**4 ТЕХНИЧЕСКОЕ ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА**

Для успешного создания программных средств необходимо обратить внимание на выбор языков программирования, сред разработки и технологий. Удачный выбор инфраструктуры для разработки программных средств позволяет снизить расходы на создание программного средства, а также увеличить качество и производительность труда.

Ниже представлены некоторые факторы, которые повлияли на выбор технологий:

* удобство разработки, позволяющее быстро и качественно разрабатывать программные средства;
* переносимость программного средства между различными персональными компьютерами;
* доступная документация для обращения при возникших затруднениях во время разработки;
* распространённость ОС, для которой разрабатывается программное средство.

Приняв во внимание вышеперечисленные факторы, а также рассмотрев особенности различных технологий, были выбраны следующие технологии:

* операционная система Windows 7;
* язык программирования C# 6.0;
* платформа .Net Framework 4.5;
* среда разработки Visual Studio 2015;
* фреймворк ASP.NET MVC.

Далее представлено подробное описание используемых языков программирования и технологий.

**4.1 Используемые технологии**

**4.1.1** Язык программирования C#

C# – элегантный, типобезопасный объектно-ориентированный язык, предназначенный для разработки разнообразных безопасных и мощных приложений, выполняемых в среде .NET Framework. С помощью языка C# можно создавать обычные приложения Windows, XML-веб-службы, распределённые компоненты, приложения «клиент-сервер», приложения баз данных и т. д. Visual C# предоставляет развитый редактор кода, конструкторы с удобным пользовательским интерфейсом, встроенный отладчик и множество других средств, упрощающих разработку приложений на базе языка C# и .NET Framework [10].

Синтаксис C# очень выразителен, но прост в изучении. Все, кто знаком с языками C, C++ или Java с лёгкостью узнают синтаксис с фигурными скобками, характерный для языка C#. Разработчики, знающие любой из этих языков, как правило, смогут добиться эффективной работы с языком C# за очень короткое время. Синтаксис C# делает проще то, что было сложно в C++, и обеспечивает мощные возможности, такие как типы значений Nullable, перечисления, делегаты, лямбда-выражения и прямой доступ к памяти, чего нет в Java. C# поддерживает универсальные методы и типы, обеспечивая более высокий уровень безопасности и производительности, а также итераторы, позволяющие при реализации коллекций классов определять собственное поведение итерации, которое может легко использоваться в клиентском коде. Выражения LINQ делают строго типизированный запрос очень удобной языковой конструкцией.

Как объектно-ориентированный язык, C# поддерживает понятия инкапсуляции, наследования и полиморфизма. Все переменные и методы, включая метод Main – точку входа приложения – инкапсулируются в определения классов. Класс может наследовать непосредственно из одного родительного класса, но может реализовывать любое число интерфейсов. Для методов, которые переопределяют виртуальные методы в родительском классе, необходимо ключевое слово override, чтобы исключить случайное повторное определение. В языке C# структура похожа на облегченный класс: это тип, распределяемый в стеке, реализующий интерфейсы, но не поддерживающий наследование.

В дополнение к основным описанным объектно-ориентированным принципам, язык C# упрощает разработку компонентов программного обеспечения благодаря нескольким инновационным конструкциям языка, в число которых входят следующие:

* инкапсулированные сигнатуры методов, называемые делегатами, которые поддерживают типобезопасные уведомления о событиях;
* свойства, выступающие в роли методов доступа для закрытых переменных-членов;
* атрибуты с декларативными метаданными о типах во время выполнения;
* встроенные комментарии XML-документации;
* LINQ, предлагающий встроенные возможности запросов в различных источниках данных.

Если потребуется обеспечить взаимодействие с другим программным обеспечением Windows, таким как объекты COM или собственные библиотеки DLL Win32, в языке C# можно использовать процесс, который называется Interop. Процесс Interop позволяет программам на C# выполнять практически любые действия, которые может выполнять исходное приложение на C++. Язык C# поддерживает даже указатели и понятие «небезопасного» кода для тех случаев, когда прямой доступ к памяти имеет крайне важное значение.

Процесс построения C# по сравнению с C и C++ прост и является более гибким, чем в Java. Нет отдельных файлов заголовка, а методы и типы не требуется объявлять в определенном порядке. В исходном файле C# может быть определено любое число классов, структур, интерфейсов и событий.

**4.1.2** Платформа .Net Framework

Программа на языке C# выполняется в среде .NET Framework – интегрированном компоненте Windows, содержащем виртуальную систему выполнения (среда CLR) и унифицированный набор библиотек классов. Среда CLR представляет собой коммерческую реализацию Майкрософт инфраструктуры CLI, международного стандарта, основы сред выполнения и разработки с тесным взаимодействием языков и библиотек [11].

Исходный код, написанный на языке C#, компилируется в промежуточный язык (IL) в соответствии со спецификацией CLI. Код IL и ресурсы, такие как растровые изображения и строки, хранятся на диске в исполняемом файле, называемом сборкой, с расширением EXE или DLL в большинстве случаев. Сборка содержит манифест со сведениями о типах сборки, версии, языке и региональных параметрах, и требованиях безопасности.

При выполнении программы на C# сборка загружается в среду CLR в зависимости от сведений в манифесте. Далее, если требования безопасности соблюдены, среда CLR выполняет JIT-компиляцию для преобразования кода IL в инструкции машинного кода. Среда CLR также предоставляет другие службы, относящиеся к автоматическому сбору мусора, обработке исключений и управлению ресурсами. Код, выполняемый средой CLR, иногда называют «управляемым кодом» в противопоставление «неуправляемому коду», который компилируется в машинный код, предназначенный для определенной системы.

Взаимодействие между языками является ключевой особенностью .NET Framework. Поскольку код на промежуточном языке (IL), создаваемый компилятором C#, соответствует спецификации CTS, код IL на основе C# может взаимодействовать с кодом, создаваемым версиями языков Visual Basic, Visual C++, Visual J# платформы .NET Framework и еще более чем 20 CTS-совместимых языков. В одной сборке может быть несколько модулей, написанных на разных языках платформы .NET Framework, и типы могут ссылаться друг на друга, как если бы они были написаны на одном языке.

Помимо служб времени выполнения, в .NET Framework также имеется обширная библиотека, состоящая из более чем 4000 классов, организованных по пространствам имен, которые обеспечивают разнообразные полезные функции для любых действий, начиная от ввода и вывода файлов для управления строками для разбивки XML, и заканчивая элементами управления Windows Forms. В обычном приложении на языке C# библиотека классов .NET Framework интенсивно используется для «устройства» кода.

**4.1.3** Фреймворк ASP.NET MVC

ASP.NET MVC Framework — фреймворк для создания веб-приложений, который реализует шаблон Model-view-controller.

Платформа ASP.NET MVC базируется на взаимодействии трех компонентов: контроллера, модели и представления. Контроллер принимает запросы, обрабатывает пользовательский ввод, взаимодействует с моделью и представлением и возвращает пользователю результат обработки запроса.

Модель представляет слой, описывающий логику организации данных в приложении. Представление получает данные из контроллера и генерирует элементы пользовательского интерфейса для отображения информации [12].

Для управления разметкой и вставками кода в представлении используется движок представлений. До версии MVC 5 использовались два движка: Web Forms и Razor. Начиная с MVC 5 единственным движком, встроенным по умолчанию, является Razor. Движок WebForms использует файлы .aspx, а Razor — файлы .cshtml и .vbhtml для хранения кода представлений. Основой синтаксиса Razor является знак @, после которого осуществляется переход к коду на языках C#. Также возможно и использование сторонних движков. Файлы представлений не являются стандартными статическими страницами с кодом html, а в процессе генерации контроллером ответа с использованием представлений компилируются в классы, из которых затем генерируется страница html.

При обработке запросов фреймворк ASP.NET MVC опирается на систему маршрутизации, которая сопоставляет все входящие запросы с определенными в системе маршрутами, которые указывают, какой контроллер и метод должен обработать данный запрос. Встроенный маршрут по умолчанию предполагает трехзвенную структуру: контроллер/действие/параметр.

ASP.NET MVC и ASP.NET Web Forms являются двумя родственными технологиями, в основании которых лежит одна платформа ASP.NET. И все же ASP.NET MVC имеет ряд преимуществ перед ASP.NET Web Forms:

* разделение ответственности (отдельная разработка разных компонентов - контроллера, моделей, представлений);
* улучшенная тестируемость;
* соответствие протоколу HTTP (ASP.NET MVC не поддерживает объект ViewState);
* повышенная гибкость и настраиваемость под собственные нужды.

