ПРИМЕНЕНИЕ СОВРЕМЕННЫХ ТЕХНОЛОГИЙ, ПОДХОДОВ И ШАБЛОНОВ ПРОЕКТИРОВАНИЯ ПРИ РЕАЛИЗАЦИИ МАСШТАБИРУЕМОГО КЛИЕНТ-СЕРВЕРНОГО ПРОГРАММНОГО КОМПЛЕКСА

APPLICATION OF MODERN TECHNOLOGIES, APPROACHES AND DESIGN PATTERNS IN THE IMPLEMENTATION OF A SCALABLE CLIENT-SERVER SOFTWARE PACKAGE

Yu. Kostikov
V. Pavlov
A. Romanenkov

Summary. There is a constant need to process information of absolutely arbitrary nature, texture and structure in different branches of science and technology. In practice there are often no firmly standardized formats of input data and it complicates the procedure of input of initial information into the computer system. We propose in this article a methodology for organizing interaction with a database and implementing protocols for interaction between components of a monolithic or distributed system using modern approaches to constructing the architecture of software complexes.

Keywords: data storage, software complex, data processing, database, interface, ASPCore.netframework 4.6., WPF.

Костиков Юрий Александрович
К.ф.-м.н., Московский авиационный институт (национальный исследовательский университет)
jkostikov@mail.ru
Павлов Виталий Юрьевич
К.ф.-м.н., Московский авиационный институт (национальный исследовательский университет)
vitaly_pavlov@hotmail.ru
Романенков АлександрМихайлович
К.т.н., доцент, Московский авиационный институт (национальный исследовательский университет)
romanaleks@gmail.com

Аннотация. В разных отраслях науки и техники постоянно появляется необходимость обработки информации абсолютно произвольной природы, текстуры и структуры. Зачастую в практике отсутствуют твёрдо стандартизированные форматы входных данных, что усложняет процедуру ввода исходной информации в компьютерную систему. В данной работе предлагается методика организации взаимодействия с базой данных и реализация протоколов взаимодействия между компонентами монолитной или распределенной системы с помощью современных подходов построения архитектуры программных комплексов.

Ключевые слова: хранение данных, программный комплекс, обработка данных, база данных, интерфейс, ASPCore.netframework 4.6., WPF.

Введение

В современном мире постоянно растет потребность в различного рода программных продуктах и технологиях, значительно упрощающих обработку данных. Однако сложность самих программных систем постоянно растёт и, как следствие, растут требования к квалификации пользователей этих систем. Вместе с этим возрастают и потребности пользователей: приложения должны работать быстро, быть понятными и удобными с точки зрения пользовательского интерфейса, а также надежными в плане безопасности.

Для разработчиков программного обеспечения важно, чтобы приложение было легко поддерживаемым и модифицируемым, с сохранением понятной архитектуры и архитектуры.

Программное обеспечение, разработанное в 2000-х годах, которое одновременно отвечает и за формирование внешнего вида, и за эффективные вычисления, и за доступ к данным, уже не отвечает современным требованиям и признается устаревшим. Такие приложения перестают поддерживаться, и коммерческие фирмы вынуждены от них отказываться. Для удовлетворения вышеописанных современных требований к программному обеспечению сейчас повсеместно применяется клиент-серверная архитектура, позволяющая оптимально сбалансировать требования к быстродействию, удобству, надёжности и безопасности. Как следует из названия, эта архитектура разделяет обязанности приложения, выделяя две стороны: клиент и сервер.

Функциональные обязанности клиентского приложения заключаются в формировании запросов к серверному приложению, получения ответа от серверного приложения и обеспечение представления результатов в формате, удобном для пользователя.

