# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_Toc509196369)

[ВВЕДЕНИЕ 3](#_Toc509196370)

[1 Общая информация 4](#_Toc509196371)

[1.1 Журнал практики 4](#_Toc509196372)

[2 Преддипломная составляющая практики 5](#_Toc509196373)

[2.1 Серверная часть web-приложения “ArchiveManager” 5](#_Toc509196374)

[2.1.1 Постановка задачи 5](#_Toc509196375)

[2.1.2 Создание проекта 5](#_Toc509196376)

[2.1.3 Установка зависимостей в проект 6](#_Toc509196377)

[2.1.4 Создание структуры проекта 8](#_Toc509196378)

[2.1.5 Создание доменной модели 9](#_Toc509196379)

[2.1.6 Создание сервисов 12](#_Toc509196380)

[2.1.7 Добавление контроллеров 16](#_Toc509196381)

[2.1.8 Конфигурация приложения 20](#_Toc509196382)

[2.1.9 Миграция данных 22](#_Toc509196383)

[2.2 Клиентская часть web-приложения “ArchiveManager” 23](#_Toc509196384)

[2.2.1 Постановка задачи 23](#_Toc509196385)

[2.2.2 Создание проекта 23](#_Toc509196386)

[2.2.3 Реализация логики приложения 25](#_Toc509196387)

[3 МАСТЕР-КЛАСС 28](#_Toc509196388)

[ЗАКЛЮЧЕНИЕ 29](#_Toc509196389)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 30](#_Toc509196390)

[ПРИЛОЖЕНИЕ А 31](#_Toc509196391)

[ПРИЛОЖЕНИЕ Б 32](#_Toc509196392)

# ВВЕДЕНИЕ

На сегодняшний день во многих сферах человеческой жизнедеятельности используются ресурсы всемирной паутины. И почти в каждый человек имеет доступ в интернет. Это способствовало тому, что преобладающему количеству пользователей проще открыть страницу в браузере, чем найти, скачать, установить программное обеспечение на свой компьютер. Для реализации web-приложений существуют различные инструменты.

Платформа ASP.NET Core представляет технологию от компании Microsoft, предназначенную для создания различного рода веб-приложений: от небольших веб-сайтов до крупных веб-порталов и веб-сервисов. [2]

React - это JavaScript-библиотека, которая используется для создания пользовательского интерфейса. Данная библиотека является достаточно популярной, что подкрепляется ее быстротой, гибкостью, производительностью.

На преддипломную практику были вынесены следующие задачи:

* изучить платформу ASP.NET Core;
* изучить JavaScript-библиотеку React;
* разработать серверную часть web-приложения “ArchiveManager” на платформе ASP.NET Core;
* разработать клиентскую часть web-приложения “ArchiveManager” используя JavaScript-библиотеку React.

# Общая информация

## Журнал практики

|  |  |
| --- | --- |
| **Дата** | **Описание** |
| 05.02.2018 | Прибытие в фирму, инструктажи |
| 06.02.2018  -16.02.2018 | Изучение книги Адам Фримен ASP.NET Core [1]. С реализацией примеров из книги. |
| 16.02.2018  -23.02.2018 | Изучение JS-библиотеки React. С реализацией примеров при изучении. |
| 26.02.2018 | Установка Visual Studio 2017. |
| 27.02.2018 | Начало разработки серверной части приложения “ArchiveManager”. Проектирование приложения. |
| 28.03.2018 | Создание проекта, реализация основного каркаса приложения. |
| 01.03.2018 | Реализация возможности загрузки на сервер файла. |
| 02.03.2018 | Рассмотрение net-библиотек для управления архивами. |
| 03.03.2019 | Реализация распаковки архива на сервере. |
| 05.03.2018 | Реализация получения доступа к содержимому загруженного на сервер архива. |
| 06.03.2018 | Изучение npm. |
| 07.03.2018 | Изучение webpack. |
| 12.03.2018 | Добавление в проект поддержки webpack. Его конфигурация. |
| 13.03.2018 | Изучение Reflux |
| 14.08.2018 | Реализация основного каркаса клиентской части. |
| 15.08.2018 | Изучение jQuery Ajax |
| 16.08.2018 | Реализация возможности скачать архив с сервера. |

# Преддипломная составляющая практики

## Серверная часть web-приложения “ArchiveManager”

### Постановка задачи

Требуется создать web-приложение на платформе ASP.NET Core. Которое будет предоставлять API управления архивом. Должны быть реализованы такие возможности:

* загрузка архива на сервер;
* получение доступа к содержимому архива;
* возможность изменить содержимое архива;
* возможность просматривать другим пользователям содержимое архива.