**4.1.4** Библиотека jQuery

jQuery — [библиотека JavaScript](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_JavaScript), фокусирующаяся на взаимодействии [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) и [HTML](https://ru.wikipedia.org/wiki/HTML). Библиотека jQuery помогает легко получать доступ к любому элементу [DOM](https://ru.wikipedia.org/wiki/Document_Object_Model), обращаться к атрибутам и содержимому элементов [DOM](https://ru.wikipedia.org/wiki/Document_Object_Model), манипулировать ими. Также библиотека jQuery предоставляет удобный [API](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) для работы с [AJAX](https://ru.wikipedia.org/wiki/AJAX).

Преимущества jQuery:

* движок кросс-браузерных CSS-селекторов [Sizzle](https://ru.wikipedia.org/w/index.php?title=Sizzle&action=edit&redlink=1" \o "Sizzle (страница отсутствует)), выделившийся в отдельный проект;
* переход по дереву DOM, включая поддержку XPath как плагина;
* события;
* визуальные эффекты;
* AJAX-дополнения;
* JavaScript-плагины.

Точно так же, как [CSS](https://ru.wikipedia.org/wiki/CSS) отделяет визуализацию от структуры HTML, JQuery отделяет поведение от структуры HTML. Например, вместо прямого указания на обработчик события нажатия кнопки, управление передаётся JQuery, которая идентифицирует кнопки и затем преобразует его в обработчик события клика. Такое разделение поведения и структуры также называется принципом [ненавязчивого JavaScript](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%BD%D0%B0%D0%B2%D1%8F%D0%B7%D1%87%D0%B8%D0%B2%D1%8B%D0%B9_JavaScript).

Библиотека jQuery содержит функциональность, полезную для максимально широкого круга задач. Тем не менее, разработчиками библиотеки не ставилась задача совмещения в jQuery функций, которые подошли бы всюду, поскольку это привело бы к большому коду, бо́льшая часть которого не востребована. Поэтому была реализована архитектура компактного универсального ядра библиотеки и плагинов. Это позволяет собрать для ресурса именно ту JavaScript-функциональность, которая на нём была бы востребована.

**4.1.5** Внедрение зависимостей, контейнер зависимостей Ninject

Внедрение зависимости ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Dependency injection, DI) — процесс предоставления внешней зависимости программному компоненту. Является специфичной формой «[инверсии управления](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)» ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) Inversion of control, IoC), когда она применяется к управлению зависимостями. В полном соответствии с [принципом единственной ответственности](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF_%D0%B5%D0%B4%D0%B8%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D0%B1%D1%8F%D0%B7%D0%B0%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D0%B8) объект отдаёт заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

Работа фреймворка, обеспечивающая внедрение зависимости, описывается следующим образом. Приложение, независимо от оформления, исполняется внутри контейнера [IoC](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F" \o "Инверсия управления), предоставляемого фреймворком. Часть объектов в программе по-прежнему создается обычным способом языка программирования, часть создается контейнером на основе предоставленной ему конфигурации.

Условно, если [объекту](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) нужно получить доступ к определенному [сервису](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B5%D1%80%D0%B2%D0%B8%D1%81_(%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC)&action=edit&redlink=1), объект берет на себя ответственность за доступ к этому сервису: он или получает прямую [ссылку](https://ru.wikipedia.org/wiki/%D0%A1%D1%81%D1%8B%D0%BB%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) на местонахождение сервиса, или обращается к известному «сервис-локатору» и запрашивает ссылку на реализацию определенного типа сервиса. Используя же внедрение зависимости, объект просто предоставляет свойство, которое в состоянии хранить ссылку на нужный тип сервиса; и когда объект создается, ссылка на реализацию нужного типа сервиса автоматически вставляется в это свойство, используя средства среды.

Внедрение зависимости более гибко, потому что становится легче создавать альтернативные реализации данного типа сервиса, а потом указывать, какая именно реализация должна быть использована в, например, [конфигурационном файле](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB_%D0%BA%D0%BE%D0%BD%D1%84%D0%B8%D0%B3%D1%83%D1%80%D0%B0%D1%86%D0%B8%D0%B8), без изменений в объектах, которые этот сервис используют. Это особенно полезно в [юнит-тестировании](https://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D1%82-%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), потому что вставить реализацию «[заглушки](https://ru.wikipedia.org/wiki/Mock-%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82)» сервиса в тестируемый объект очень просто.

С другой стороны, излишнее использование внедрения зависимостей может сделать приложения более сложными и трудными в сопровождении: так как для понимания поведения программы программисту необходимо смотреть не только в исходный код, а еще и в конфигурацию, а конфигурация, как правило, невидима для [IDE](https://ru.wikipedia.org/wiki/%D0%A1%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), которые поддерживают анализ ссылок и [рефакторинг](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3), если явно не указана поддержка [фреймворков](https://ru.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA) с внедрениями зависимостей.

Ninject - это легкая платформа, упрощающая внедрения зависимостей для приложений .NET. Позволяющая разделить приложение на множество слабосвязанных фрагментов, а затем внедрить их в гибкой манере. Использование Ninject для поддержки архитектуры программного обеспечения позволит повторно использовать модули, упростить тестирование и модифицировать.

Основные преимущества Ninject:

* Сосредоточенность на выполняемых функциях. Слишком много существующих контейнеров зависимостей приносят в жертву удобство использования для функций, которые не часто необходимы. Каждый раз, когда функция добавляется в Ninject, ее необходимость сравнивается с той сложностью, которую она добавляет к ежедневному использованию. У Ninject есть много продвинутых функций, но их понимание не требуется для использования основных функций.
* Быстрота. Вместо того, чтобы полагаться на механизм отражение для внедрения зависимостей, Ninject использует генерацию кода в CLR. Это может привести к резкому (8-50 раз) улучшению производительности во многих ситуациях.
* Точность. Вместо того, чтобы полагаться на файлы сопоставления XML и идентификаторы строк для подключения компонентов, Ninject обеспечивает надежный предметно-ориентированный язык. Это означает, что Ninject использует возможность языка C# такую как безопасность типов и возможности IDE такие как IntelliSense и автоматическое дописывание кода.
* Скрытность. Ninject не вторгается в код. Можно легко изолировать зависимости в одной сборки в проекте.

**4.1.6** Entity Framework

Entity Framework представляет специальную объектно-ориентированную технологию на базе фреймворка .NET для работы с данными. Если традиционные средства ADO.NET позволяют создавать подключения, команды и прочие объекты для взаимодействия с базами данных, то Entity Framework представляет собой более высокий уровень абстракции, который позволяет абстрагироваться от самой базы данных и работать с данными независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне, который нам предлагает Entity Framework, мы уже работает с объектами [13].

Центральной концепцией Entity Framework является понятие сущности или entity. Сущность представляет набор данных, ассоциированных с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их наборами. Любая сущность, как и любой объект из реального мира, обладает рядом свойств. Свойства необязательно представляют простые данные типа int, могут представлять более комплексные структуры данных. И у каждой сущности может быть одно или несколько свойств, которые будут отличать эту сущность от других и будут уникально определять эту сущность. Подобные свойства называют ключами.

При этом сущности могут быть связаны ассоциативной связью один-ко-многим, один-ко-одному и многие-ко-многим, подобно тому, как в реальной базе данных происходит связь через внешние ключи. Отличительной чертой Entity Framework является использование запросов LINQ для выборки данных из БД. С помощью LINQ можно не только извлекать определенные строки, хранящие объекты, из базы данных, но и получать объекты, связанные различными ассоциативными связями.

Другим ключевым понятием является Entity Data Model. Эта модель сопоставляет классы сущностей с реальными таблицами в БД. Entity Data Model состоит из трех уровней: концептуального, уровень хранилища и уровень сопоставления (маппинга).

На концептуальном уровне происходит определение классов сущностей, используемых в приложении.

Уровень хранилища определяет таблицы, столбцы, отношения между таблицами и типы данных, с которыми сопоставляется используемая база данных.

Уровень сопоставлен служит посредником между предыдущими двумя, определяя сопоставление между свойствами класса сущности и столбцами таблиц. Таким образом, появляется возможность через классы, определенные в приложении, взаимодействовать с таблицами из базы данных.

Способы взаимодействия с БД Entity Framework предполагает три возможных способа взаимодействия:

* Database first. Entity Framework создает набор классов, которые отражают модель конкретной базы данных;
* Model first. Разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере;
* Code first. Разработчик создает класс модели данных, которые будут храниться в базе данных, а затем Entity Framework по этой модели генерирует базу данных и ее таблицы.