Обязанности серверного приложения заключаются в получении запросов от клиентского приложения и интерпретации их, выполнении запросов к базе данных, от-

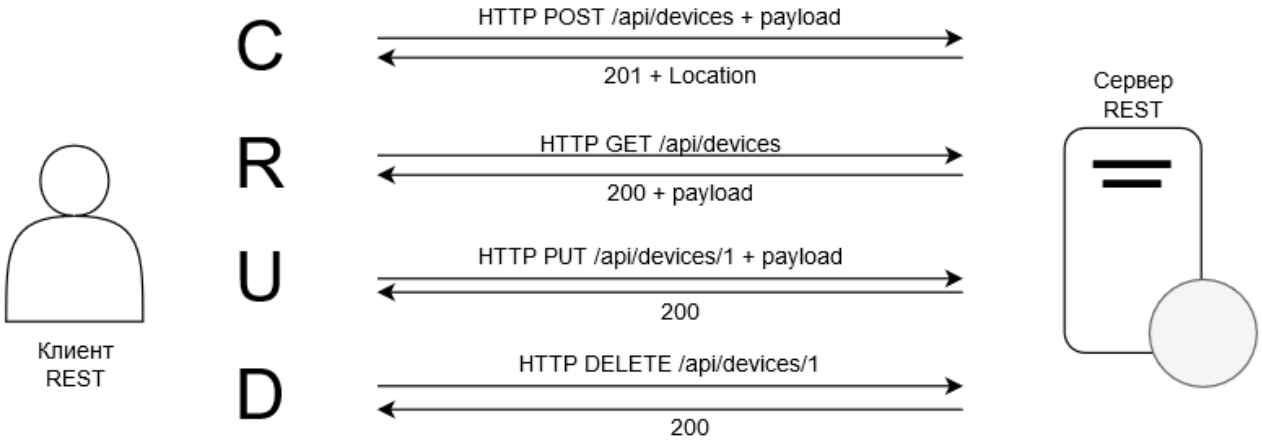

Рис. 1. REST архитектура

правку результатов клиентскому приложению, обеспечению разграничения доступа к данным, поддержание стабильного многопользовательского режима.

Серверное веб-приложение

В мире программных продуктов происходят заметные изменения. В последние годы наибольшей популярностью начинают пользоваться веб-приложения, которые развились от первоначального набора статических html-страниц до полноценных приложений, по своей сложности сравнимых с классическими оконными приложениями.

Тем не менее, сохраняется ряд идеологических отличий, которые заключаются в следующем: в использовании транспортного протокола и реального сетевого расположения, а так же в отсутствии состояния, то есть физического конкретного экземпляра приложения.

HTTP — протокол прикладного уровня для передачи данных в гипертекстовом формате — был представлен в 1992 году. В современном мире HTTP и его защищенный аналог HTTPS стал де факто стандартом взаимодействия между приложениями. Основополагающей моделью взаимодействия является схема запрос-ответ. Для каждого запроса клиентского приложения с серверной стороны возвращается ответ. Ключевая особенность для HTTP и всей веб-разработки в целом заключается в отсутствии состояния: веб-сервер может обрабатывать тысячи запросов от тысяч клиентов в секунду, причем веб-сервер не делает никаких отличий между различными клиентами, в отличие от классических приложений, где каждое ориентировано на конкретного пользователя. Все данные, которыми оперирует веб-приложение, хранятся на серверах баз данных, а доступ к ним контролируется с помощью систем управления базой данных (СУБД).

Для организации обмена данными и сообщениями между приложениями и между модулями приложений весьма эффективно используется REST архитектура.

Архитектура REST (REpresentational State Transfer) — стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения веб-служб. В общем случае REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат. Для HTTP-протокола действие над данными задается с помощью методов:

1. GET (получить)
2. PUT (добавить, заменить)
3. POST (добавить, изменить, удалить)

Схема взаимодействия между клиентом и сервером в архитектуре REST представлено на рисунке 1.

Опишем предлагаемую реализацию REST взаимодействия. Будем считать, что ресурс представляет собой один или несколько экземпляров класса, реализованного на языке C# 5.0.[1] Действия над ресурсом определены с помощью методов интерфейса языка C# 5.0. Отправка запроса разбивается на несколько стадий:

a. Авторизация на сервере
b. Формирование URL и отправка HTTP-запроса

c. Формирование тела ответа (в случае Post / Put) на сервере и отправка HTTP-ответа
d. Преобразование ответа в клиентском приложении и его интерпретация

Далее, организуем такой способ взаимодействия, который бы позволил один раз реализовать всю логику обращения к веб-серверу и интерпретацию ответов, а после использовать ее для каждого ресурса.

