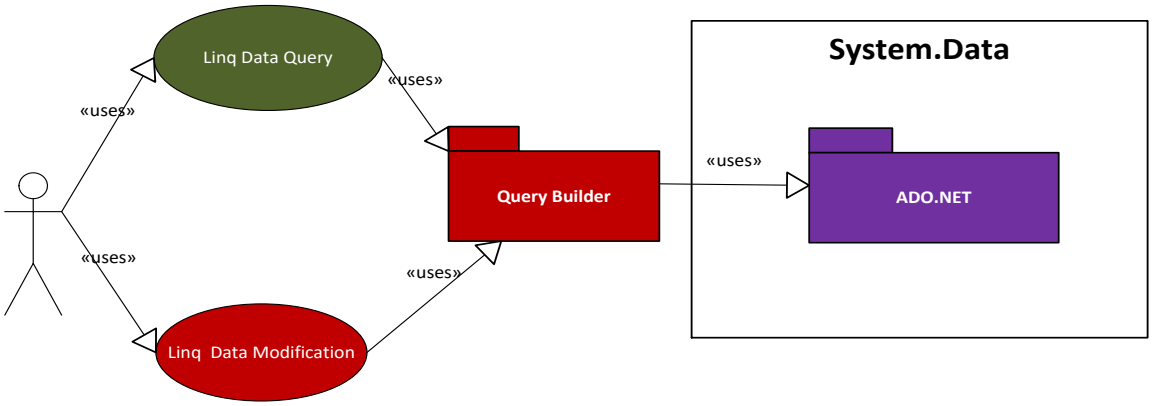
Mail : dementey@outlook.com

1. Основная концепция и проблематика



Основная концепция разработки функционала "Query Builder" заключается в создании максимально простого транслятора "Linq to MS SQL" с возможностью использования особенностей MS SQL, помогающих оптимизировать sql запросы (с возможностью максимально простого механизма определения метаданных).
Данный функционал призван существенно повысить производительность работы с базой данных. Аналоги подобных систем существуют (Entity Framework, Hibernate) но, как правило, являются законченными OR/M системами. В рамках вышеупомянутых нельзя управлять SQL - командами в полной мере.

Например:

EF : `var project = context.Projects.Where(x=>x.Name=="QueryBuilder");`
SQL: `SELECT * FROM [project] WHERE Name='QueryBuilder'`
В данном случае отсутствует возможность (в Entity Framework) указывать в sql – запросе Table Hints и прочие специальные расширения синтаксиса.

Пример:
SQL: `SELECT * FROM [project] with(nolock,index=ByName) WHERE Name='QueryBuilder'`

Стоит отметить, что вышеописанный пример позволяет значительно ускорить выполнение запроса за счет принудительного процедуры анализа автоблокировок (nolock) и процедуры выбора индекса для поиска данных(index=) MS SQL .