**4.1.7** Среда разработки Visual Studio

Visual Studio – интегрированная среда разработки ПО от компании Microsoft. Visual Studio используется для разработки консольных приложений, приложений с графическим интерфейсом, веб-сайтов, веб-приложений, программ с поддержкой Windows Forms, а также для работы с системой построения клиентских приложений – Windows Presentation Foundation (WPF). Можно разрабатывать веб-сервисы и в родном, и в управляемом кодах для всех платформ, поддерживаемых .NET Framework, Microsoft Windows, Windows Mobile, Microsoft Silverlight и .NET Compact Framework [14].

Visual Studio состоит из редактора кода, поддерживающего IntelliSense и рефакторинг кода. Встроенный отладчик можно использовать как для отладки уровня исходного кода, так и в качестве отладчика машинного уровня. Другие встроенные инструменты включают:

– редактор форм для разработки приложений GUI;

– веб-редактор;

– редактор классов;

– редактор схемы баз данных.

Visual Studio даёт возможность устанавливать плагины для расширения функциональных возможностей почти на любом уровне, в том числе для поддержки систем контроля исходного кода и новых инструментов для редактирования на предметно-ориентированных языках программирования, а также инструментов для других аспектов жизненного цикла разработки программного обеспечения.

Visual Studio предоставляет языковые сервисы, которые позволяют редактору кода и отладчику поддерживать практически любой язык программирования. Встроенные языки программирования включают:

– C/C++;

– VB.NET;

– C#;

– F#.

Также Visual Studio поддерживает XML/XSLT, HTML/XHTML, JavaScript и CSS.

* 1. **Описание проектной структуры**

**4.2.1** Взаимодействие проектов

В разделе 3.1 рассматривалась архитектура программного средства, однако эта архитектура абстрагирована от конкретного фреймворка на котором и производится разработка.

При использовании ASP.NET MVC как правило, приложение в целом будет представлять решение (англ. Solution) в Visual Studio, а отдельные уровни - проекты. В то же время неверно полагать, что уровень обязательно должен соответствовать отдельному проекту. При необходимости можно раздробить один уровень на несколько проектов, главное, чтобы его функционал представлял единое логическое звено [9].

На рисунке 4.1 представлена структура программного средства в рамках решения в Visual Studio. Из него видно, что структура разделена на три слоя:

* слой представления – проект MVCNBlog;
* слой бизнес-логики – проекты BLL.Interface и BLL;
* слой доступа к данным – проекты DAL.Interface, DAL и ORM.

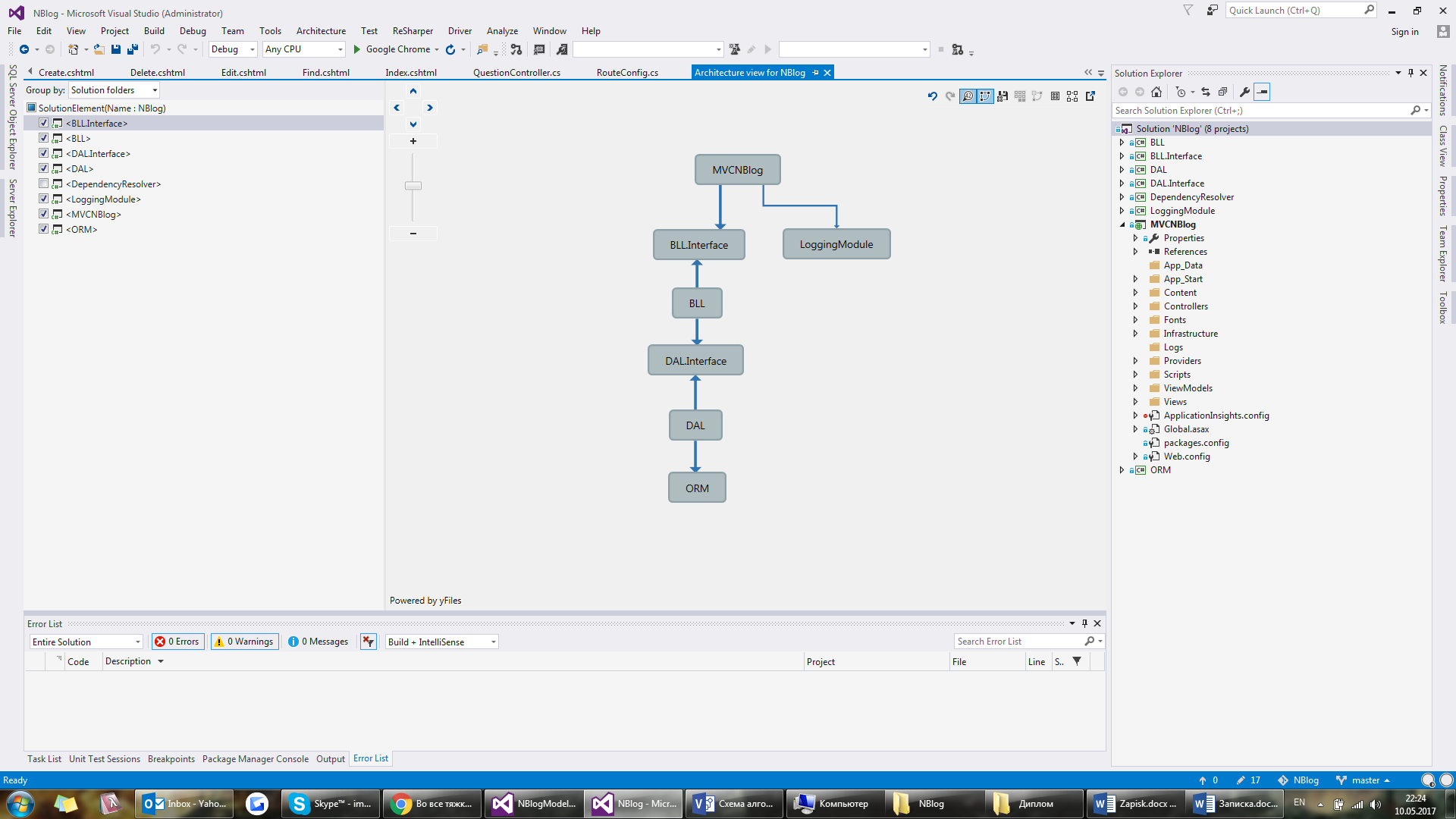


Рисунок 4.1 - Структура решения в Visual Studio

Каждый слой связан со следующим с помощью проекта с припиской Interface, это сделано для того, чтобы отделить логику того или иного слоя от интерфейсов. Это позволяет, в случае необходимости заменить логику слоя, для этого необходимо заменить проект, который содержит логику (BLL, DAL), на новый реализующий интерфейсы этого слоя [12].

Так же слой доступа к данным содержит проект под названием ORM, который выполняет роль связующего звена с базой данных. Для доступа к разным базам данных используются разные библиотеки, написанные разными фирмами, поэтому вынесение логики, связанной с доступом к специфической базе может быть полезно при последующей смене базы данных.

Проектом в структуре, который на прямую не связан ни с одним слоем является проект LogginModule, который используется для логирования исключительных ситуаций. Логирование может быть использовано на любом из слоёв. При разработки данного программного средства было принято решения логировать исключительные ситуации на слое представления, куда в любом случае попадут оповещения о ошибках с нижних слоёв.

**4.2.2** Структура проекта MVCNBlog

На рисунке 4.2 представлена файловая структура проекта MVCNBlog который входит в слой представления и по сути является пользовательским интерфейсом [7]. Проект состоит из следующих частей:

* App\_Data. В данной папке хранятся все необходимые файлы и ресурсы, такие как, базы данных, используемые приложением. После развертывания приложения только непосредственно приложение может работать с этой папкой, доступ же простых пользователей в эту папку запрещен.
* Файл Global.asax и папка App\_Start. Папка App\_Start включает весь функционал конфигурации приложения, который в предыдущих версиях содержался в файле Global.asax, а теперь перенесен в набор статичных классов, вызываемых в Global.asax. Эти статичные классы содержат некоторую логику инициализации приложения, выполняющуюся при запуске.
* Файл Web.config. Файл конфигурации приложения, который находится в корневой папке приложения.
* Content. В данной папке хранятся вспомогательные файлы, которые не включают код на c# или javascript, и которые развертываются вместе с приложением. В частности, здесь могут размещаться файлы стилей css. Так, в этой папке находится файл Site.css, который содержит стили приложения, а также папка с темами.
* Controllers. Содержит контроллеры - классы, отвечающие за работу приложения.
* Infrastructure. В данной папке содержится основная логика слоя представления такая как: валидация, маппинг.
* Logs. В данной папке содержатся логи, название каждого лога соответствует дате его создание, а состоит он из записей о исключительных ситуациях.
* Fonts. В данной папке содержатся шрифты и иконки, используемые в пользовательском интерфейсе.
* Providers. В данной папке содержатся классы, предоставляющие пользователям права для пользования порталом.
* Scripts. В данной папке содержатся скрипты на JavaScript, используемые в приложении. По умолчанию эта папка уже содержат файлы библиотеки jQuery.
* ViewModels. Содержит модели, используемые слоем представления.
* Views. Здесь размещаются представления. Представления группированы по папкам, каждая из которых соответствует одному контроллеру. После получения и обработки запроса контроллер, отправляет одно из этих представлений, заполненных некоторыми данными, клиенту. Кроме того, имеется папка общих для контроллеров представлений - папка Shared.

