МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное

образовательное учреждение

высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ

ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ

УНИВЕРСИТЕТ ИМЕНИ Н.Г.ЧЕРНЫШЕВСКОГО»

УТВЕРЖДАЮ

|  |  |  |
| --- | --- | --- |
| Зав. профилирующей кафедрой | | |
|  | | |
| уч. ст., уч. зв. | | |
|  | | |
|  |  |  |
| подпись, дата |  | иниц., фамилия |

**ОТЧЕТ ПО ПРАКТИКЕ**

Студента\_\_\_\_ курса \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ факультета

наименование факультета

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

фамилия, имя, отчество

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

вид практики

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

кафедра

курс\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

семестр\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

продолжительность\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

кол. недель, сроки практики

Руководитель практики от университета

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| должность, уч. ст., уч. зв. |  | личная подпись, дата |  | инициалы, фамилия |

Руководитель практики от организации (учреждения, предприятия)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| должность, уч. ст., уч. зв. |  | личная подпись, дата |  | инициалы, фамилия |

Оглавление

[Введение 3](#_Toc528148391)

[Трёхслойная архитектура 4](#_Toc528148392)

[Слой клиента 4](#_Toc528148393)

[Слой бизнес логики 5](#_Toc528148394)

[Слой базы данных 8](#_Toc528148395)

[Ninject 9](#_Toc528148396)

[Сущность (Entity) 11](#_Toc528148397)

[База данных 14](#_Toc528148398)

[Паттерн MVC 15](#_Toc528148399)

[Модель 17](#_Toc528148400)

[Представление 18](#_Toc528148401)

# Введение

Цель практики – изучение трёхслойной архитектуры, которая хранит информацию в базе данных, и паттерна MVC на платформе .Net.

В ходе практики были изучены такие понятия, как:

1. уровень работы с базой данных;
2. уровень бизнес логики;
3. пользовательский интерфейс;
4. создание базы данных для хранения информации;
5. создание хранимых процедур;
6. построение веб-приложения на основе MVC.

# Трёхслойная архитектура

Трёхслойная архитектура – это архитектурная модель программного комплекса, включающая следующие компоненты: клиент, сервер приложений, сервер базы данных.

К достоинствам можно отнести следующее:

1. масштабируемость;
2. конфигурируемость;
3. высокая безопасность;
4. высокая надёжность;
5. низкие требования к скорости канала между терминалами и сервером приложений;
6. низкие требования к производительности и техническим характеристикам терминалов, как следствие снижение их стоимости.

Из достоинств вытекают следующие недостатки:

1. более высокая сложность создания приложений;
2. сложнее в разворачивании и администрировании;
3. высокие требования к производительности серверов приложений и сервера базы данных, а, значит, и высокая стоимость серверного оборудования;
4. высокие требования к скорости канала между сервером базы данных и серверами приложений.

Сначала приложение создаётся как просто консольное приложение. Слой клиента является консольным приложением. Дальнейшие компоненты добавляются как библиотеки классов.

## Слой клиента

Этот компонент отвечает за взаимосвязь с пользователем, то есть за интерфейс, который пользователь будет видеть на экране компьютера или телефона.

В программе, которая была реализована во время летней практики, этот компонент называется PL (Presentation Layer). Интерфейс представляет собой консоль, в которой пользователь может выбирать определённые действия, такие как добавление пользователя, медали или награждение пользователя определённой наградой:

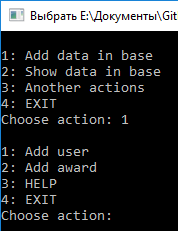


Рисунок 1 – Пример консольного интерфейса

Слой клиента взаимодействует с бизнес логикой, используя слабое связывание, то есть через интерфейс. В данном случае используются интерфейсы слоя логики IUserLogic и IAwardLogic, в которых перечислены методы для работы с логикой программы. Для взаимодействия со слоем логики через интерфейс в методе Main(string[] args) инициализируются переменные userLogic и awardLogic с помощью IoC-контейнера Ninject.

## Слой бизнес логики

Этот компонент отвечает за логику приложения, то есть обработку и выдачу данных в нужном формате.