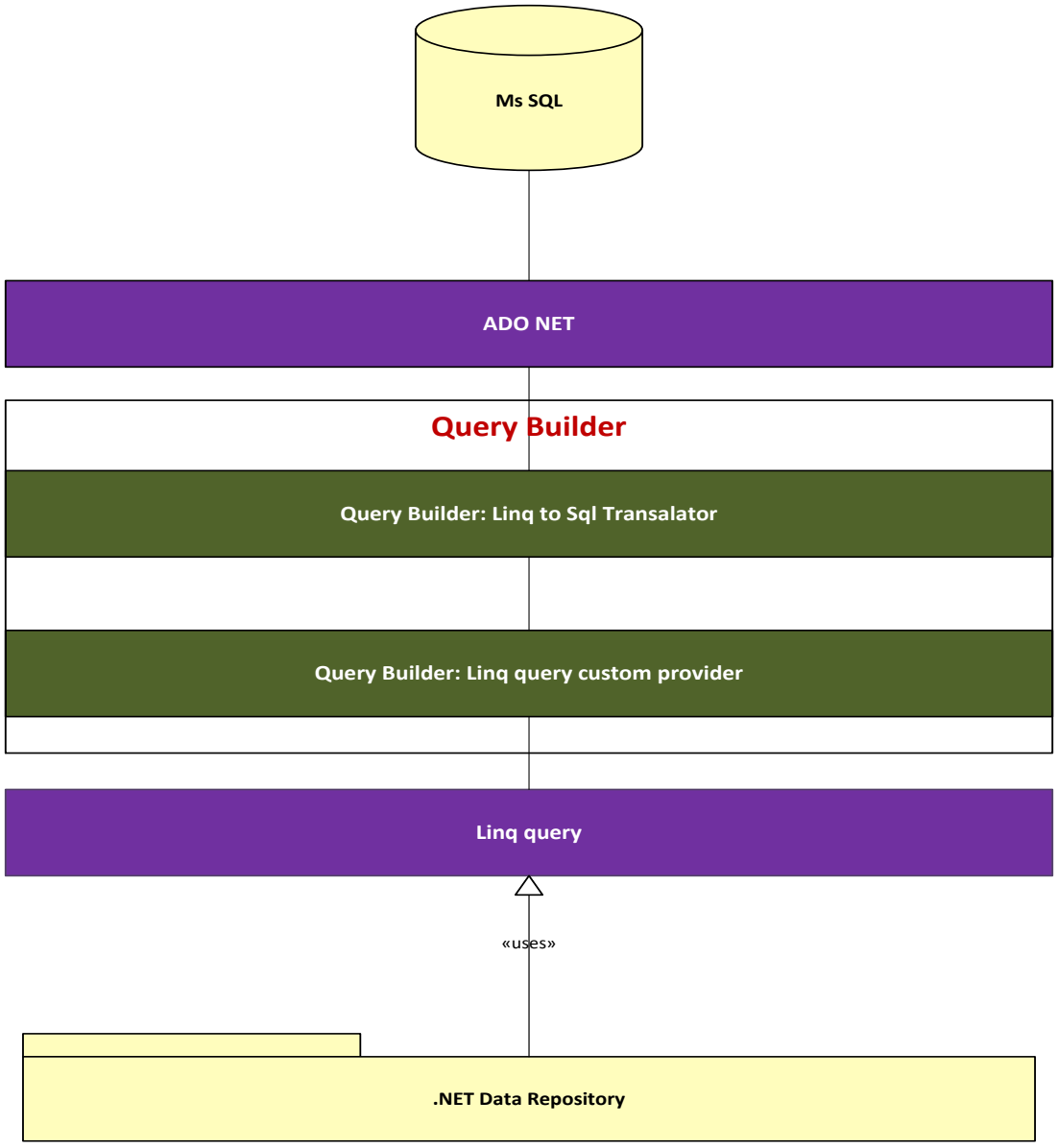
Также следует отметить, что в рамках концептуальной модели функциональных запросов LINQ не предусмотрены операций модификации (или групповой модификации) данных. Однако, модель SQL –коммуникаций предусматривает подобные взаимодействия.
Модель OQL (модель объектных запросов) в рамках стандарта предусматривает подобные возможности также.

Например, SQL команда: "`DELETE FROM [project] with (rowlock) WHERE Name='QueryBuilder'` " не имеет прямого аналога не в Linq не в Entity Framework.

Дополнительно стоит отметить громоздкость описания ORM – метаданных, где зачастую приходиться определять буквально все таблицы базы данных и связи между ними. **Тут**

ставиться задача определения только необходимых таблиц и связей с возможностью описания связей между сущностями как на уровне LINQ, так и на уровне SQL синтаксиса (sql injection).

2. Архитектура



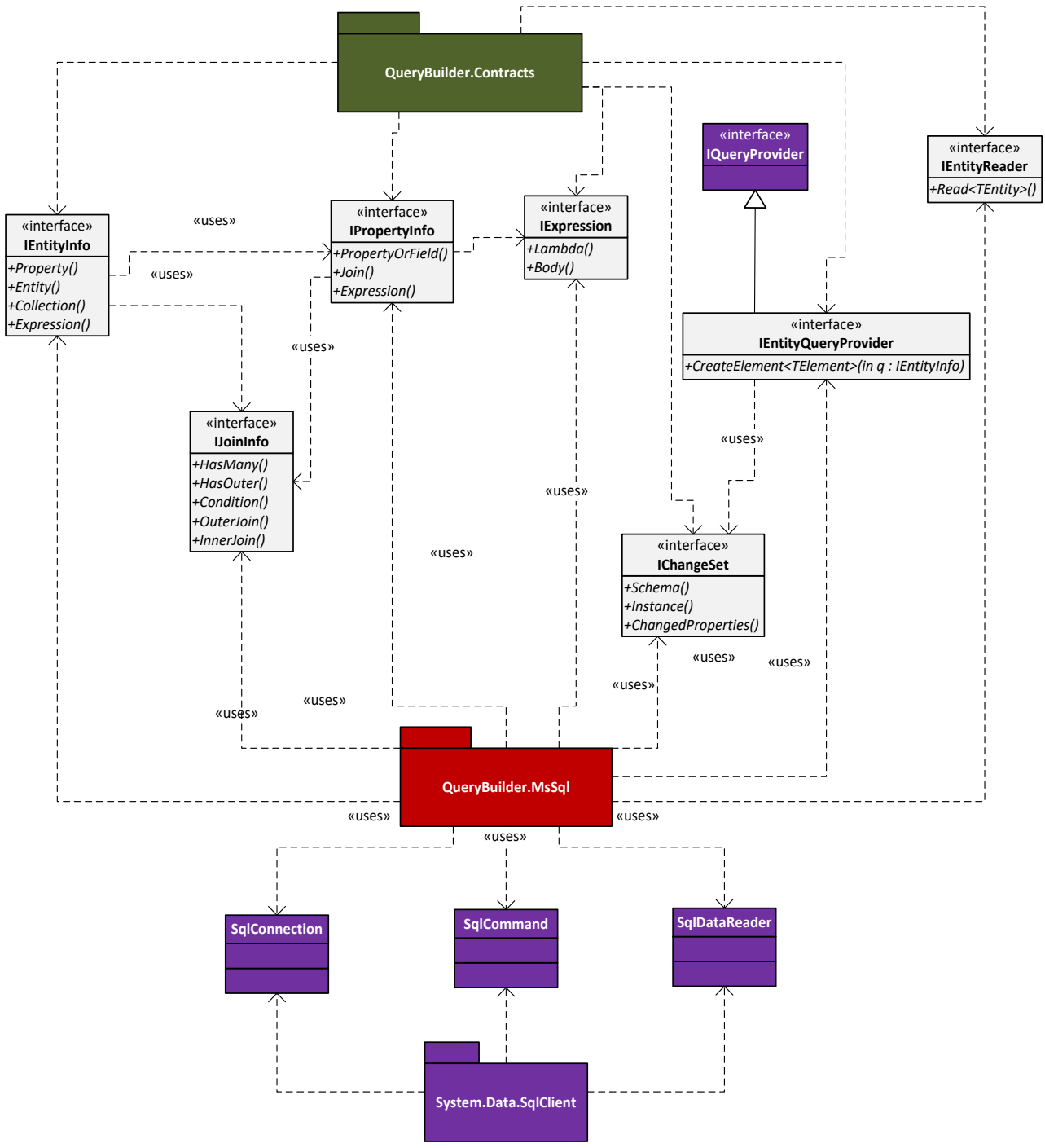
Архитектурно функционал "QueryBuilder" можно определить как промежуточный фунциональный слой , выполняющий роль транслятора LINQ - запросов в SQL - команды.