### Создание проекта

Средой разработки приложения была выбрана Visual Studio 2017. Для создания проекта, по разработке ASP.NET Core приложения, следует выполнит следующие действия:

* в меню «*Файл*» щелкнуть «*Создать проект*». Откроется диалоговое окно «*Новый проект*»;
* в списке шаблонов проектов Visual C# выбрать проект “Веб-приложение ASP.NET Core” и присвоить ему имя «*ArchiveManager*». Нажать кнопку ОК;
* в списке предоставленных шаблонов типов проектов выбрать “Пустой”. Нажать кнопку ОК.

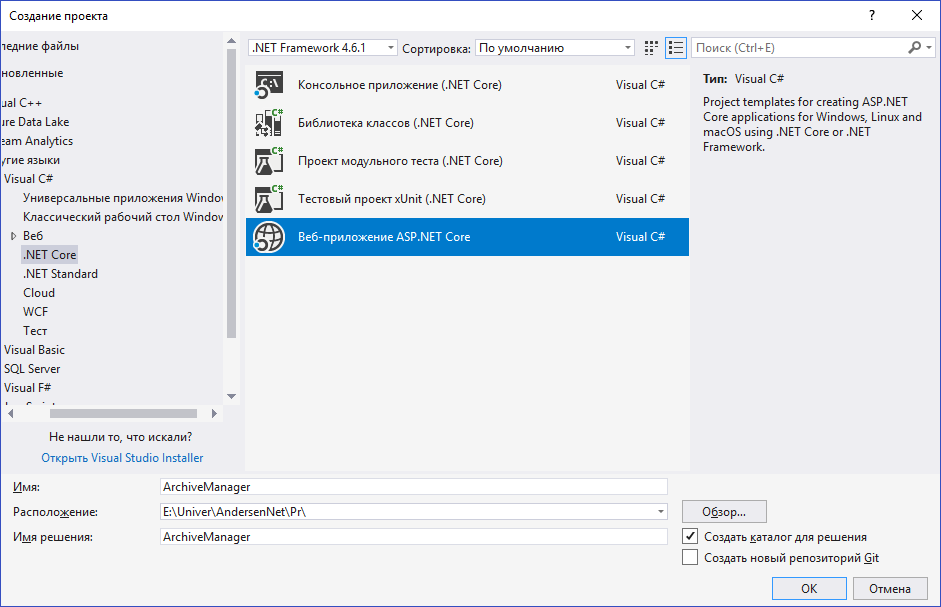


Рисунок 2.1.2.1 – Создание проекта

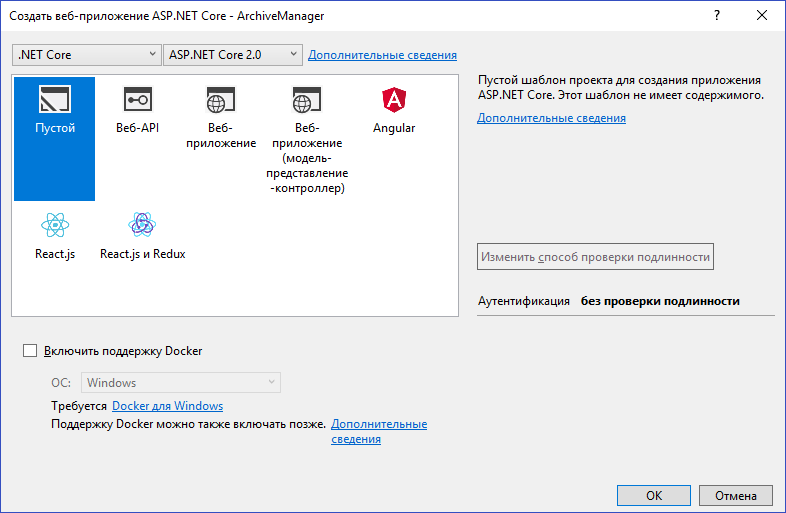


Рисунок 2.1.2.2 – Выбор типа проекта

### Установка зависимостей в проект

По умолчанию в зависимости проекта добавляется ссылка на “Microsoft.AspNetCore.All”. Данный пакет содержит основную функциональность для разработки ASP.NET Core приложений. Но необходимо добавить дополнительные зависимости, такие как:

* “Microsoft.EntityFrameworkCore.Sqlite”. Данный пакет предоставляет возможность управлять базой данных SqlLite;
* “Microsoft.EntityFrameworkCore.Tools”. Данный пакет дает возможность синхронизировать состояние базы данных и классов модели в проекте. То есть данный пакет представляет сущности в проекте в виде таблиц в базе данных;
* “Microsoft.VisualStudio.Web.CodeGeneration.Design”. Предоставляет возможность IDE генерировать готовые сущности по определенному шаблону;
* “SharpCompress”. Предоставляет возможность управления различными типами архивов.