В программе, которая была реализована во время летней практики, этот компонент называется BLL (Business Logic Layer). Слой представляет из себя набор методов для обработки входящих данных. Например, метод UpdateUser(string id, string name, string birthday), который обновляет данные пользователя и возвращает либо true, либо false, сообщая о том, прошла операция или нет. На вход метод получает номер пользователя – id, новое имя пользователя – name и новую дату рождения – birthday. Внутри данные преобразуются в нужный формат для работы с базой данных. После обработки данных метод передаёт параметры другому методу UpdateUser, расположенный в слое базы данных. Если данные введены пользователем неправильно, то метод возвращает false и сообщение об ошибке. В параметры не входит возраст пользователя, так как он высчитывает с помощью отдельного приватного метода SetAge(DateTime birthday), который возвращает целочисленное значение, то есть возраст пользователя.

…..

public int SetAge(DateTime birthday)

{

if (DateTime.Today.Month > birthday.Month)

{

return (DateTime.Today.Year - birthday.Year);

}

else

{

return (DateTime.Today.Year - birthday.Year) - 1;

}

}

…..

…..

public bool UpdateUser(string id, string name, string birthday)

{

DateTime dateTime;

int userId;

if (DateTime.TryParse(birthday, out dateTime) && (Int32.TryParse(id, out userId)))

{

if (GetUserById(userId) != null)

{

\_userDao.UpdateUser(userId, name, dateTime, SetAge(dateTime));

return true;

}

else

{

Console.WriteLine($"DB has no information");

return false;

}

}

else

{

Console.WriteLine($"Incorrect id or birthday");

return false;

}

}

…..

Слой бизнес логики взаимодействует со слоем базы данных, используя слабое связывание, то есть интерфейс. В данном случае используется интерфейс слоя базы IUserDao, в котором перечислены методы для работы с базой данных. Для взаимодействия со слоем базы данных через интерфейс в конструкторе инициализируется переменная \_userDao.

…..

private readonly IUserDao \_userDao;

public UserLogic(IUserDao userDao)

{

\_userDao = userDao;

}

…..

## Слой базы данных

Этот компонент отвечает за работу с самой базой данных, которая хранит все себе данные о пользователях и наградах.

В программе, которая была реализована во время летней практики, этот компонент называется DAL (Data Access Layer). Слой представляет собой набор методов для работы с базой данных. Например, метод UpdateUser(int wantedId, string wantedName, DateTime wantedBirthday, int wantedAge). С помощью SqlConnection осуществляется подключение к базе данных. Далее, используя метод CreateCommand(), указывается используемая хранимая процедура. В данном случае используется хранимая процедура под именем UpdateUser. После определяются параметры для хранимой процедуры: @ID, @NAME, @BIRTHDAY, @AGE, которые соотносятся с входящими параметрами соответственно. С помощью метода Open() открывается подключение к базе данных, а метод ExecuteNonQuery() выполняет процедуру, то есть обновление пользователя в базе данных.

…..

public int UpdateUser(int wantedId, string wantedName, DateTime wantedBirthday,

int wantedAge)

{

using (var connection = new SqlConnection(\_connectionString))

{

var command = connection.CreateCommand();

command.CommandType = System.Data.CommandType.StoredProcedure;

command.CommandText = "UpdateUser";

var id = new SqlParameter("@ID", SqlDbType.Int)

{

Value = wantedId

};

command.Parameters.Add(id);

var name = new SqlParameter("@NAME", SqlDbType.VarChar)

{

Value = wantedName

};

command.Parameters.Add(name);

var birthday = new SqlParameter("@BIRTHDAY", SqlDbType.DateTime)

{

Value = wantedBirthday

};

command.Parameters.Add(birthday);

var age = new SqlParameter("@AGE", SqlDbType.Int)

{

Value = wantedAge

};

command.Parameters.Add(age);

connection.Open();

return (int)(decimal)command.ExecuteNonQuery();

}

}

…..

## Ninject

Ninject – это быстрая и лёгкая библиотека для внедрения зависимостей (Dependency Injection) для приложений платформы .Net.

Данная библиотека помогает организовать слабое связывание между слоями трёхслойного приложения. Иными словами, используется последний принцип SOLID – принцип инверсий зависимостей (Dependency Inversion Principle).

В программе, которая была реализована во время летней практики, в папке Common добавлена библиотека классов под именем UserAward.Container, в котором находится статический класс NinjectCommon, осуществляющий работу с Ninject.

Чтобы управлять зависимостями через Ninject, сначала надо создать объект Ninject.IKernel с помощью встроенной реализации этого интерфейса – класса StandardKernel. После этого нужно установить отношения между интерфейсами и их реализациями. Для этого используется конструкция .Bind<>().To<>(), в которой в Bind указывается запрашиваемый интерфейс, а в To куда надо направить запрос, то есть, если надо обратиться к методу в логике для работы с пользователем, то обращение идёт к интерфейсу IUserLogic. Далее обращение переходит к классу, который реализует этот интерфейс, то есть к UserLogic. В статическом методе без выходных параметров Registration() записываются подобные конструкции для работы со слоями.