Функциональный слой "QueryBuilder" состоит из двух уровней:

- **Query Builder: Linq To Sql Transalator** – преобразует функциональные запросы Linq в SQL команды.

- **Query Builder: Linq query custom provider** – является реализацией стандартного “linq query provider” и обрабатывает стандартные для linq provider события.

3. Реализация



Mail : dementey@outlook.com

Используемые технологии:

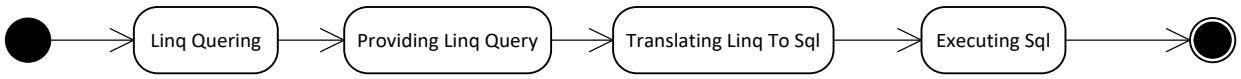
Типы	Технологии
Frameworks	.NET Framework 4.5 , ADO.NET
Programming Language	C#/Sql/Linq

Список реализуемых компонентов:

Component	Comment
QueryBuilder.Contracts	Реализует набор интерфейсов, мета классов, абстрактных классов , аспектов для и предоставляет уровень метаданных для описания запроса к абстрактному хранилищу данных.
QueryBuilder.MsSQL	Реализует функциональность построителя SQL- запросов и набор операций чтения/модификации данных для базы данных MS SQL.

Контракты системы:

Интерфейс	Описание
IEntityInfo	Интерфейс предназначен для создания схемы метаданных типа сущность(Entity).
IJoinInfo	Интерфейс предназначен для создания схемы метаданных типа связь между сущностями (Entity Relationship).
IPropertyInfo	Интерфейс предназначен для создания схемы метаданных типа свойство сущности(Entity Property).
IExpression	Интерфейс определения SQL или LINQ выражения для вычисляемых свойств сущности (Calculated Entity Property)
IEnityQueryProvider	Интерфейс расширяет стандартный **IQueryProvider** интерфейс путем добавления набора базовых операции модификации данных.
IChangeSet	Интерфейс определяет набор измененных свойств определенной сущности.
IEntityReader	Чтение состояния сущности (Entity Persistent State) из абстрактного хранилища данных.



В рамках **QueryBuilder.MsSQL** реализован функционал типа SqlQueryProvider провайдер определяющий процесс трансляции Linq запросов в Sql и обеспечивающий коммуникацию между QueryBuilder и ADO.

В рамках функциональности QueryBuilder не обрабатываются sql ошибки(исключения ado net). SQL- исключения транслируются на прикладной уровень стандартными средствами ado net.