Добавить зависимости в проект можно различными способами. Один из них — это использовать “Менеджер управления пакетами NuGet”.

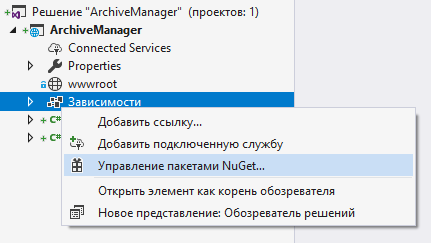


Рисунок 2.1.3.1– Запуск менеджера управления пакетами NuGet

Так как с релиза NET.Core прошло еще не много времени, некоторые разработчики пакетов еще не успели выпустить стабильные версии своих продуктов. Поэтому некоторые пакеты находятся в стадии “Предварительный доступ”. В менеджере пакет можно включить показ таких пакетов.

### Создание структуры проекта

При разработке приложение ASP.NET существуют некоторые соглашения, которым должны следовать разработчики. Некоторые соглашения:

* контроллеры должны находится в папке “Controllers”;
* доменные сущности должны находиться в папке “Models”;

Было принято решение создать следующую структуру проекта.

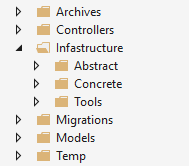


Рисунок 2.1.4.1– Структура проекта

Содержимое папок:

* Archives – здесь будут находиться распакованные версии архивов пользователей;
* Cotrollers – все контроллеры;
* Infrastructure – реализация различных сервисов;
* Migrations-эта папка будет создана автоматически при использовании миграций;
* Temp – временное хранилище файлов

### Создание доменной модели

В данном приложении будет использоваться подход “Code First”. Данный подход подразумевает, что сначала будут написаны сущности в .NET, а после при помощи миграции на основе полученных сущностей будет построена структура базы данных.

Основными сущностями в данном приложении являются “Archive”, ”Folder”, “File”.

Листинг 2.1.5.1 – Сущность “Archive”

public class Archive{

public int Id { get; set; }

public string Key { get; set; }

public int? FolderId { get; set; }

public Folder Folder { get; set; }

}

Листинг 2.1.5.2 – Сущность “File”

public class File {

public int Id { get; set; }

[StringLength(255)]

public string Name { get; set; }

public string ContentType { get; set; }

public int FolderId { get; set; }

public Folder Folder { get; set; }

}

Листинг 2.1.5.3 – Сущность “Folder”

public class Folder {

public int Id { get; set; }

public string Name {

get {

return IO.Path.GetFileName(Path);

}

}

public int? ParentId { get; set; }

public Folder Parent { get; set; }

public int ArchiveId { get; set; }

public Archive Archive { get; set; }

public ICollection<Folder> Folders { get; set; }

public ICollection<File> Files { get; set; }

public string Path { get; set; }

public Folder() {

Folders = new List<Folder>();

Files = new List<File>();

}

}

На основании данных сущностей будет построена база данных. Так же, чтоб инструмент миграции данных “увидел” на основании каких классов строить базу, необходимо создать контекст данных.

Листинг 2.1.5.4 – Контекст данных

public class ApplicationContext : DbContext {

public DbSet<Archive> Archives { get; set; }

public DbSet<Folder> Folders { get; set; }

public DbSet<File> Files { get; set; }

public ApplicationContext(DbContextOptions<ApplicationContext>

options) : base(options) {}

protected override void OnModelCreating(

ModelBuilder modelBuilder){}

}

В самых простых случаях миграция данных проходит без каких либо проблем. В более сложных, необходимо помочь утилите понять каким образом сопоставлять сущности. Это можно сделать в методе “OnModelCreating”.

Листинг 2.1.5.5 – Реализация метода “OnModelCreating”.

void OnModelCreating( ModelBuilder modelBuilder) {

modelBuilder.Entity<Folder>()

.Property(f => f.ParentId).IsRequired(false);

modelBuilder.Entity<Archive>()

.Property(a => a.FolderId).IsRequired(false);

modelBuilder.Entity<Folder>()

.HasMany(f => f.Files).WithOne(f => f.Folder)

.HasForeignKey(f => f.FolderId);

modelBuilder.Entity<Folder>()

.HasMany(f => f.Folders).WithOne(f => f.Parent)

.HasForeignKey(f => f.ParentId);

modelBuilder.Entity<Folder>()

.HasOne(f => f.Archive).WithMany()

.HasForeignKey(f=>f.ArchiveId);

modelBuilder.Entity<Archive>()

.HasOne<Folder>(a => a.Folder).WithOne()

.HasForeignKey<Archive>(a => a.FolderId);

}

В данном методе описывается какие связи должны быть между сущностями.