В методе Main(string[] args) происходит внедрение зависимостей:

userLogic = NinjectCommon.Kernel.Get<IUserLogic>();

В данном примере происходит обращение с интерфейсу IUserLogic и присваивание результата переменной userLogic. После этого с помощью переменной можно обращаться к методам класса UserLogic.

Перед внедрением зависимостей в методе Main(string[] args) обязательно нужно обратить к методу Registration().

…..

public static class NinjectCommon

{

private static readonly IKernel \_kernel = new StandardKernel();

public static IKernel Kernel => \_kernel;

public static void Registration()

{

\_kernel.Bind<IUserDao>().To<UserDao>();

\_kernel.Bind<IUserLogic>().To<UserLogic>();

\_kernel.Bind<IAwardDao>().To<AwardDao>();

\_kernel.Bind<IAwardLogic>().To<AwardLogic>();

}

}

…..

…..

private static IUserLogic userLogic;

private static IAwardLogic awardLogic;

static void Main(string[] args)

{

NinjectCommon.Registration();

userLogic = NinjectCommon.Kernel.Get<IUserLogic>();

awardLogic = NinjectCommon.Kernel.Get<IAwardLogic>();

…..

## Сущность (Entity)

Для работы с таблицами, которые хранятся в базе данных, в трёхслойном приложении нужны сущности, которые имеют такие же поля, как и таблицы в базе данных. Например, в таблице Award существуют следующие поля: id\_award, Title, Description. Каждое поле имеет свой тип, например, Title имеет тип varchar с размерностью в 50 символов (ссылка на приложение). В самой программе в библиотеке классов Entity существует класс Award, который состоит из следующих автосвойств: IdAward, Title, Description. Каждое автосвойство имеет тип, который соответствует типу в таблице базы данных, то есть для Title это будет тип string. Такой же класс существует для таблицы базы данных, которая хранит данные о пользователе.

…..

public class Award

{

public int IdAward { get; set; }

public string Title { get; set; }

public string Description { get; set; }

}

…..

Данные сущности используются в слое логики и слое базы данных. В слое логики они нужны для создания объекта класса с нужными параметрами и передачи его как параметра в методы слоя базы данных. В слое базы данных используются параметры переданного объекта класса.

…..

var title = new SqlParameter("@TITLE", System.Data.SqlDbType.VarChar)

{

Value = award.Title

};

…..

Также сущность используется для извлечения информации из базы данных. Например, метод GetAwards(), который возвращает коллекцию, состоящую из объектов класса Award.

…..

public IEnumerable<Award> GetAwards()

{

using (var connection = new SqlConnection(\_connectionString))

{

var command = connection.CreateCommand();

command.CommandType = System.Data.CommandType.StoredProcedure;

command.CommandText = "GetAwards";

connection.Open();

using (var reader = command.ExecuteReader())

{

while (reader.Read())

{

yield return new Award

{

IdAward = (int)reader["id\_award"],

Title = (string)reader["Title"],

Description = (string)reader["Description"]

};

}

}

}

}

…..

# База данных

Во время летней практики для трёхслойного приложения нужно было создать базу данных, которая хранит информацию о пользователях и наградах. Помимо этого, нужно было создать хранимые процедуры, чтобы, например, найти пользователя по его номеру или имени.

Для написания базы данных был использован Microsoft Server SQL.

База данных состоит из трёх таблиц: User, Award, User\_Award. В таблице User хранится информация о пользователях, то есть номер, имя, день рождения и возраст. В таблице Award хранится информация о наградах, то есть номер, название и описание награды. В таблице User\_Award хранится информация о том, какой пользователь какую награду имеет, то есть номер пользователя и номер награды.

Целью написания хранимых процедур была упрощение работы с данными из базы данных. Используя хранимые процедуры, сокращается код программы. Например, в методе для обновления данных о пользователе в базе данных происходит обращение к хранимо процедуре, которая, с помощь переданных ей параметров, обновляет данные. Без хранимой процедуры пришлось бы писать полный SQL-скрипт.

# Паттерн MVC

Model-View-Controller или Модель-Представление-Контроллер — схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

Модель предоставляет данные и методы работы с ними: запросы в базу данных, проверка на корректность. Модель не зависит от представления или контроллера, но реагирует на команды контроллера, изменяя своё состояние.