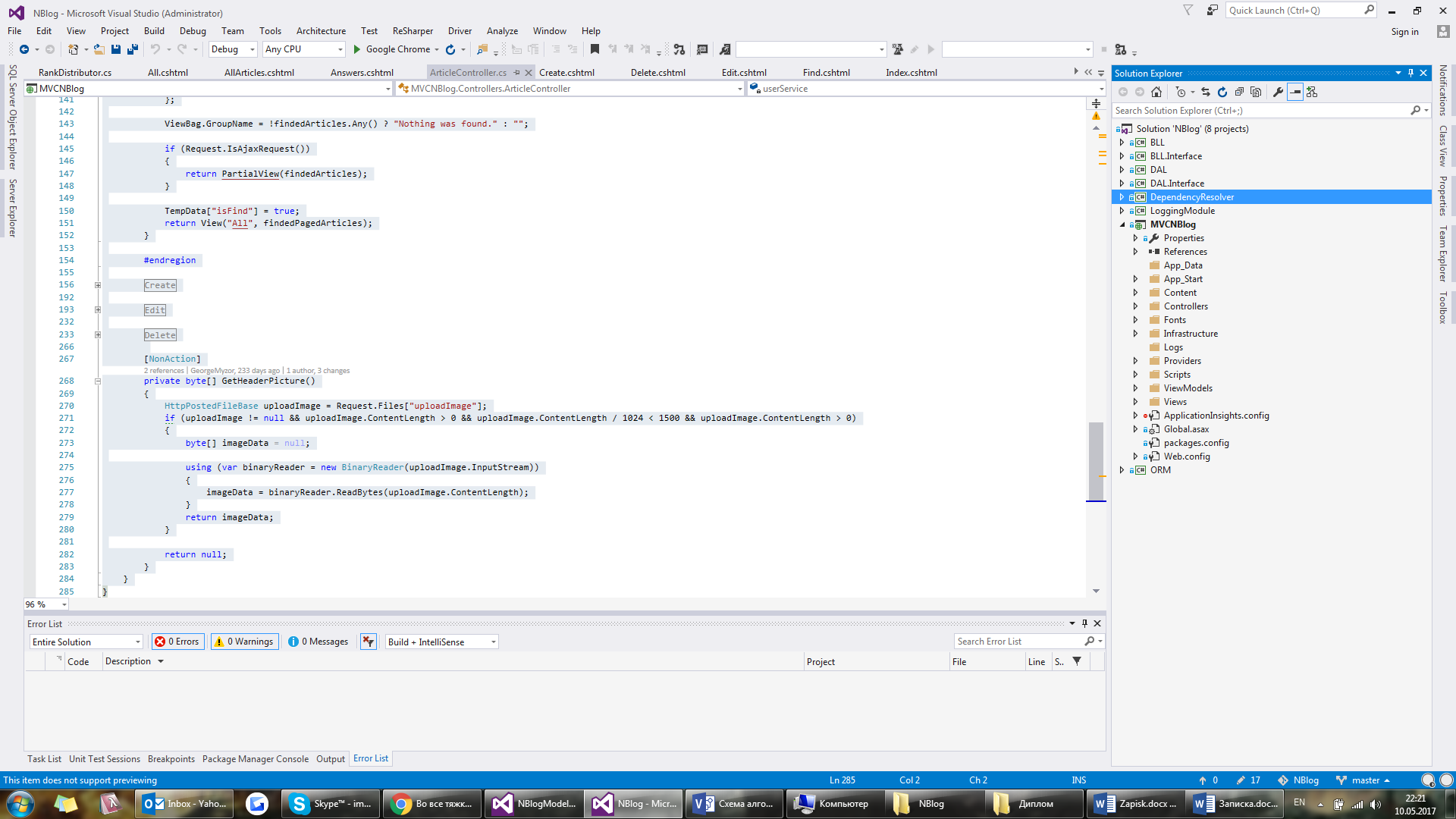


Рисунок 4.2 - Файловая структура проекта MVCNBlog

**3.1 Уровень доступа к базе данных**

В данном подразделе приводится описание шаблонов доступа к данным и их реализации. Уровень доступа к базе данных отвечает за чтение, создание, редактирование, удаление информации. Реализован с помощью MySQL с применением объектно-реляционного отображателя Entity Framework.

**3.1.1 Традиционные объекты CLR**

*Традиционный объект CLR* (Plain Old CLR Object – POCO) – это класс .NET, который используется для представления класса сущности (модели). Этот класс фокусируется на ключевых атрибутах (свойства) и поведении (методы) сущности и связанных с ней сущностей, не требуя никакого специфичного для базы данных кода [5].

Основная цель использования классов POCO заключается в проектировании бизнес-модели проекта с *игнорированием постоянства* (persistence ignorance – PI). Такой подход к проектированию позволяет бизнес-модели проекта развиваться независимо от модели доступа к данным. Поскольку бизнес-модель не содержит кода доступа к данным, облегчается тестирование модели в изоляции, и лежащие в основе данные могут легко быть заменены в случае изменения бизнес-требований.

**3.1.2 Использование шаблона Repository**

*Шаблон Repository (Репозиторий)* – это шаблон доступа к данным, который способствует слабо связанному подходу в отношении доступа к данным. Вместо помещения логики доступа к данным на уровень выше, ответственность за постоянство бизнес-модели проекта возлагается на отдельный класс или набор классов, называемых *репозиторием*.

За счет применения этого шаблона изолируется уровень доступа к данным от остальной части проекта.

Для проектирования репозитория существуют разные подходы, которые кратко описаны ниже:

1. Один на бизнес-модель. Наиболее прямолинейный способ заключается в создании репозитория для каждого класса бизнес-модели. Хотя этот подход и прост, он может приводить к проблемам, таким как дублирование кода или сложность, когда множеству репозиториев понадобится взаимодействовать друг с другом.
2. Использование агрегированного корня. *Агрегированный корень* – это класс, который может существовать сам по себе и являться ответственным за управление ассоциациями с другими связанными классами. Например, в проекте электронной коммерции может существовать класс OrderRepository, который будет обрабатывать создание заказа и связанных с ним элементов в заказе.
3. Обобщенный репозиторий. Вместо создания специфических классов репозитория можно воспользоваться обобщениями .NET для построения общего репозитория, который будет применяться множеством приложений.

Ниже представлена базовая структура обобщенного класса репозитория:

public class BaseRepository<TEntity> where TEntity : Entity

{

public BaseRepository()

{

}

public void Add(TEntity entity)

{

}

public IQueryable<TEntity> Read(TEntity entity)

{

}

public TEntity ReadById(object id)

{

}

public void Update(TEntity entity)

{

}

public void Delete(TEntity entity)

{

}

}

Сервисы из уровня бизнес-логики взаимодействуют с репозиториями для загрузки и сохранения бизнес-модели проекта. За счет использования технологии внедрения зависимостей (DI) репозитории могут внедряться через конструктор сервиса. На рисунке 3.1 изображено отношение между репозиторием и контекстом данных EntityFramework, при котором сервисы взаимодействуют с репозиторием, а не напрямую с EntityFramework.

В следующем примере показано, как репозиторий внедряется в сервис с применением технологии внедрения зависимостей, и демонстрируется использование внедренного репозитория для извлечения списка фильмов.



Рисунок 3.1 – Организация взаимодействия при наличии репозитория

Применение внедрения зависимостей упрощает тестирование сервиса за счет имитации передаваемого репозитория:

public class FilmService : IFilmService

{

private readonly IRepository \_repository;

public FilmService(IRepository repository)

{

\_repository = repository;

}

public IEnumerable<FilmDomainModel> Read()

{

IQueryable<Film> films = \_repository Read();

IEnumerable<FilmDomainModel> filmsDomain = Mapper.Map<IEnumerable<Film>, IEnumerable<FilmDomainModel>>(films);

return filmsDomain;

}

}

**3.1.3 Выбор подхода доступа к данным**

В Microsoft признают, что единый подход для доступа к данным не работает. Существует подход, ориентированный на данные, сосредоточенный на проектировании базы данных первой с последующей генерацией для проекта бизнес-модели, во многом управляемой структурой базы данных. В другом подходе используются классы POCO для определения структуры бизнес-модели и либо генерации базы данных из классов POCO, либо привязки модели к существующей базе данных.

