таблицы, который используется для джоина
- `REFERENCES [Sales].[SalesTerritory] ([TerritoryID])` - указание второй таблицы и ее столбца для джоина

Раздел 3.3. Roslyn

> Это куски курсача. Возможно, стоит сделать вычитку

Roslyn - компилятор исходного кода .NET, который предоставляется как CaaP (Compiler-as-a-Platform). Является проектом Microsoft с открытым исходным кодом. Поддерживаемыми языками являются C# и VB, который используют .NET в качестве среды выполнения. Основная идея проекта Roslyn заключается в раскрытии процесса компиляции, предоставление доступа пользователям к взаимодействию с различными его этапами. Гибкость в плане расширений и возможность кастомизации дают возможность применять данный инструмент в качестве внешнего анализатора кода.

Архитектура Roslyn'а имеет ряд особенностей, которые объясняются набором функционал, который данное продукт реализует. В первую очередь, данное средство автоматизации является компилятором и именно логика процессов компиляции лежит в основании всей системы. Остальные компоненты являются функциональными расширениями процесса компиляции и являются зависимыми от компоненты "Compilers". Не смотря на то, что другие компонент группируются на основе какой-то функционала, они взаимодействуют между собой, переиспользуют структуры данных других модулей, используют их зависимости и расширяют их. Этим обусловлена сильная связанность между ними. На диаграмме представлена модель системы, где элементами являются именно эти компоненты.

Раскрытие «черного ящика» компиляторов дает широкий ряд возможностей для улучшения условий работы с кодом. Программный компонент Roslyn, который является платформой предоставляющей функционал компилятора, как раз таки нацелена на интеграцию этих процессов, их управление и использование в процессе разработки. Часть этих компонент предоставляются как библиотеки для проектов с целью их расширения и имплементации на их основе нового функционала. При анализе было отмечено, что программный компонент представляется на разных уровнях абстракции – с ним можно интегрироваться как на уровне добавления правил код-стайла, так и внедрить дополнительную логику разбора исходного кода. Обобщив информацию полученную в ходе работы над Roslyn, можно прийти к заключению, что это современный инструмент, который значительно может упростить работы как с анализом кода, так и его написанием (например, кодогенерацией), а также дает возможность пользователям разрабатывать инструменты, которые решают непосредственно их задачу и настраивать уже существующие под себя.

Раздел 3.4. Непосренственно кодген

Поскольку все сущности, столбцы и связи между ними имеют зависимость от базы, было решено реализовать кодогенерацию их описания по SQL-скрипту создания базы, который можно вытащить из существующей.

Рассмотрим генерацию любого класса `*EntityAccessor`. В первую очередь стоит определиться с ожидаемой структурой класса, нужным набором интерфейсов и методов:

1. Для того, чтобы класс компилился, в файле должны быть сгенерированы нужные using'и. А также, класс должен быть помещен в специальный `Namespace`.
2. Класс должен иметь соответствующее таблице название, а также реализовывать набор всех нужных `IJoinable` интерфейсов в зависимости от FK базы. Помимо этого, должны быть еще два интерфейса - `IEntityAccessor` и `ISelector`.
3. Нужные для работы поля, которые описаны в IEntityAccessor - `Guid EntityId` и `TableDescriptor TableDescriptor`, метод `GenerateJoinRow`
4. Описание столбцов таблицы с помощью `Column<>` и их инициализация
5. Имплементация методов из `IJoinable`
6. Имплементация методов из `ISelector`

Раздел 4. Демо

> Возможно, в конце стоит описать как это вкручено в какой-то опенсорс проект. Или узнать, что по NDA можно рассказать и описать реальные кейсы