Основой данного решения является тот факт, что все методы взаимодействия являются однотипными, исходя как из их логики, так и из их программной реализации. Отличительными чертами каждого запроса являются:

1. Метод протокола HTTP (Get, Post, Put, Delete)
2. Имя контроллера веб-приложения, обрабатывающего запрос
3. Метод контроллера
4. Параметры запроса

В связи с этим можно определить некий один общий, базовый, метод, в который будут передаваться характеристики, и тот на их основании будет формировать запрос и принимать ответ.

Такое поведение позволяет реализовать библиотека Castle Dynamic Proxies, с помощью которой можно динамически генерировать классы на основе интерфейсов, при этом подставляя в реализацию каждого метода класса предопределенный код.

Создание и отправка HTTP-запросов происходят с помощью стандартных средств языка C# 5.0, но перед этим необходимо получить параметры запроса. Первые три удобно представить в виде атрибутов, при этом первый (метод протокола HTTP) будет применяться ко всему интерфейсу ресурса, а остальные два (имя и метод контроллера) будут относиться к каждому отдельному методу. Параметры запроса передаются в параметры метода.

Возникает следующая проблема: очевидно, что методы класса ресурса должны работать с самим ресурсом: принимать в качестве аргумента и возвращать из метода его результат, но в обобщенной генерации запроса ничего не известно о типе ресурса. В связи с этим используется рефлексия, позволяющая динамически во время выполнения приложения получать информацию о типах данных и непосредственно сами типы данных.

Предлагаемый алгоритм работы:

1. Определение базового поведения для взаимодействия со всеми ресурсами
   a. Представление запроса в виде сообщения протокола HTTP
   b. Отправка запроса веб-серверу

```
[HttpResource("someresource")]
public interface ISomeResourceClient
{
    [HttpMethod("list")]
    [HttpRequest(HttpRequestType.Get)]
    Task<IList<SomeResource>> ListAsync();

    [HttpMethod]
    [HttpRequest(HttpRequestType.Get)]
    Task<SomeResource> GetAsync(int id);

    [HttpMethod]
    [HttpRequest(HttpRequestType.Put)]
    Task CreateAsync(SomeResource resource);

    [HttpMethod]
    [HttpRequest(HttpRequestType.Post)]
    Task UpdateAsync(SomeResource resource);
}
```

   c. Получение ответа
   d. Извлечение данных из сообщения
2. Определение интерфейса, реализующего конкретный ресурс, и его методы
   a. Динамическое создание класса на языке C#, реализующего интерфейс ресурса
   b. Вызов соответствующих методов для доступа к ресурсам

На рисунке выше приведен пример кода на языке C# 5.0. интерфейса ресурса.

Организация работы с СУБД

Доступ к базе данных является одной из популярных тем для обсуждения в области разработки программного обеспечения. На протяжении долгого времени, подавляющей популярностью пользовался шаблон ActiveRecord [2]. Структура данного шаблона представлена на рисунке 2.

Суть шаблона сводится к следующему: каждый объект класса представляет собой обертку над строкой из соответствующей таблицы, содержит в себе поля, однозначно сопоставимые с колонками таблицы, и набор методов, каждый из которых генерирует запросы или команды на языке SQL.

Подход хорошо работает, когда связи между таблицами вида многие-ко-многим и один-ко-многим отсутствуют или сведены к минимуму, а также набор полей сущности невелик. К тому же, динамическое составление SQL запросов чревато ошибками и потенциальной угрозой