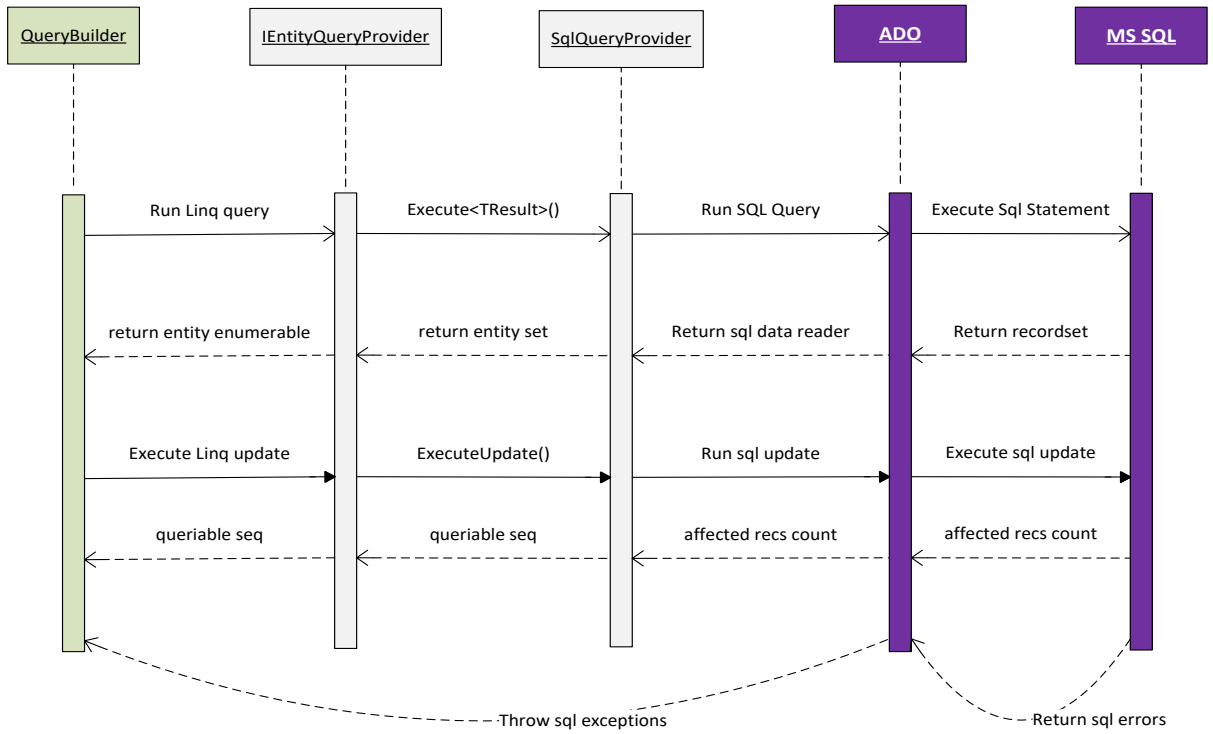


Диаграмма вызовов наглядно иллюстрирует процесс преобразования Linq запроса в sql команду. Так же для удобства прикладного использования необходимо определить статический аспект, расширяющий IQueriable интерфейс, для определения набора базовых операции изменения данных на уровне LINQ запроса.

```
  public static class QueryHelper
  {
   public static IQueryable<TEntity> Insert<TEntity>(this IQueryable<TEntity> query,
Expression<Func<TEntity>> newInit)
   {...}
    public static void Update<TEntity>(this IQueryable<TEntity> query, Expression<Func<TEntity,TEntity>>
init)
        {...}
   public static void Delete<TEntity>(this IQueryable<TEntity> query)
        {...}
}
```

Пример:

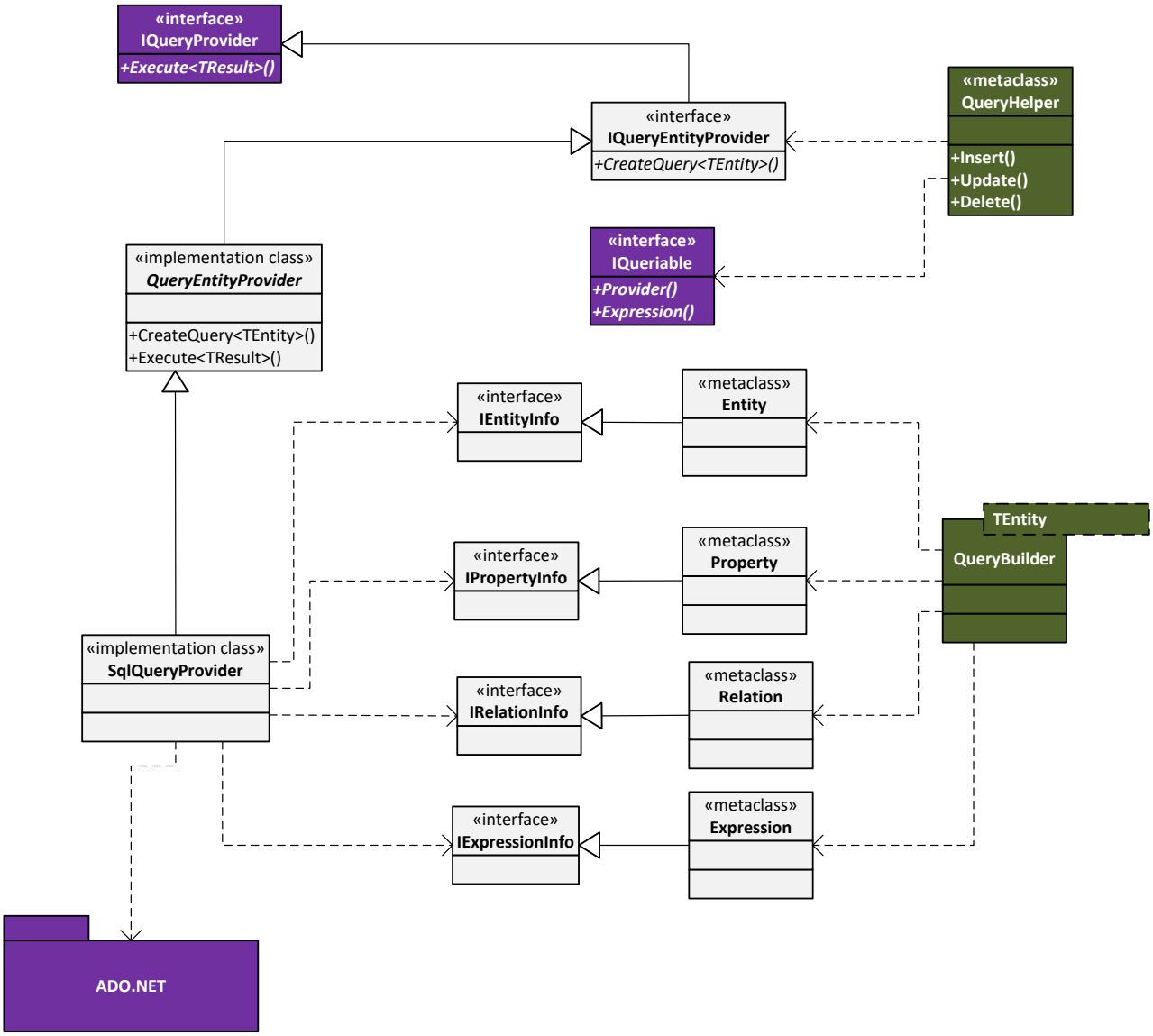
LINQ:

```
    queryTask.AsQueryable()
              .Where (x => x.Project.Name.Contains("ip"))
              .Update(x => new TaskModel {Estimation = x.Estimation + 1});
```

SQL:

```
UPDATE [projectpoint] with(rowlock) set [estimation]  = [estimation] + 1
FROM   [projectpoint] left join [project] with(nolock) on [projectpoint].[project] =
[project].[id]
WHERE  [project].[Name] like N'%ip%'
```

Mail : dementey@outlook.com



То есть MS SQL провайдер Linq-запросов использует указанный набор метаданных для корректного построения sql-запросов к базе данных MS SQL.

Пример:

```
QueryBuilder<EmployeeModel> queryEmployee = new QueryBuilder<EmployeeModel>("employee");

            queryEmployee
                  .Property(x => x.Id)
                  .Property(x => x.Name)
                  .Property(x => x.IsAdminRole);


queryEmployee.Provider(new SqlQueryProvider("Data Source=.;Initial
Catalog=projects;Integrated Security=true"));
```

LINQ:

```
var employees=queryEmployee.AsQueryable().Where(x=>x.IsAdminRole).ToArray();
```

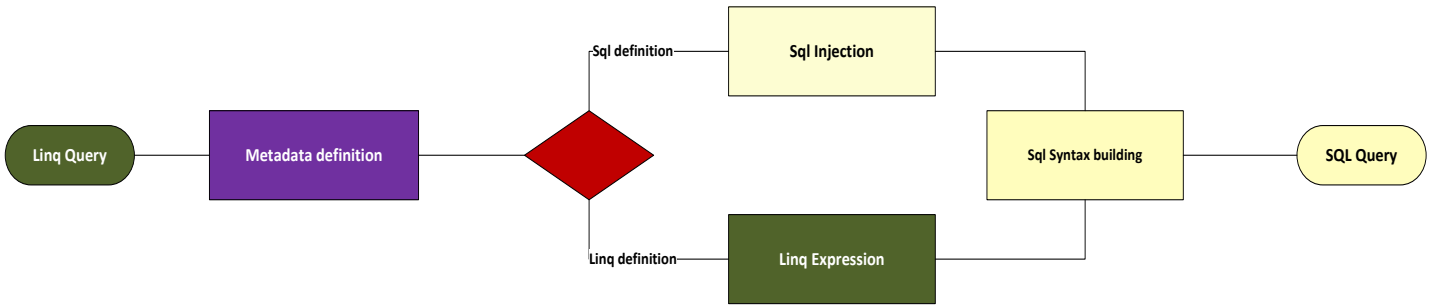
SQL:

```
SELECT Id=[employee].[Id],IsAdminRole=[employee].[IsAdminRole],Name=[employee].[Name] FROM
[employee] with(nolock) WHERE ([employee].[IsAdminRole]=1)
```

