**Министерство образования и науки Российской Федерации**

**федеральное государственное автономное образовательное учреждение высшего образования**

**“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ**

**УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ,**

**МЕХАНИКИ И ОПТИКИ”**

Факультет информационных технологий и программирования

Образовательная программа: программирование и интернет-технологии

Направление подготовки (специальность): информационные системы и технологии

О Т Ч Е Т

о *производственной* практике

Тема задания: Разработка модуля для взаимодействия с базой данных в клиент-серверной архитектуре

Обучающийся: Ивницкий Алексей, M3305

Руководитель практики от профильной организации: Бабушкин Д.О., руководитель группы

Руководитель практики от университета: Зубок Д. А., доцент

Практика пройдена с оценкой \_\_\_\_

Подписи членов комиссии:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *Зубок Д. А.*

(подпись)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *Маятин А. В.*

(подпись)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *Повышев В. В.*

(подпись)

Дата \_\_\_\_

Санкт-Петербург

2019

# Постановка задачи

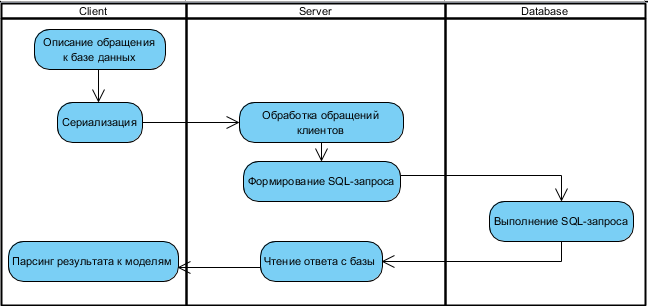
Основной задачей в ходе прохождении практики была разработка модуля взаимодействия с базой данных в клиент-серверной системе.

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

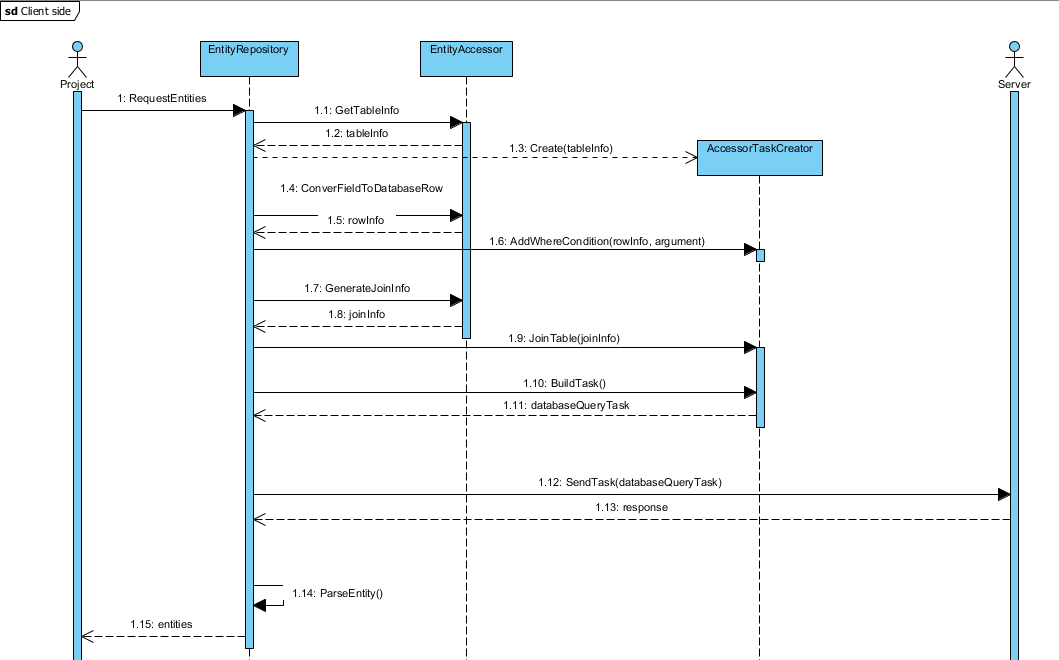
# Схема взаимодействия компонентов системы

Одним из вариантов решения данной проблемы является формализация команды-запроса для обращения клиента. Идея заключается в описании обращения клиента определенным JSON или XML объектом и реализации механизма его обработки на стороне клиента. Ниже представлена схема работы данной системы:



# Дополнительная логика 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 ограниченного набора методов, которые будут доступны к использованию.

За формирование команды-запроса отвечает AccessorTaskCreator. Это сущность, которая реализует паттерн Builder - принимает на вход параметры, нужные для построения команды, и на выходе генерирует ее. Основные требования - реализация возможности добавления в команду-запрос информации о join- и where-командах (для описания логики работы соответствующих SQL-выражений).

Одной из особенностей, заложенных в архитектуру, является типизация. Идея заключается в описании сущностей базы данных (таких как таблицы, столбцы и связи) средствами языка, чтобы в дальнейшем поддерживать строгую типизацию и минимизировать проблемы, находить их на этапе компиляции. Соответствие этим требованием реализовано в слое абстракции EntityAccessor. Отдельно взятый экземпляр данной абстракции представляет собой таблицу в базе данных: дескриптор таблицы (полное название таблицы), список столбцов (с указанием названия и типа), описание возможных join-ов к данной таблице.