Представление отвечает за получение необходимых данных из модели и отправляет их пользователю, реагируя на изменение модели. Представление не обрабатывает введённые данные пользователя

Контроллер интерпретирует действия пользователя, оповещая модель о необходимости изменений. Контроллер обеспечивает связь между пользователем и системой, контролирует и направляет данные от пользователя к системе и наоборот, использует модель и представление для реализации необходимого действия.

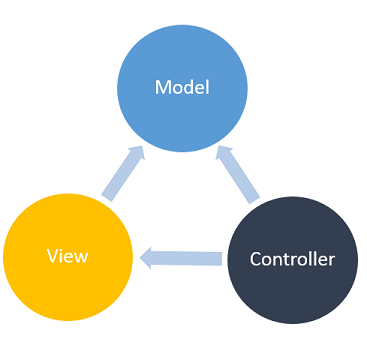


Рисунок 2 – Паттерн MVC

Созданный проект имеет следующую структуру:

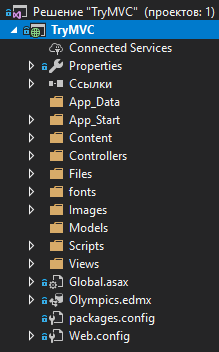


Рисунок 3 – Структура проекта MVC

Папка App\_Data содержит файлы, ресурсы и базы данных, используемые приложением.

Папка App\_Start хранит ряд статических файлов, которые содержат логику инициализации приложения при запуске.

Папка Content содержит вспомогательные файлы, которые не включают код на C# или javascript, и которые развертываются вместе с приложением, например, файлы стилей css.

Папка Controllers содержит файлы классов контроллеров. По умолчанию в эту папку добавляется HomeController.

Папка fonts хранит дополнительные файлы шрифтов, используемых приложением.

Папка Models содержит файлы моделей.

Папка Scripts – каталог со скриптами и библиотеками на языке javascript.

Папка Views хранит представления. Все представления группируются по папкам, каждая из которых соответствует одному контроллеру. После обработки запроса контроллер отправляет одно из этих представлений клиенту. Также здесь имеется каталог Shared, который содержит мастер-страницу.

Global.asax – файл, запускающийся при старте приложения и выполняющий начальную инициализацию. Как правило, здесь срабатывают методы классов, определенных в папке App\_Start.

Web.config – файл конфигурации приложения.

Папка Files была добавлена по заданию для хранения файла, который можно было скачивать. В этом файле хранилась информация о пользователях.

Папка Images хранит изображения.

В Olympics.edmx хранятся модели, созданные на основе таблиц и хранимых процедур базы данных, а также диаграмма.

## Модель

В реализованном за время летней практики проекте все модели были созданы автоматически с помощью EDM – это набор основных понятий, которые описывают структуру данных независимо от формы хранения. В результате создались классы, которые описывают таблицы и процедуры, которые хранились в базе данных.

Пример класса, созданного на основе процедуры GetAwardById:

…..

public partial class GetAwardByID\_Result

{

public int id\_award { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public byte[] AwardImage { get; set; }

}

…..

## Представление

Представления представляют из себя html страницы, которые будут видны пользователю.

Чтобы использовать код C# в представлениях, используется Razor Pages — это аспект платформы MVC ASP.NET Core, который делает создание кодов сценариев для страниц проще и эффективнее. Чтобы начать использовать Razor, нужно прописать специальный символ «@», а после писать код:

…..

@if (item.UserPhoto != null)

{

@Html.Raw("<img style='width:80px; height:60px;' src=\"data:image/jpeg;base64, Convert.ToBase64String(item.UserPhoto) + "\" />")

}

else

{

<img style='width:80px; height:60px;' src="~/Images/usualPic.jpg" />

}

…..

Как видно из примера, можно одновременно использовать как синтаксис C#, так и синтаксис HTML.

Фреймворк ASP.NET MVC обладает также таким мощным инструментом как HTML-хелперы, позволяющие генерировать html-код. Например, хелпер Html.BeginForm. Он позволяет создать html-форму, которая по нажатию на кнопку отправляет все введенные данные запросом POST на нужный адрес. Метод BeginForm принимает в качестве параметров имя метода действия, имя контроллера, тип запроса и атрибут enctype, который определяет способ кодирования данных формы при их отправке на сервер. В следующем примере идёт обращение к методу Create, который находится в контроллере Award. В качестве типа запроса указывается Post, а дальше указывается кодировка.

…..