Определение связей и вычисляемых полей QueryBuilder:

```
        queryTask = new QueryBuilder<TaskModel>("projectPoint");


             queryTask
             .Property(x => x.Id       )
             .Property(x => x.Estimation)

             //Linq relations definition
             .Entity(x => x.Developer ).InnerJoin(queryEmployee, (x, y) => x.Developer.Id == y.Id)
             .Entity(x => x.Project   ).InnerJoin(queryProject , (x, y) => x.Project.Id   == y.Id)

            //sql injections
             .Entity(x => x.Project).InnerJoin(queryProject, "[projectPoint].project = [projects].id and
[projects].name is not null");



            queryProject.Collection(x => x.Tasks).OuterJoin(queryTask, (x, y) => x.Id == y.Project.Id);

            //sql injections:
            queryProject.Expression(x => x.CountOfTasks, "SELECT COUNT(*) FROM [projectPoint] with(nolock)
WHERE [projectPoint].project=[projects].id");

            //Linq expressions:
            queryProject.Expression(x => x.CountOfTasks, x=>x.Tasks.Count());
```

То есть на базе метаданных QueryBuilder необходимо определять связи и вычисляемые поля как на уровне Linq механизма, так и при помощи SQL – синтаксиста(sql injections).



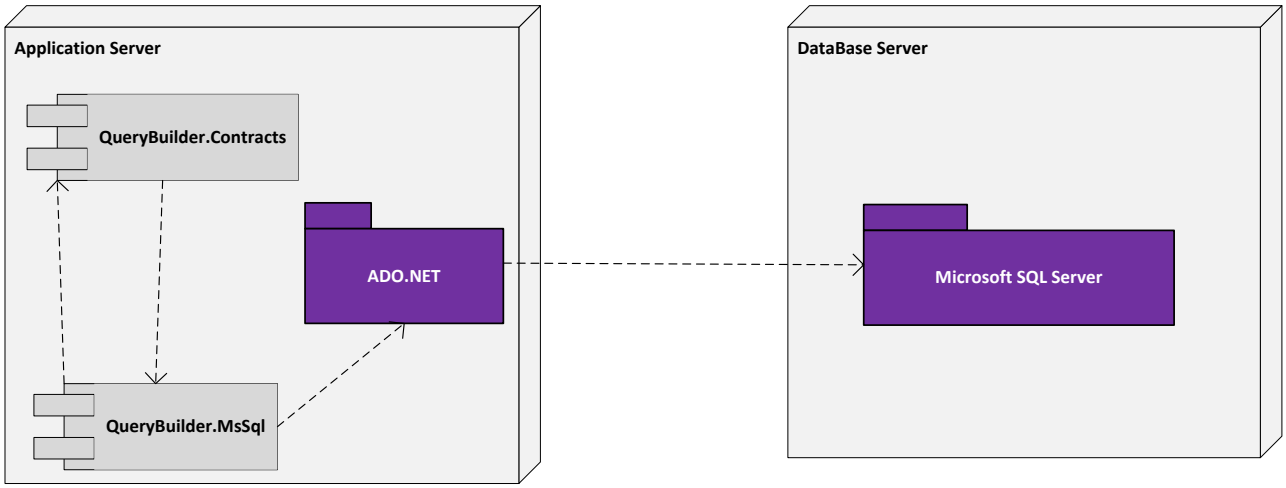
Необходимо учитывать некоторый набор проблем, возникающий при реализации двойного определения атрибутов сущностей через Linq и Sql-injections.
Например, полную или частичную потерю интероперабельности по базе данных (серверу СУБД), так как синтаксис запросов к SQL – базе данных и NoSQL – хранилищу существенно различается.

Однако, в определенных случаях, параметром интероперабельности можно пренебречь. Например, для достижения оптимальной производительность для конкретного сервера СУБД.

В случае же наличия двух и более разнотипных СУБД стоит использования Linq метод описания связей и атрибутов некоторой сущности.

Mail : dementey@outlook.com

4. Поставка



Поставка "QueryBuilder" осуществляется в виде двух dll:

1) *QueryBuilder.Contracts – Набор классов для определения запроса к абстрактному хранилищу данных в общем виде!*
2) *QueryBuilder.MsSql - Провайдер запросов для работы с MS SQL.*

Использование функциональности "QueryBuilder" показано для реализации внутренней логики работы сервера приложений, например для реализации паттерна "Repository", инкапсулирующего логику доступа к базе данных.