# Логика на стороне сервера

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

# Практическая часть

В ходе работы был разработан прототип. В качестве тестовых данных использована база данных Adventure World.

В первую очередь реализованы EntityAccessor, например для таблицы Production.Product:

**public** **class** ProductAccessor : IEntityAccessor<AdventureWorksTable>

{

**public** **readonly** SqlFieldDescriptor<ProductAccessor, Boolean> FinishedGoodsFlag;

**public** **readonly** SqlFieldDescriptor<ProductAccessor, Boolean> MakeFlag;

**public** **readonly** SqlFieldDescriptor<ProductAccessor, String> Name;

**public** **readonly** SqlFieldDescriptor<ProductAccessor, Int32> ProductId;

**public** **readonly** SqlFieldDescriptor<ProductAccessor, Int32?> ProductModelId;

**public** **readonly** SqlFieldDescriptor<ProductAccessor, String> ProductNumber;

**public** **readonly** SqlFieldDescriptor<ProductAccessor, Int32?> ProductSubcategoryId;

**public** **readonly** SqlFieldDescriptor<ProductAccessor, Int16> ReorderPoint;

**public** AdventureWorksTable EntityType { **get**; } = AdventureWorksTable.ProductionProduct;

**public** TableDescriptor TableDescriptor { **get**; } = **new** TableDescriptor("Production", "Product");

}

SqlFieldDescriptor – типизированная обертка для столбца таблицы, для нее явно указывается тип. Это используется в дальнейшем как для формирования where-выражений, так и для парсинга ответа из базы данных. Для этой же таблицы написан пример репозитория:

**public** **class** ProductRepository : IRepository<AdventureWorksTable, ProductAccessor>

{

**public** AccessorTaskBuilder<AdventureWorksTable, ProductAccessor> Creator { **get**; } =

**new** AccessorTaskBuilder<AdventureWorksTable, ProductAccessor>(Production.Product);

**public** DatabaseQueryTask<AdventureWorksTable> GetByCategoryName(**string** categoryName)

    {

**return** Creator

            .SelectEntity()

            .Join(ProductAccessor.WithSubcategory()

                .Join(ProductSubcategoryAccessor

                    .WithCategory()

                    .Where(Production.ProductCategory.Name == categoryName)))

            .Build();

    }

**public** **static** Product Parse(TransferObject transfer)

    {

**return** **new** Product

        {

            Name = Production.Product.Name.GetFrom(transfer),

            FinishedGoodsFlag = Production.Product.FinishedGoodsFlag.GetFrom(transfer),

            ProductId = Production.Product.ProductId.GetFrom(transfer),

            MakeFlag = Production.Product.MakeFlag.GetFrom(transfer),

            ProductModelId = Production.Product.ProductModelId.GetFrom(transfer),

            ProductNumber = Production.Product.ProductNumber.GetFrom(transfer),

            ProductSubcategoryId = Production.Product.ProductSubcategoryId.GetFrom(transfer),

            ReorderPoint = Production.Product.ReorderPoint.GetFrom(transfer)

        };

    }

}

Он содержит тестовый метод создания запроса на получения из базы продуктов по указанной категории. Продемонстрирован синтаксис использования join- и where-выражений. Ниже представлен код билдера запроса. Стоит отметить, что для поддержания типизации было использовано много обобщений. Это позволит системе быть гибкой, но при этом контролировать валидность всех шагов построения запроса.

public class AccessorTaskBuilder<TEntityTypeEnum, TEntityAccessor>

where TEntityTypeEnum : Enum

where TEntityAccessor : IEntityAccessor<TEntityTypeEnum>

{

private DatabaseQueryTask<TEntityTypeEnum> \_currentTask;

private readonly TEntityAccessor \_entityAccessor;

public AccessorTaskBuilder(TEntityAccessor entityAccessor)

{

\_currentTask = new DatabaseQueryTask<TEntityTypeEnum>(entityAccessor.EntityType, new String[0]);

\_entityAccessor = entityAccessor;

}

public DatabaseQueryTask<TEntityTypeEnum> Build()

{

return \_currentTask;

}

public AccessorTaskBuilder<TEntityTypeEnum, TEntityAccessor> SelectAllField()

{

\_currentTask = new DatabaseQueryTask<TEntityTypeEnum>(\_entityAccessor.EntityType, new String[0]);

return this;

}

public AccessorTaskBuilder<TEntityTypeEnum, TEntityAccessor> Join<TJoinAccessor>(

JoinDescriptor<TEntityTypeEnum, TEntityAccessor, TJoinAccessor> joinDescriptor)

where TJoinAccessor : IEntityAccessor<TEntityTypeEnum>

{

\_currentTask.JoinedTables.Add(joinDescriptor.ToTypeless());

return this;

}

public AccessorTaskBuilder<TEntityTypeEnum, TEntityAccessor> Where<TFieldValue>(

WhereDescriptor<TEntityAccessor, TFieldValue> whereDescriptor)

{

\_currentTask.WhereConditions.Add(whereDescriptor.ToTypeless());

return this;

}

}

# Вывод

В ходе работы над модулем для взаимодействия с базой данных была проанализирована проблема в клиент-серверных системах. Описан возможный подход для решения проблемы, архитектура модуля, написана и представлена его реализация в соответствии с заявленным заданием. Продемонстрирована работа разработанного модуля на примере базы данных, основной функционал покрыт unit-тестами.