@using (Html.BeginForm("Create", "Awards", FormMethod.Post, new { enctype = "multipart/form-data" }))

…..

Данный хелпер создает как открывающий тег <form>, так и закрывающий тег </form>. Поэтому при рендеринге представления в выходной получается тот же самый html-код, что и с применением тега form. Поэтому оба способа идентичны.

Если в представление передаётся какой-либо параметр, то его можно передать разными способами. Например, передать в @model:

…..

@model TryMVC.Award

…..

После этого можно использовать ключевое слово Model, которое хранит в себе то, что было передано:

…..

@foreach (var item in Model)

…..

Недостатком такого способа является то, что Model может быть только один.

Можно использовать ViewData, который представляет из себя словарь:

…..

ViewData["Head"] = "Привет мир!";

…..

Также можно использовать ViewBag, который может принять что угодно:

…..

ViewBag.Head = "Привет мир!";

…..

## Контроллер

В контроллерах прописываются методы для работы с представлениями и моделями, а также некоторая логика. Например, метод Rewarding. Этот метод создан для награждения людей медалями. Как можно заменить, реализовано два метода: один использует Get, второй использует Post. Если не указывать тип запроса явно, то метод будет использовать Get и Post одновременно. Методы типа ActionResult должны что-то возвращать. В данном примере метод Rewarding возвращает страницу с ошибкой, если параметр id равен null. Если id не null и существует пользователь с таким id, то метод возвращает представление, в которое передаёт данные.

…..

[HttpGet]

public ActionResult Rewarding(int? id)

{

if (id == null)

{

return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

}

User user = db.Users.Find(id);

if (user == null)

{

return HttpNotFound();

}

ViewBag.UserId = id;

SelectList awards = new SelectList(db.Awards, "id\_award", "Title");

ViewBag.Awards = awards;

return View(user);

}

[HttpPost]

public ActionResult Rewarding(Award award, User\_Award user\_Award, User user)

{

if (ModelState.IsValid)

{

if (db.Awards.Find(award.id\_award) != null)

{

user\_Award.id\_award = award.id\_award;

user\_Award.id\_user = user.id\_user;

db.User\_Award.Add(user\_Award);

db.SaveChanges();

return RedirectToAction("Index");

}

}

return View(user);

}

…..

Далее представлен пример страницы со списком пользователей:

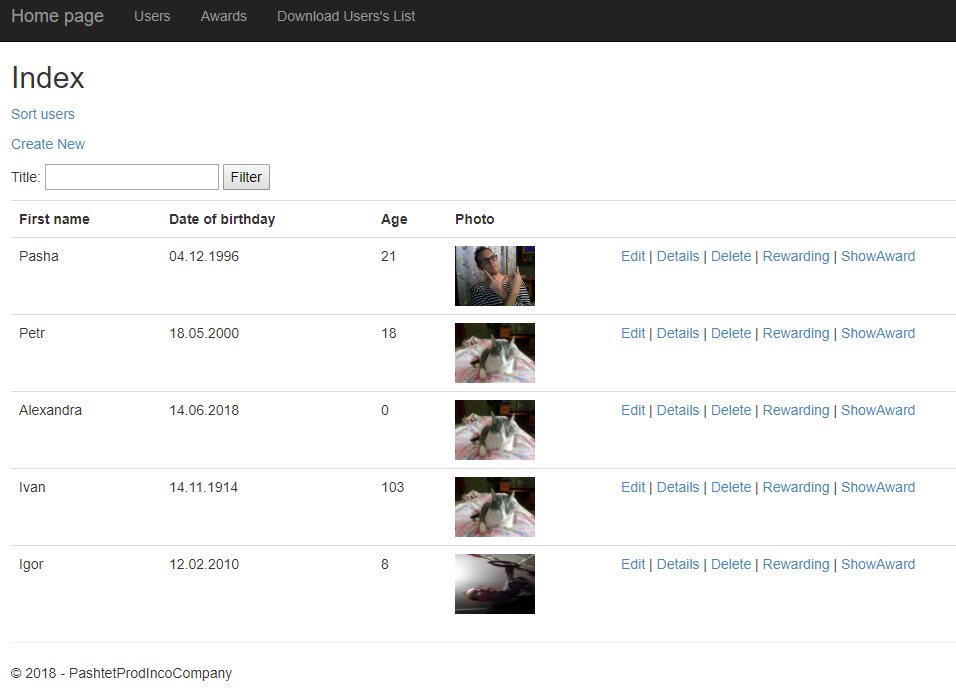


Рисунок 4 - Пример страницы с пользователями