Restfull Web Api Controller example:

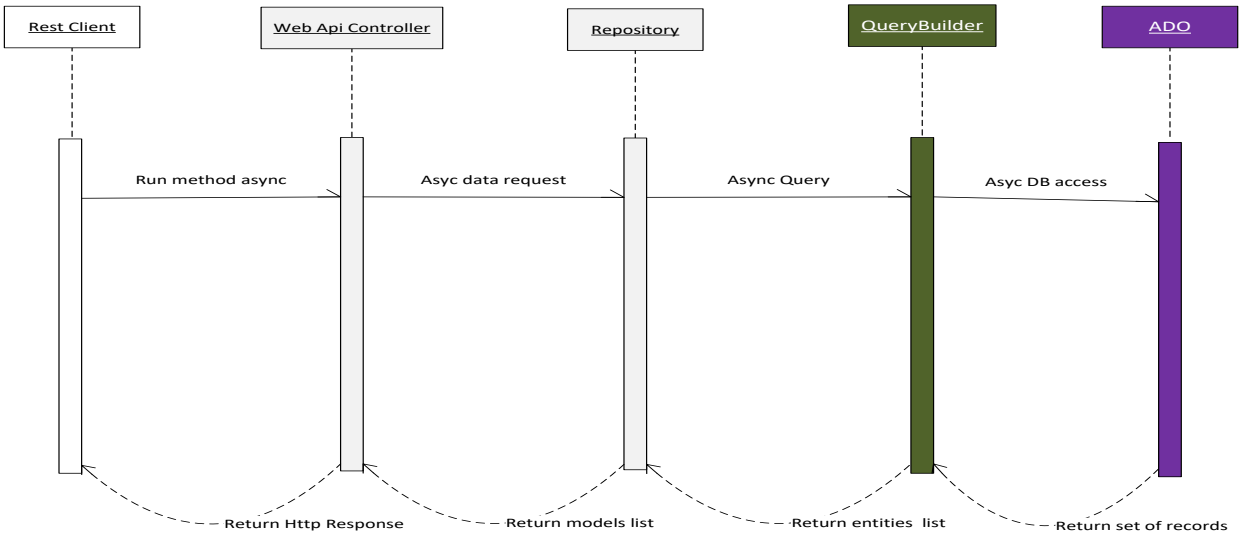
```
public class ProjectsController : ApiController
{
    private IProjectRepository repository = new ProjectRepository();

    [HttpGet , Route("projects/{maxCount}")]
    public  IEnumerable<ProjectModel> GetProjects(int maxCount)
    {
        return repository.GetProjects().Take(maxCount);
    }

     [HttpGet, Route("project/{employeeId}")]
    public ProjectModel GetProject(int projectId)
    {
        return repository.GetProjects().FirstOrDefault (x => x.Id == projectId);
    }



    [HttpPost, Route("project")]
    public ProjectModel AddProject(string name)
    {
        return repository.AddProject(name);
    }



    [HttpPut, Route("projects")]
    public void UpdateProject(ProjectModel project)
    {
        repository.UpdateProject(project);
    }
```

```
        [HttpDelete, Route("project/{projectId}")]
        public void DeleteProject(int projectId)
        {
            repository.DeleteProject(projectId);
        }
    }
```

Project Repository:

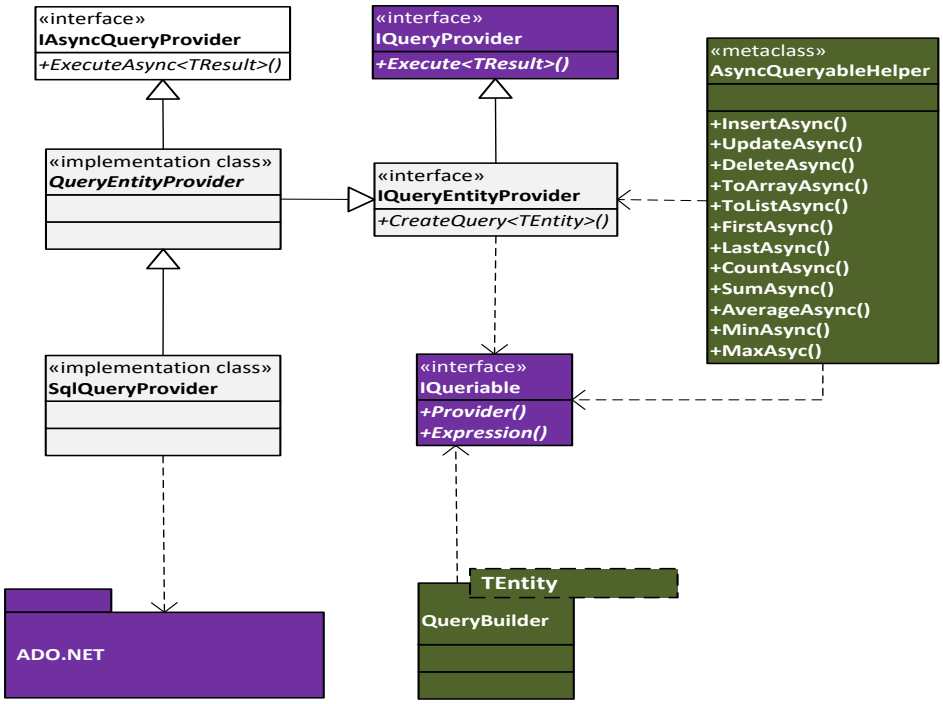
```
public class ProjectRepository : IProjectRepository
{
        private QueryBuilder<ProjectModel> queryProject;


        public ProjectRepository()
        {
             queryProject = new QueryBuilder<ProjectModel>("projects");

             queryProject
                    .Property(x => x.Id)
                    .Property(x => x.Name);

             queryProject.Provider(new SqlQueryProvider("Data Source=.; Initial Catalog=projects; Integrated
Security=true"));
        }



        public IQueryable<ProjectModel> GetProjects()
        {
            return queryProject.AsQueryable();
        }

        public ProjectModel AddProject(string projectName)
        {

            ProjectModel project = GetProjects().FirstOrDefault(x=>x.Name == projectName);

            if (project ==null)
            {
                GetProjects().Insert (()=> new ProjectModel {Name = projectName});
                project = GetProjects().FirstOrDefault(x=>x.Name == projectName);
            }

            return project;
       }


        public void UpdateProject(ProjectModel project)
        {

            GetProjects()
              .Where (x=>x.Id == project.Id)
              .Update(x => project);
        }

        public void DeleteProject(int projectId)
        {
            GetProjects()
                     .Where (x => x.Id == projectId)
                     .Delete();
        }

}
```