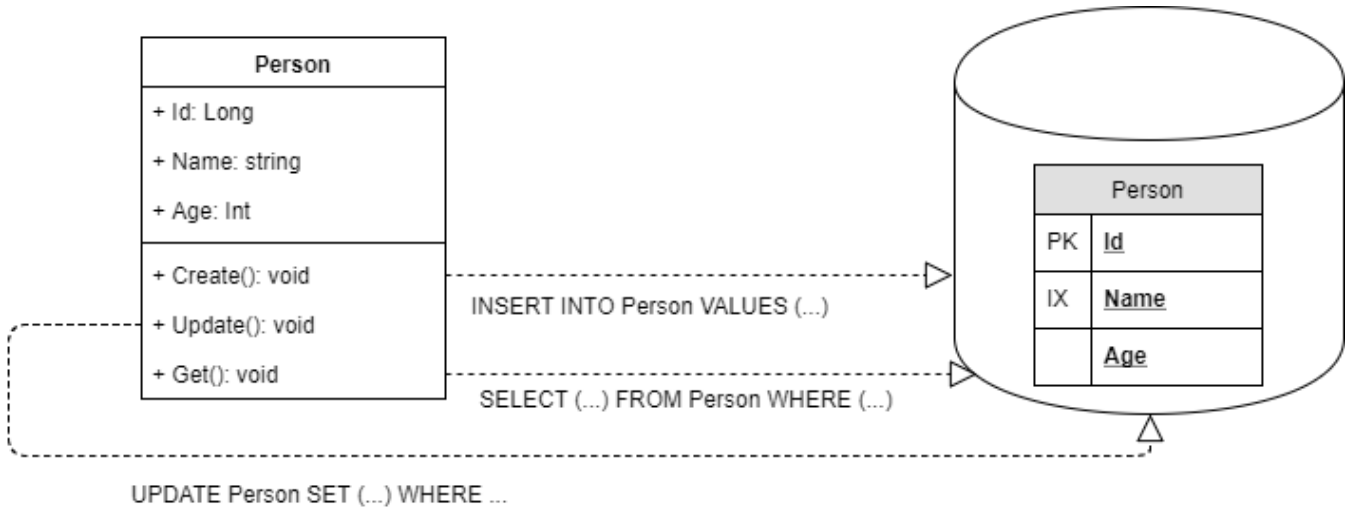

Рис. 2. Шаблон ActiveRecord

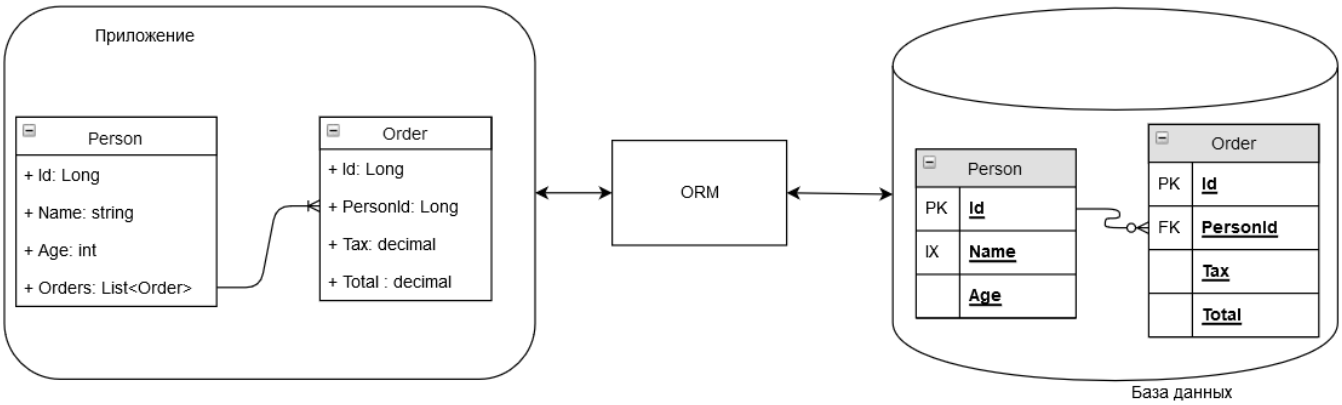

Рис. 3. Технология ORM

безопасности данных приложения, например, SQL-инъекции. (SQL Injection)

В противном случае используется альтернатива — технология объектно-реляционного сопоставления (ORM), которая позволяет отделить и вынести всю логику, специфичную для генерации запросов SQL в отдельные классы, а в самих сущностях оставить лишь данные и бизнес логику, специфичную для приложения. Графическая модель ORMпредставлена на Рисунке 3.

Также, технология ORM облегчает написание операторов соединения (JOIN), группировки (GROUP BY, HAVING), фильтрации (WHERE) и сортировку (ORDER BY) предоставляя удобный и объектно ориентированный способ составления динамических запросов [3, 4]. Некоторые реализации ORM поддерживают систему кеширования и встроенную защиту от инъекций SQL кода, направленных на порчу или неконтролируемый доступ к данным, путем экранирования параметров запроса при его составлении.

С точки зрения реализации, технологии ORM основана на совместном использовании двух шаблонов проектирования [2]: Repository (Репозиторий) и UnitOfWork (Единица работы). «Репозиторий» служит посредником (Mediator), задача которого — отделить логику приложения от логики доступа к данным, предоставляя интерфейс, подобный обычной коллекции в памяти, для вызывающего кода.