В результате ADO.NET Entity Framework позволяет осуществлять выбор между тремя стилями:

1. Database First (Сначала база данных). Для тех, кто предпочитает применять более ориентированный на данные подход или начинать с существующей базы данных, Entity Framework предлагает возможность генерировать бизнес-модель на основе таблиц и их строк в реляционной базе данных. Инфраструктура Entity Framework использует специальный конфигурационный файл (с расширением .edmx) для хранения информации о схеме базы данных, концептуальной модели данных и сведений об отображении между ними.
2. Model First (Сначала модель). Если существующей базы данных нет, Entity Framework предлагает визуальный конструктор, который может применяться для создания концептуальной модели данных. Как и в случае подхода Database First, инфраструктура Entity Framework использует файл схемы для хранения информации, связанной с отображением модели на схему базы данных. После того как модель создана, визуальный конструктор EF может сгенерировать схему базы данных, посредством которой можно создать саму базы данных.
3. Code First (Сначала код). Разработчики, желающие применять подход с большим игнорированием постоянства, могут создавать бизнес-модель прямо в коде. Инфраструктура Entity Framework предоставляет специальный API-интерфейс отображения и поддерживает набор соглашений, обеспечивающих работу этого подхода. При подходе Code First инфраструктура Entity Framework не использует внешний конфигурационный файл для хранения схемы базы данных, поскольку API-интерфейс отображения применяет эти соглашения для генерации схемы базы данных динамически во время выполнения.

**3.1.4 Использование подхода Entity Framework Code First**

Движущей силой подхода Code First является возможность применения классов POCO. В рамках подходов Database First и Model First классы модели, сгенерированные EF, унаследованы от базового класса EntityObject, который предоставляет необходимое оснащение для отображения класса на лежащую в основе схему базы данных. Поскольку подходы Database First и Model First требуют наследования классов хранения от класса EntityObject, они не поддерживают игнорирование постоянства.

Вместо использования для отображения базового сущностного класса в подходе Code First установлен набор соглашений для отображения классов POCO:

* имена таблиц определяются с применением формы множественного числа имени сущности класса;
* имена столбцов выводятся из имен свойств;
* первичные ключи основываются на свойствах с именами ID и имяКлассаID.
* стандартная строка соединения соответствует имени класса DataContext.

Инфраструктура Entity Framework включает множество атрибутов аннотирования данных, которые можно использовать для управления обработкой отображаемых сущностей (см. таблицу 3.1).

Таблица 3.1 – Аннотации данных Code First

|  |  |
| --- | --- |
| Аннотация | Описание |
| Column | Имя столбца базы данных, порядковый номер и тип данных для отображения свойств |
| ComplexType | Используется на классах, которые не содержат ключей и не могут управляться инфраструктурой Entity Framework. Обычно применяется для управления скалярными свойствами в связанной сущности |
| DatabaseGenerated | Используется для пометки свойства, которое должно быть сгенерировано базой данных |
| ForeignKey | Используется для идентификации связанной сущности; представляет ограничение внешнего ключа, устанавливаемое между столбцами |
| InverseProperty | Используется для идентификации свойства, которое представляет другой конец отношения |
| Key | Одно или более свойств, применяемых для уникальной идентификации сущности |
| MaxLength | Максимальная длина свойства (столбца) |
| MinLength | Минимальная длина свойства (столбца) |
| NotMapped | Помечает свойство, которое не будет отображаться инфраструктурой Entity Framework |
| Required | Помечает свойство как являющееся обязательным (не допускающим null) |
| StringLength | Определяет минимальную и максимальную длину поля |
| Table | Используется для определения имени таблицы, применяемого для сущности |

Хотя соглашения направлены на улучшение продуктивности разработки, в Entity Framework допускается наличие ситуаций, когда необходимо нарушить одно и более применяемых соглашений, и для этого предлагается API-интерфейс, который позволяет разработчикам обходить существующие соглашения.

Ниже приведен пример сущностного класса, сконфигурированного с использованием атрибута Key, то есть аннотации данных, которая переопределяет стандартное отображение первичного ключа:

public class Entity

{

[Key]

public long Id { get; set; }

}

Подход доступа к данным ADO.NET Entity Framework Code First предполагает создание класса контекста доступа к данным, который унаследован от DbContext. Этот класс должен содержать свойства для каждой сущности в модели предметной области. Специальный класс контекста доступа может переопределять методы базового класса контекста для поддержки любой специализированной логики запросов и сохранения данных, а также реализовывать любую необходимую логику для отображения сущностей.

Ниже приведен контекст данных Entity Framework Code First, который содержит сущности, соответствующие таблицы базы данных. После определения класса контекста данных с помощью запроса LINQ можно совершать различные операции над любым списком сущностей:

public class ApplicationDbContext : DbContext

{

public DbSet<Cinema> Cinemas { get; set; }

public DbSet<Film> Films { get; set; }

public DbSet<Review> Reviews { get; set; }

public DbSet<Seance> Seances { get; set; }

public DbSet<Ticket> Tickets { get; set; }

public DbSet<PaymentMethod> PaymentMethods { get;set; }

public DbSet<Seat> Seats { get; set; }

public DbSet<ReservedSeat> ReservedSeats { get; set; }

public DbSet<Hall> Halls { get; set; }

public ApplicationDbContext() {}

public static ApplicationDbContext Create()

{

return new ApplicationDbContext();

}

}

Вместо работы напрямую с контекстом данных Entity Framework используется шаблон Repository. Это устанавливает уровень абстракции между сервисами уровня бизнес-логики проекта и инфраструктурой Entity Framework. Обеспечение реализации классом репозитория интерфейса IRepository позволяет применять контейнер IoC для внедрения репозитория в клиентский код (например, сервис).

Ниже приведен пример репозитория, который настроен для абстрагирования от контекста данных Entity Framework:

public class BaseRepository<TEntity> : IRepository<TEntity> where TEntity : Entity

{

private readonly ApplicationDbContext \_context;

public BaseRepository(ApplicationDbContext context)

{

\_context = context;

}

}

**3.1.5 Описание моделей уровня доступа к данным**

Центральной концепцией Entity Framework является понятие сущности или entity. Сущность представляет набор данных, ассоциированных с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их наборами. Эти объекты являются проекциями таблиц базы данных (см. чертеж ГУИР.400201.035 РР.2).

Любая сущность, как и любой объект из реального мира, обладает рядом свойств. Например, если сущность описывает кинотеатр, то можно выделить такие свойства, как название, адрес, номер телефона, путь к файлу с изображением кинотеатра и другие. Свойства необязательно представляют простые данные типа int, но и могут представлять более комплексные структуры данных. И у каждой сущности может быть одно или несколько свойств, которые будут отличать эту сущность от других и будут уникально определять эту сущность.

**3.1.5.1 Класс Entity**

Класс Entity является базовым классом для моделей уровня доступа к данным.

Свойства класса Entity:

* public long Id – первичный ключ для уникальной идентификации сущностей, которые наследуются от данного класса.

**3.1.5.2 Класс ApplicationUser**

Класс ApplicationUser является сущностью, которая соответствует таблице ApplicationUser базы данных, и реализует интерфейс IdentityUser.

Свойства класса ApplicationUser:

* public string FirstName – хранит имя пользователя, помечено атрибутами Required и StringLength;
* public string LastName – хранит фамилию пользователя, помечено атрибутами Required и StringLength;
* public bool IsBanned – определяет заблокированный статус пользователя;
* public virtual ICollection<Ticket> Tickets – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Ticket;
* public virtual ICollection<Review> Reviews – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Review;
* public virtual ICollection<ReservedSeat> ReservedSeats – навигационное свойство, обозначающее отношение между данной сущностью и сущностью ReservedSeats.

Методы класса ApplicationUser:

* public async Task<ClaimsIdentity> GenerateUserIdentityAsync(UserManager<ApplicationUser> manager) – асинхронный метод, создающий объект ClaimsIdentity.

**3.1.5.3 Класс Cinema**

Класс Cinema является сущностью, которая соответствует таблице Cinema базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса Cinema:

* public string Name – хранит название кинотеатра, помечено атрибутами Required и StringLength;
* public string ImagePath – хранит относительный путь к файлу изображения кинотеатра, помечено атрибутами Required и StringLength;
* public string Address – хранит адрес кинотеатра, помечено атрибутами Required и StringLength;
* public string PhoneNumber – хранит номер телефона кинотеатра, помечено атрибутами Required и StringLength;
* public virtual ICollection<Seance> Seances – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Seance;
* public virtual ICollection<Hall> Halls – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Hall.