5. Развитие



Основная концепция развития функциональности “QueryBuilder” на данный момент заключается (с учетом развития асинхронного программирования в рамках технологий .NET) в развитии асинхронных методов работы с данными. На диаграмме вызовов представлена схема работы асинхронной цепочки вызовов в Rest – ориентированный системе. То есть Rest Client асинхронно вызывает ApiController, который в свою очередь асинхронно вызывает репозиторий. Репозиторий транслирует асинхронный вызов на Query Builder, который в свою очередь асинхронно работает с ADO.

Для обеспечения подобной асинхронности необходимо предусмотреть некоторые архитектурные дополнения инфраструктуры Query Builder:



То есть нужно определить некоторый набор методов, для асинхронной работы с данными на основе аспекта Query Helper и интерфейс IAsyncQueryProvider для реализации асинхронной работы с данными на уровне SqlQueryProvider.

Пример работы с асинхронным Query Builders:

Web Api Controller:

```
public class ProjectsController : ApiController
{
        private IProjectRepository repository = new ProjectRepository();

        [HttpGet , Route("projects/{maxCount}")]
        public  Task<IEnumerable<ProjectModel>> GetProjects(int maxCount)
        {
            return repository
                        .GetProjects ()
                        .Take        (maxCount)
                        .AsEnumerableAsync();
        }

        [HttpGet, Route("project/{employeeId}")]
        public Task<ProjectModel> GetProject(int projectId)
        {
            return repository.GetProjects().FirstOrDefaultAsync(x => x.Id == projectId);
        }

        [HttpPost, Route("project")]
        public Task<ProjectModel> AddProject(string name)
        {
            return repository.AddProject(name);
        }



        [HttpPut, Route("project")]
        public Task UpdateProject(ProjectModel project)
        {
            return repository.UpdateProject(project);
        }


        [HttpDelete, Route("project/{projectId}")]
        public Task DeleteProject(int projectId)
        {
            return repository.DeleteProject(projectId);
        }
 }
```

Repository:

```
  public class ProjectRepository : IProjectRepository
    {
        private QueryBuilder<ProjectModel> queryProject;
```

```
        public ProjectRepository()
        {
             queryProject = new QueryBuilder<ProjectModel>("projects");

             queryProject
                    .Property(x => x.Id)
                    .Property(x => x.Name);


             queryProject.Provider(new SqlQueryProvider("Data Source=.;Initial
Catalog=projects;Integrated Security=true"));
        }



        public IQueryable<ProjectModel> GetProjects()
        {
            return queryProject.AsQueryable();
        }


        public async Task<ProjectModel> AddProject(string projectName)
        {
            ProjectModel project = await GetProjects().FirstOrDefaultAsync(x=>x.Name ==
projectName);

           if (project ==null)
           {
            await GetProjects().InsertAsync(()=> new ProjectModel {Name = projectName});
            project = await GetProjects().FirstOrDefaultAsync(x => x.Name == projectName);
           }

            return project;
      }


        public Task UpdateProject(ProjectModel project)
        {

            return GetProjects().Where(x=>x.Id == project.Id).UpdateAsync(x => project);
        }

        public Task DeleteProject(int projectId)
        {
            return GetProjects().Where(x => x.Id == projectId).DeleteAsync();
        }



    }
```