Проблема, которую решает шаблон «Единица работ» — это контроль состояния объектов и всех изменений над объектами, полученных путем чтения данных с БД. Такой контроль позволяет объединять отправляемые команды SQL в пакеты команд и исполнять их

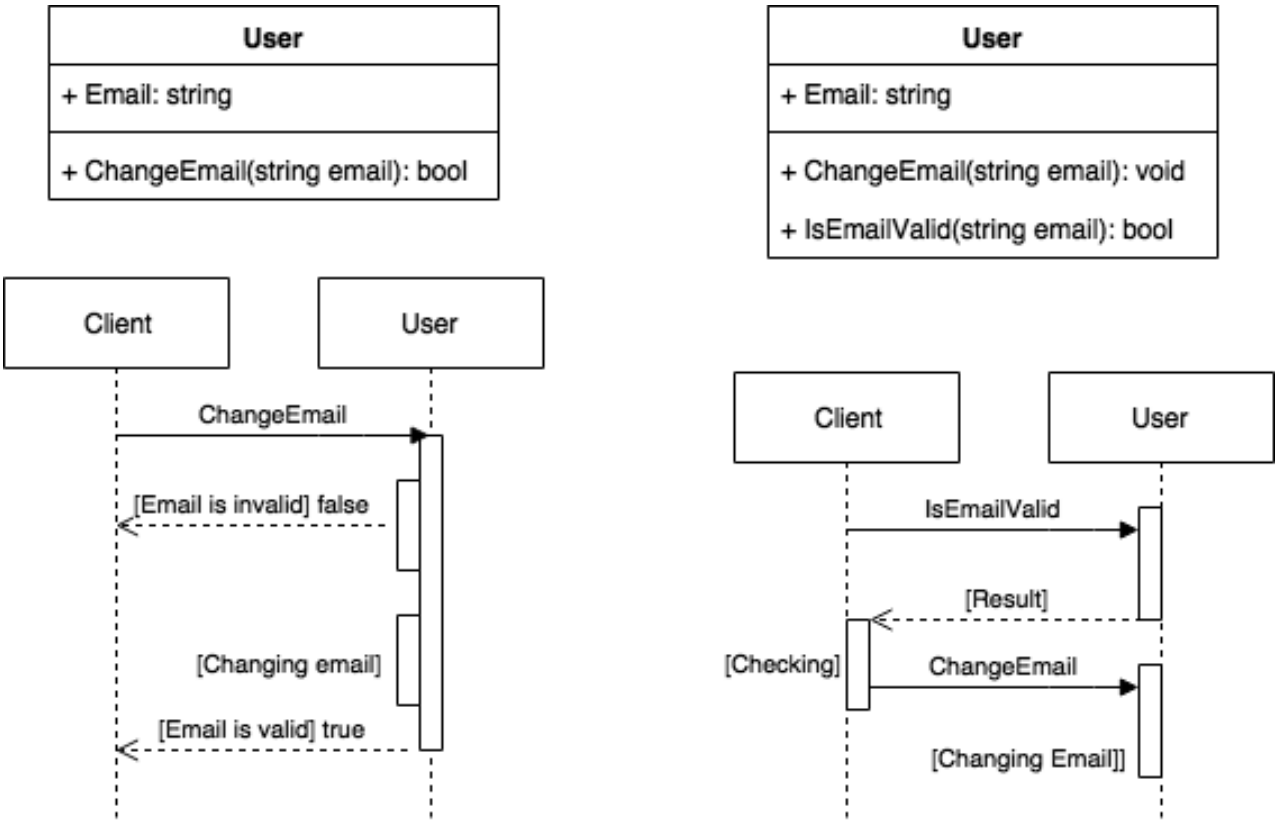

Рис. 4. CQS

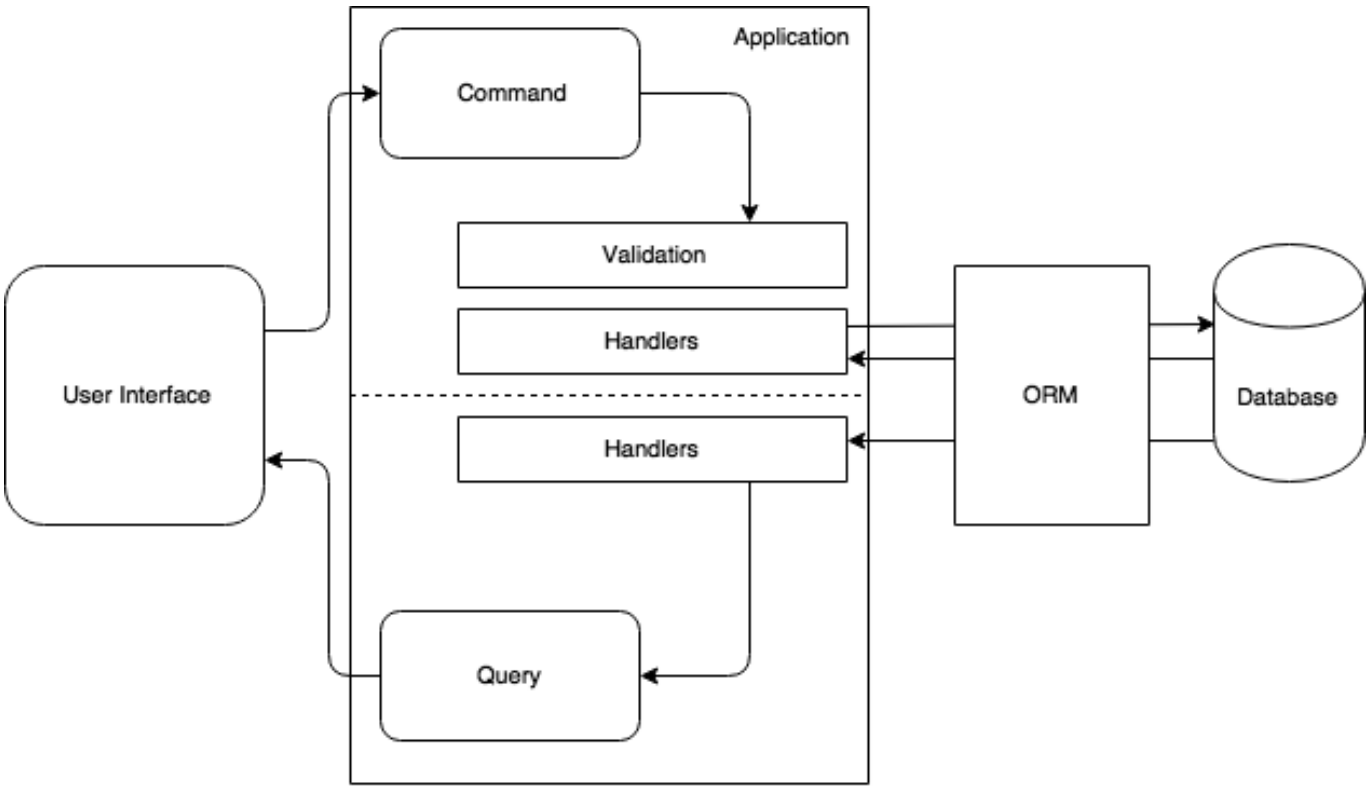

Рис. 5. CQRS

```
public interface ICommandHandler<in TCommand>
{
    Task HandleAsync(TCommand command);
}

public interface IQueryHandler<in TQuery, TResult>
{
    Task<TResult> HandleAsync(TQuery query);
}
```

Рис. 6

```
public sealed class LoggingDecorator<TCommand> : ICommandHandler<TCommand>
{
    public LoggingDecorator(ICommandHandler<TCommand> decorated, ILogger logger)
    {
        _decorated = decorated;
        _logger = logger;
    }

    public async Task HandleAsync(TCommand command)
    {
        _logger.Info($"Execution of {command} started");
        try
        {
            await _decorated.HandleAsync(command);
            _logger.Info($"Execution of {command} comleted successfully");
        }
        catch (Exception exception)
        {
            _logger.Error($"Execution of {command} failed");
            throw;
        }
    }

    private readonly ILogger _logger;
    private readonly ICommandHandler<TCommand> _decorated;
}
```

Рис. 7

на сервере баз данных под определенным уровнем изоляции, гарантируя консистентность данных.