**3.1.5.4 Класс Film**

Класс Film является сущностью, которая соответствует таблице Film базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса Film:

* public string Title – хранит название фильма, помечено атрибутами Required и StringLength;
* public string Description – хранит описание фильма, помечено атрибутами Required и StringLength;
* public string ImagePath – хранит относительный путь к файлу изображения фильма;
* public int Duration – хранит длительность фильма, помечено атрибутами Required и Range;
* public virtual ICollection<Review> Reviews – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Review;
* public virtual ICollection<Seance> Seances – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Seance.

**3.1.5.5 Класс Hall**

Класс Hall является сущностью, которая соответствует таблице Hall базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса Hall:

* public string Name – хранит название зала кинотеатра, помечено атрибутами Required и StringLength;
* public int RowAmount – хранит количество рядов в зале, помечено атрибутом Range;
* public int ColumnAmount – хранит количество мест в ряду зала, помечено атрибутом Range;
* public virtual Cinema Cinema – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Cinema;
* public long CinemaId – внешний ключ, связывающий данную сущность и сущность Cinema;
* public virtual ICollection<Seat> Seats – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Seat;
* public virtual ICollection<Seance> Seances – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Seance.

**3.1.5.6 Класс PaymentMethod**

Класс PaymentMethod является сущностью, которая соответствует таблице PaymentMethod базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса PaymentMethod:

* public string Name – хранит название способа оплаты, помечено атрибутами Required и StringLength;
* public virtual ICollection<Ticket> Tickets – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Ticket.

**3.1.5.7 Класс Seat**

Класс Seat является сущностью, которая соответствует таблице Seat базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса Seat:

* public int RowNumber – хранит номер ряда, в котором находится конкретное место в зале;
* public int ColumnNumber – хранит номер места в ряду зала;
* public virtual Hall Hall – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Hall;
* public long HallId – внешний ключ, связывающий данную сущность и сущность Hall.

**3.1.5.8 Класс Review**

Класс Review является сущностью, которая соответствует таблице Review базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса Review:

* public int Rate – хранит значение рейтинга данного конкретного отзыва о фильме, помечено атрибутом Range;
* public string Description – хранит описание фильма, помечено атрибутами Required и StringLength;
* public DateTime ReviewDate – хранит дату создания отзыва о фильме;
* public virtual ApplicationUser ApplicationUser – навигационное свойство, обозначающее отношение между данной сущностью и сущностью ApplicationUser;
* public string ApplicationUserId – внешний ключ, связывающий данную сущность и сущность ApplicationUser;
* public virtual Film Film – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Film;
* public long FilmId – внешний ключ, связывающий данную сущность и сущность Film.

**3.1.5.9 Класс ReservedSeat**

Класс ReservedSeat является сущностью, которая соответствует таблице ReservedSeat базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса ReservedSeat:

* public int RowNumber – хранит номер ряда, в котором находится заказанное место в зале;
* public int ColumnNumber – хранит номер места в ряду зала;
* public DateTime ReservationTime – хранит дату заказа места в зале;
* public bool IsSold – определяет заказанный статус места в зале;
* public virtual Seance Seance – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Seance;
* public long SeanceId – внешний ключ, связывающий данную сущность и сущность Seance;
* public string ApplicationUserId – внешний ключ, связывающий данную сущность и сущность ApplicationUser.

**3.1.5.10 Класс Seance**

Класс Seance является сущностью, которая соответствует таблице Seance базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса Seance:

* public DateTime Date – хранит дату проведения сеанса;
* public DateTime Time – хранит время проведения сеанса;
* public decimal Price – хранит цену сеанса;
* public virtual Film Film – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Film;
* public long FilmId – внешний ключ, связывающий данную сущность и сущность Film;
* public virtual Hall Hall – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Hall;
* public long HallId – внешний ключ, связывающий данную сущность и сущность Hall;
* public virtual ICollection<Ticket> Tickets – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Ticket;
* public virtual ICollection<ReservedSeat> ReservedSeats – навигационное свойство, обозначающее отношение между данной сущностью и сущностью ReservedSeat.

**3.1.5.11 Класс Ticket**

Класс Ticket является сущностью, которая соответствует таблице Ticket базы данных, и наследует свойства и методы базового класса Entity.

Свойства класса Ticket:

* public DateTime PaymentDate – хранит дату оплаты билета;
* public virtual ApplicationUser ApplicationUser – навигационное свойство, обозначающее отношение между данной сущностью и сущностью ApplicationUser;
* public string ApplicationUserId – внешний ключ, связывающий данную сущность и сущность ApplicationUser;
* public virtual Seance Seance – навигационное свойство, обозначающее отношение между данной сущностью и сущностью Seance;
* public long SeanceId – внешний ключ, связывающий данную сущность и сущность Seance;
* public virtual PaymentMethod PaymentMethod – навигационное свойство, обозначающее отношение между данной сущностью и сущностью PaymentMethod;
* public long PaymentMethodId – внешний ключ, связывающий данную сущность и сущность PaymentMethod;
* public long SeatId – внешний ключ, связывающий данную сущность и сущность Seat.

**3.1.6 Использование шаблона Unit Of Work**

*Шаблон Unit Of Work (Единица работы)* – это шаблон проектирования, который обслуживает набор объектов, изменяемых в бизнес-транзакции (бизнес-действии) и управляет записью изменений и разрешением проблем конкуренции данных.

Когда необходимо писать и читать из базы данных, важно следить за тем, что было изменено, и если не изменено, то не записывать данные в базу данных. Также необходимо вставлять данные о новых объектах и удалять данные о старых.

Можно записывать в базу данных каждое изменение объекта, но это приведёт к большому количеству мелких запросов к базе данных, что закончится замедлением работы проекта. Более того, это требует держать открытую транзакцию все время работы приложения, что непрактично, если проект обрабатывает несколько запросов одновременно. Ситуация еще хуже, если необходимо следить за чтением из базы данных, чтобы избежать неконсистентного чтения.

Реализация паттерна Unit Of Work следит за всеми действиями проекта, которые могут изменить базу данных в рамках одного бизнес-действия. Когда бизнес-действие завершается, Unit Of Work выявляет все изменения и вносит их в базу данных (см. чертеж ГУИР.400201.035 РР.1).

Ниже представлена структура класса, реализующего шаблон Unit Of Work:

public class UnitOfWork : IUnitOfWork

{

public ApplicationDbContext Context { get; private set; }

public UserManager<ApplicationUser> UserManager { get; set; }

public RoleManager<IdentityRole> RoleManager { get; private set; }

public UnitOfWork(ApplicationDbContext applicationDbContext)

{}

public IRepository<TEntity> GetRepository<TEntity>() where TEntity : Entity

{}

public void Save()

{}

}

Шаблон UnitOfWork призван отслеживать все изменения данных, которые производятся над доменной моделью в рамках бизнес-транзакции. После того, как бизнес-транзакция закрывается, все изменения попадают в БД в виде единой транзакции.

**3.2 Уровень бизнес**-**логики**

**3.2.1 Описание сервисов уровня бизнес-логики**

Сервисы на данном уровне проекта являются промежуточным звеном в процессах обработки данных, которые перемещаются между уровнями проекта. Также сервисы играют роль поставщиков данных, когда уровень представления запрашивает их с помощью интерфейса, предоставляемого сервисами. Сервисы используют экземпляр класса, реализующего паттерн Unit Of Work, как средство связи с уровнем доступа к данным. С помощью Unit Of Work сервисы составляют различные запросы к данным:

* сохранение данных;
* извлечение данных по различным критериям;
* удаление данных;
* модификация данных.

**3.2.1.1 Класс BaseService**

Класс BaseService является базовым классом для сервисов уровня доступа к данным, реализует интерфейс IService.

Поля класса BaseService:

* readonly IUnitOfWork Uow – объект, реализующий интерфейс IUnitOfWork.

Конструктор класса BaseService:

* BaseService(IUnitOfWork unitOfWork) – конструктор, инициализирующий поля данного класса.

**3.2.1.2 Класс UserManagerService**

Класс UserManagerService является сервисом, который предоставляет перечень операций, совершаемых над сущностью ApplicationUserDomainModel, наследуется от класса BaseService, реализует интерфейс IUserManagerService.