Архитектура CQRS

Рассмотрение данной архитектуры можно начать с принципа CQS (Command-Query-Separation), основная идея которого заключается в том, что в объекте методы могут быть двух типов:

1. Команды: изменяют состояние объекта, не возвращая результатов.
2. Запросы: возвращают результат, не изменяя состояние объекта. Другими словами, запросы никоим образом не влияют на состояние объекта, не модифицируют его явно или неявно.

На рисунке 4 показана схема принципа CQS.

Такое разделение методов класса облегчает поддержку и сопровождение кода, позволяя выносить все проверки состояния и покрывать их автоматическими тестами.

В такой архитектуре явно разделяются модель исполнения запросов и модель исполнения команд. С каждым действием системы связан класс (класс команды), содер-

```
public interface ICommandFactory
{
    ICommandHandler<TCommand> Create<TCommand>();
}

public sealed class Dispatcher
{
    public Dispatcher(ICommandFactory factory)
    {
        _factory = factory;
    }

    public Task Execute<TCommand>(IPrincipal principal, TCommand command)
    {
        this.ValidateUserPrincipal(principal);
        this.ValidateInput(command);

        var handler = _factory.Create<TCommand>();
        return handler.HandleAsync(command);
    }

    private void ValidateInput<TCommand>(TCommand command){...}

    private void ValidateUserPrincipal(IPrincipal principal){...}

    private readonly ICommandFactory _factory;
}
```

Рис. 8

жащий все необходимые данные для выполнения этого действия. С каждым классом команды связан класс-обработчик действия. Такое же описание справедливо и для запросов.

В контексте языка программирования C#.NET Framework 4.6.2 данное описание может быть выражено следующими базовыми интерфейсами (рис. 6).

Тогда для осуществления всех действий и запросов создаются классы, реализующие заданные интерфейсы, причем каждый такой класс будет сосредоточен на реализации конкретной задачи, сохраняя принцип единственной ответственности (SRP).

Весь скользящий функционал, связанный с логированием, обработкой ошибок, измерением производительности и т.д. реализуется с помощью дополнительного шаблона проектирования «Декоратор» (Decorator) [2] (рис. 7).

Задачи конструирования объектов реализации, проверка прав доступа и валидирование входных параметров реализуются с помощью отдельных шаблонов проектирования — «Фасад» (Facade) и «Фабричный метод» (FactoryMethod), рис. 8.

Применение продемонстрированных техник обеспечивают надежную, расширяемую и поддерживаемую архитектуру.

Отдельно необходимо отметить возможности дальнейшего масштабирования производительности. Существует несколько техник:

1. Оптимизация структуры базы данных: добавление индексов, материализованных представлений на ускоренный доступ чтения данных.
2. Использование более легковесных конструкций для взаимодействия с базой данных
3. Разделение единого SQL хранилища: использование технологий кэширования и NoSQL.

В первом случае масштабирование нагрузки производится за счет базы данных, и код приложения не будет затронут. В остальных случаях гибкая архитектура позволяет заменять отдельные команды и запросы их оп-

тимизированными аналогами, не совершая существенные изменения в архитектуре веб-приложения.

Заключение

В работе предложен вариант взаимодействия на основе архитектуры REST между компонентами программной системы. Описана и продемонстрирована методика разработки приложений для взаимодействия с СУБД на основе современных паттернов проектирования. Приведены конкретные рекомендации и примеры кода для реализации гибкого расширяемого программного комплекса. Рассмотренный подход является актуальным в настоящее время и позволяет разрабатывать легко поддерживаемое программное обеспечение.

ЛИТЕРАТУРА

1. Эндрю Троелсон. Язык программирования C# 5.0 и платформа.NET 4.5. Изд. Вильямс. 2015
2. Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Изд. Питер. 2016
3. К. Дж. Дэйт. Введение в системы баз данных. Изд. Вильямс. 2017
4. Ицик Бен-Ган. Microsoft SQL Server 2012. Высокопроизводительный код T-SQL. Оконные функции. Русская Редакция, БХВ-Петербург. 2013

Павлов Виталий Юрьевич (vitaly_pavlov@hotmail.ru), РоманенковАлександрМихайлович (romanaleks@gmail.com).
Журнал «Современная наука: актуальные проблемы теории и практики»

Московский авиационный институт