Основные методы класса UserManagerService:

* FindAsync(string email, string password) – асинхронно совершает поиск пользователя по данным авторизации;
* CreateAsync(ApplicationUserDomainModel userDomainModel, string password) – асинхронно создает пользователя с указанным паролем;
* ConfirmEmailAsync(string userId, string code) – асинхронно подтверждает адрес электронной почты пользователя;
* FindByNameAsync(string email) – асинхронно совершает поиск пользователя по имени;
* ResetPasswordAsync(string userId, string code, string password) – асинхронно производит сброс пароля;
* FindByIdAsync(string userId) – асинхронно совершает поиск пользователя по идентификатору;
* GetUsers() – извлекает список пользователей;
* IsInRole(string userId, string role) – проверяет, принадлежит ли пользователь указанной роли;
* RemoveFromRole(string userId, string role) – удаляет у пользователя указанную роль;
* AddToRole(string userId, string role) – добавляет пользователю указанную роль;
* GetRoles(string userId) – извлекает список ролей пользователя;
* BanUser(string userId) – блокирует пользователя;
* UnbanUser(string userId) – снимает блокировку с пользователя.

**3.2.1.3 Класс CinemaService**

Класс CinemaService является сервисом, который предоставляет перечень операций, совершаемых над сущностью CinemaDomainModel, наследуется от класса BaseService, реализует интерфейс ICinemaService.

Основные методы класса CinemaService:

* Add(CinemaDomainModel entity) – создает новый кинотеатр;
* Read() – извлекает список кинотеатров;
* ReadById(object id) – извлекает кинотеатр по идентификатору;
* Update(CinemaDomainModel entity) – модифицирует данные кинотеатра;
* Delete(CinemaDomainModel entity) – удаляет кинотеатр.

**3.2.1.4 Класс FilmService**

Класс FilmService является сервисом, который предоставляет перечень операций, совершаемых над сущностью FilmDomainModel, наследуется от класса BaseService, реализует интерфейс IFilmService.

Основные методы класса FilmService:

* Add(FilmDomainModelentity entity) – создает новый фильм;
* Read() – извлекает список фильмов;
* ReadById(object id) – извлекает фильм по идентификатору;
* Update(FilmDomainModelentity entity) – модифицирует данные фильма;
* Delete(FilmDomainModelentity entity) – удаляет фильм.

**3.2.1.5 Класс HallService**

Класс HallService является сервисом, который предоставляет перечень операций, совершаемых над сущностью HallDomainModel, наследуется от класса BaseService, реализует интерфейс IHallService.

Основные методы класса HallService:

* Add(HallDomainModel entity) – создает новый зал;
* Read() – извлекает список залов;
* ReadById(object id) – извлекает зал по идентификатору;
* Update(HallDomainModel entity) – модифицирует данные зала;
* Delete(HallDomainModel entity) – удаляет зал.

**3.2.1.6 Класс PaymentMethodService**

Класс PaymentMethodService является сервисом, который предоставляет перечень операций, совершаемых над сущностью PaymentMethodDomainModel, наследуется от класса BaseService, реализует интерфейс IPaymentMethodService.

Основные методы класса PaymentMethodService:

* Add(PaymentMethodDomainModel entity) – создает новый способ оплаты;
* Read() – извлекает список способов оплаты;
* ReadById(object id) – извлекает способ оплаты по идентификатору;
* Update(PaymentMethodDomainModel entity) – модифицирует данные способа оплаты;
* Delete(PaymentMethodDomainModel entity) – удаляет способ оплаты.

**3.2.1.7 Класс SeatService**

Класс SeatService является сервисом, который предоставляет перечень операций, совершаемых над сущностью SeatDomainModel, наследуется от класса BaseService, реализует интерфейс ISeatService.

Основные методы класса SeatService:

* Add(SeatDomainModel entity) – создает новое место;
* Read() – извлекает список мест;
* ReadById(object id) – извлекает место по идентификатору;
* Update(SeatDomainModel entity) – модифицирует данные места;
* Delete(SeatDomainModel entity) – удаляет место.

**3.2.1.8 Класс ReviewService**

Класс ReviewService является сервисом, который предоставляет перечень операций, совершаемых над сущностью ReviewDomainModel, наследуется от класса BaseService, реализует интерфейс IReviewService.

Основные методы класса ReviewService:

* Add(ReviewDomainModel entity) – создает новый отзыв;
* Read() – извлекает список мест;
* ReadById(object id) – извлекает отзыв по идентификатору;
* Update(ReviewDomainModel entity) – модифицирует данные отзыва;
* Delete(ReviewDomainModel entity) – удаляет отзыв.

**3.2.1.9 Класс ReservedSeatService**

Класс ReservedSeatService является сервисом, который предоставляет перечень операций, совершаемых над сущностью ReservedSeatDomainModel, наследуется от класса BaseService, реализует интерфейс IReservedSeatService.

Основные методы класса ReservedSeatService:

* Add(ReservedSeatDomainModel entity) – создает новый отзыв;
* Read() – извлекает список мест;
* ReadById(object id) – извлекает отзыв по идентификатору;
* Update(ReservedSeatDomainModel entity) – модифицирует данные отзыва;
* Delete(ReservedSeatDomainModel entity) – удаляет отзыв;
* ExemptExpiredSeats(long currentSeanceId) – снимает статус брони устаревших мест указанного сеанса.

**3.2.1.10 Класс SeanceService**

Класс SeanceService является сервисом, который предоставляет перечень операций, совершаемых над сущностью SeanceDomainModel, наследуется от класса BaseService, реализует интерфейс ISeanceService.

Основные методы класса SeanceService:

* Add(SeanceDomainModel entity) – создает новый сеанс;
* Read() – извлекает список мест;
* ReadById(object id) – извлекает сеанс по идентификатору;
* Update(SeanceDomainModel entity) – модифицирует данные сеанса;
* Delete(SeanceDomainModel entity) – удаляет сеанс.

**3.2.1.11 Класс Ticket**

Класс TicketService является сервисом, который предоставляет перечень операций, совершаемых над сущностью TicketDomainModel, наследуется от класса BaseService, реализует интерфейс ITicketService.

Основные методы класса TicketService:

* Add(TicketDomainModel entity) – создает новый билет;
* Read() – извлекает список билетов;
* ReadById(object id) – извлекает билет по идентификатору;
* Update(TicketDomainModel entity) – модифицирует данные билета;
* Delete(TicketDomainModel entity) – удаляет билет.

**3.3 Уровень представления**

На данном уровне проекта происходит формирование интерфейса, с которым в конечном итоге взаимодействует пользователь, а также обработка его действий, совершаемых в ходе работы пользователя с разрабатываемым приложением.

Уровень представления разделен на несколько компонентов, взаимодействующих друг с другом. В этом подразделе приведено описание работы этих компонентов, примененных шаблонов проектирования и связь уровня представления с уровнем бизнес-логики.

**3.3.1 Понимание MVC паттерна**

По большому счету MVC паттерн обозначает, что MVC приложение будет разделено как минимум на три части:

1. *Модели*, которые содержат или представляют данные, с которыми работают пользователи. Это могут быть простые *модели представления*, которые только представляют данные, передаваемые от контроллера представлению, или они могут быть *доменными моделями*, которые содержат данные домена, а также операции, преобразования и правила работы с этими данными.
2. *Представления*, которые используются для того, чтобы обработать некоторые части модели в качестве пользовательского интерфейса.
3. *Контроллеры*, которые обрабатывает входящие запросы, выполняют операции для модели и выбирают представления для показа пользователю.

В MVC контроллеры являются C# классами, как правило, производными от класса System.Web.Mvc.Controller. Каждый открытый (public) метод в классе, производный от Controller, называется методом действия, который связан с настраиваемым URL через систему маршрутизации (роутингом) ASP.NET. При отправке запроса на URL, связанный с методом действия, исполняются выражения в классе контроллера для того, чтобы выполнить некоторые операции над доменной моделью, а затем выбрать представление для отображения клиенту. На рисунке 3.2 показано взаимодействие между контроллером, моделью и представлением.



Рисунок 3.2 – Взаимодействие основных компонентов в MVC приложении

MVC – это не единственный архитектурный паттерн программного обеспечения, конечно, есть много других, и некоторые из них (или по крайней мере были) чрезвычайно популярны. Далее будут кратко описаны различные подходы к структурированию приложения и сопоставлены с MVC. Некоторые из паттернов являются близкими вариациями на тему MVC, в то время как другие совершенно отличаются.

**3.3.1.1 Паттерн Smart UI**

Один из наиболее распространенных паттернов известен как smart UI. Чтобы создать Smart UI приложение, разработчики выстраивают пользовательский интерфейс, обычно путем перетаскивания набора компонентов или элементов управления на дизайнерскую поверхность. Элементы управления сообщают о взаимодействии с пользователем, представляя события для нажатия кнопок, нажатия клавиш, движения мыши и так далее. Разработчик добавляет код в ответ на эти события в серии обработчиков событий – небольших блоков кода, которые вызываются, когда происходит определенное событие для определенного компонента. И тут заканчивается монолитное приложение, как показано на рисунке 3.3. Код, который обрабатывает пользовательский интерфейс и логика смешиваются вместе, понятия вообще не разделены. Код, который определяет допустимые значения для вводимых данных, запросов данных или изменяет учетную запись пользователя, делится на маленькие кусочки, соединенные вместе по порядку, в котором ожидаются события.



Рисунок 3.3 – Smart UI паттерн

Самым большим недостатком этой конструкции является то, что ее трудно поддерживать и расширять: смешивание доменной модели и кода бизнес логики с кодом пользовательского интерфейса приводит к дублированию, где тот же фрагмент бизнес логики копируется и вставляется для поддержки вновь добавленного компонента. Поиск всех дублирующих частей и внесение правок может быть трудным, а в сложном Smart UI приложении иногда почти невозможно добавить новую функцию, не ломая существующую. Тестирование Smart UI приложений также может быть трудным, единственный способ – это имитации взаимодействий с пользователем, которая далека от идеала и не обеспечивает основу для полного тестирования.

В мире MVC Smart UI часто называют анти-паттерном, которого следует избегать любой ценой. Эта антипатия возникла, по крайней мере, частично потому, что люди пришли к MVC в поисках альтернативы, проведя часть своей карьеры, пытаясь развивать и поддерживать Smart UI приложения. Это, безусловно, относится и к нам, и мы оба несем в себе следы этих долгих лет, но мы не отвергаем Smart UI паттерн полностью. Не все гладко в Smart UI паттерне, но есть и положительные аспекты в таком подходе. Smart UI приложения быстро и легко разрабатывать: создатели инструментария приложили много усилий для того, чтобы разработка была приятной, и даже самый неопытный программист может создать нечто профессионально выглядящее и достаточно функциональное в течение нескольких часов.

**3.3.1.2 Паттерн Model-View**

Область, в которой, как правило, возникают проблемы с поддержкой в Smart UI приложениях – это бизнес-логика, и внесение изменений может стать очень неприятным процессом. Улучшения в этой области предлагает архитектура model-view (модель-представление), которая вытаскивает бизнес-логику в отдельную доменную модель. При этом данные, процессы и правила все сосредоточены в одной части приложения, как показано на рисунке 3.4.



Рисунок 3.4 – Model-View паттерн

Архитектура model-view является значительным улучшением по сравнению со Smart UI – например, ее гораздо легче поддерживать. Тем не менее, возникают две проблемы. Первое, поскольку пользовательский интерфейс и доменная модель настолько тесно интегрированы, затрудняется выполнение юнит тестирования. Вторая проблема вытекает из практики, а не из определения паттерна. Эта модель обычно содержит массу кода доступа к данным, а это значит, что модель данных содержит не только бизнес данные, операции и правила.

**3.3.1.3 Паттерн Model-View-Presenter**

MVP является вариацией MVC, который разработан, чтобы соответствовать GUI платформам, сохраняющим состояние (stateful), таким как Windows Forms или ASP.NET Web Forms, и это стоящая попытка представить лучшие аспекты Smart UI без тех проблем, которые он обычно приносит.

В этом паттерне *презентер* (presenter) имеет те же обязанности, что и MVC контроллер, но он также имеет более непосредственное отношение к представлениям, напрямую управляя значениями, отображаемыми в UI компонентах, в зависимости от действий и данных, введенных пользователем. Существуют две реализации этого паттерна:

1. Реализация *пассивного представления*, в котором представление не содержит логики – это контейнер для элементов управления UI, которые напрямую управляются презентером.
2. Реализация *надзирающего контроллера*, в котором представление может быть ответственным за некоторые элементы логики представления, такие как связанные данные, и к источнику данных можно ссылаться из доменной модели.

Разница между этими двумя подходами касается того, насколько умным является представление. В любом случае, презентер отделяется от GUI фреймворка, что делает логику презентера простой и подходит для модульного тестирования.

**3.3.1.4 Паттерн Model-View-View Model**

MVVM паттерн является самой последней вариацией MVC. Он возник в 2005 году в команде Microsoft, которая разрабатывала технологию, ставшую Windows Presentation Foundation (WPF) и Silverlight.

В MVVM паттерне модели и представления играют те же роли, что и в MVC. Разница состоит в MVVM концепции модели представления, которая является абстрактным представлением пользовательского интерфейса. Как правило, это C# класс, который раскрывает оба свойства для данных, которые будут отображаться в интерфейсе, и операции по данным, которые могут быть вызваны из пользовательского интерфейса. В отличие от MVC контроллера, модель представления MVVM не имеет понятия, что существует представление (или любая конкретная UI технология). MVVM представление использует связывающую функцию WPF/Silverlight для двунаправленного связывания свойств, предоставляемых элементами управления в представлении (пункты в выпадающем меню или результат нажатия кнопки), со свойствами, предлагаемыми моделью представления.

MVVM тесно сотрудничает со связывающими функциями WPF и поэтому это не тот паттерн, который легко применить и к другим платформам.

**3.3.2 Описание контроллеров**

**3.3.2.1 Класс UserManipulationController**

Класс UserManipulationController управляет представлениями, которые позволяют администраторам и модераторам приложения следить и управлять ролями, правами доступа других пользователей.

Основные методы класса UserManipulationController:

* UserManagement() – возвращает представление для управления пользователями;
* SetToRole(string name) – задает роль пользователя;
* ChangeUserAccess(string name) – изменяет права доступа пользователя.

**3.3.2.2 Класс HomeController**

Класс HomeControllerr управляет представлениями, которые позволяют пользователям взаимодействовать с домашней страницей сайта и совершать базовые действия.

Основные методы класса HomeController:

* Index() – возвращает представление домашней страницы сайта;
* FilmList() – возвращает представление списка доступных фильмов;
* Details(long? filmId) – возвращает представление с подробным описанием конкретного фильма;
* Browse() – возвращает представление дат проведения киносеансов;
* CinemaRowList(string date) – возвращает представление списка кинотеатров по конкретной дате.

**3.3.2.3 Класс AccountController**

Класс AccountController управляет представлениями, которые позволяют пользователям аутентифицироваться, регистрироваться в системе, сбрасывать пароль, управлять своими учетными данными.

Основные методы класса AccountController:

* Summary() – возвращает представление с общей информацией о пользователе;
* Login(string returnUrl) – возвращает представление аутентификации пользователя в системе;
* Login(LoginViewModel model, string returnUrl) – аутентифицирует пользователя в системе;
* ExternalLoginCallback(string returnUrl) – аутентификация пользователя с помощью внешней учетной записи;
* Register() – возвращает представление регистрации пользователя в системе;
* Register(RegisterViewModel model) – аутентифицирует пользователя в системе;
* ConfirmEmail(string userId, string code) – подтверждает адрес электронной почты пользователя;
* ResetPassword(string code) – возвращает представление по сбросу пароля;
* ResetPassword(ResetPasswordViewModel model) – сбрасывает пароль пользователя;
* Manage(ManageMessageId? message) – возвращает представление по управлению учетными данными пользователя;
* Manage(ManageUserViewModel model) – сохраняет изменения учетных данных пользователя;
* LogOff()– выход пользователя из системы.

**3.3.2.4 Класс CinemaController**

Класс CinemaController дает возможность администраторам и модераторам управлять данными кинотеатров.

Основные методы класса CinemaController:

* Index() – возвращает представление со списком кинотеатров;
* Details(long? id) – возвращает представление с подробным описанием конкретного кинотеатра;
* Create() – возвращает представление с формой для создания нового кинотеатра;
* Create(CinemaViewModel cinemaViewModel) – создает новый кинотеатр;
* Edit(long? id) – возвращает представление с формой для редактирования данных кинотеатра;
* Edit(CinemaViewModel cinemaViewModel) – сохранение изменений в данных о кинотеатре;
* Delete(long? id) – возвращает представление с подтверждением удаления кинотеатра;
* DeleteConfirmed(long id) – удаляет кинотеатр.

**3.3.2.5 Класс FilmController**

Класс FilmController дает возможность администраторам и модераторам управлять данными кинотеатров.

Основные методы класса FilmController:

* Index() – возвращает представление со списком фильмов;
* Details(long? id) – возвращает представление с подробным описанием конкретного фильма;
* Create() – возвращает представление с формой для создания нового фильма;
* Create(FilmViewModel filmViewModel) – создает новый фильм;
* Edit(long? filmId) – возвращает представление с формой для редактирования данных фильма;
* Edit(FilmViewModel filmViewModel) – сохранение изменений в данных о фильме;
* Delete(long? filmId) – возвращает представление с подтверждением удаления фильма;
* Delete (long filmId) – удаляет фильм.