Так же иногда удобно сделать дополнительные модели для передачи данных между клиентом и сервером, такие сущности называются DTO(Data Transfer Object).

Класс для передачи информации о файле.

Листинг 2.1.5.6 – Сущность “FileDTO”.

public class FileDTO {

public int Id { get; set; }

public string Name { get; set; }

public string Text { get; set; }

public int FolderId { get; set; }

}

Класс для передачи информации о каталоге.

Листинг 2.1.5.7 – Сущность “FolderDTO”.

public class FolderDTO {

public int Id { get; set; }

public int? ParentId { get; set; }

public string Name { get; set; }

public IEnumerable<FolderDTO> Folders { get; set; }

public IEnumerable<FileDTO> Files { get; set; }

}

Класс, который будет использоваться для проверки доступа.

Листинг 2.1.5.8 – Сущность “CartAccess”.

public class CartAccess {

public int Id { get; set; }

public string Key { get; set; }

public Archive Archive { get; set; }

}

Класс “ FileCartAccess ” унаследованный от “CartAccess”, будет проверят возможно ли получить доступ к файлу при таком ключе. Данный класс производит валидацию модели. Существует несколько подходов в которых логику валидации можно выносить в отдельные классы, но было принято решение использовать самовалидацию модели. В методе валидации модели используются интерфейсы, которые будут описаны в следующих главах.

Листинг 2.1.5.9 – Сущность “FileCartAccess”.

public class FileCartAccess : CartAccess, IValidatableObject {

public File File { get; set; }

public IEnumerable<ValidationResult>

Validate(ValidationContext validationContext){}

}

Листинг 2.1.5.10 – Реализация метода “Validate” класса “FileCartAccess”.

public IEnumerable<ValidationResult>

Validate(ValidationContext validationContext) {

List<ValidationResult> errors

= new List<ValidationResult>();

var archiveService =

validationContext

.GetService(typeof(IArchiveService))as IArchiveService;

var fileService =

validationContext

.GetService(typeof(IFileService)) as IFileService;

Archive = Archive ?? archiveService.Find(Key);

File = File ?? fileService.Find(Id);

if (File == null || Archive == null

|| File.Folder?.ArchiveId != Archive.Id) {

errors.Add(new ValidationResult("Нет доступа к файлу"));

}

return errors;

}

Класс “FolderCartAccess” имеет то же предназначение, что и “FileCartAccess ”. Исключением является проверка доступа к каталогу. Метод “Validate” имеет аналогичную логику, при этом действия производятся с интерфейсом “IFolderService”.

Листинг 2.1.3.1 – Сущность “FolderCartAccess”.

public class FolderCartAccess : CartAccess,IValidatableObject {

public Folder Folder { get; set; }

public IEnumerable<ValidationResult>

Validate(ValidationContext validationContext)

}

### Создание сервисов

Одно из нововведений в ASP.NET Core-это поддержка внедрения зависимостей, которое доступно сразу “из коробки”, и не нужно прибегать к использованию сторонних инструментов. Отталкиваясь от этого, было принято решение, разработать сервисы для взаимодействия с контекстом данных. Это дает возможность, без какого-либо труда, изменить приложение для использования другого источника данных.

Были разработаны следующие интерфейсы.

Интерфейс “IArchiveService” представляет сервис для управления архивом. Метод “Uncompress” распаковывает архив в определенное место. Метод “Compress” запаковывает архив и возвращает поток, в который записан архив.

Листинг 2.1.6.1 – Определение интерфейса “ IArchiveManager ”.

public interface IArchiveManager {

Archive Uncompress(string pathArchive, string directory);

Stream Compress(Archive archive);

}

Реализацией данного интерфейса является класс “ArchiveManagerImpl”.В данном классе используется библиотека “CharpCompress”.

Листинг 2.1.6.2 – Сущность “ ArchiveManagerImpl”.

public class ArchiveManagerImpl : IArchiveManager {

IArchiveService archiveService;

IFolderService folderService;

IFileService fileService;

public ArchiveManagerImpl(IArchiveService archiveService,

IFolderService folderService,

IFileService fileService) {

this.archiveService = archiveService;

this.folderService = folderService;

this.fileService = fileService;

}

public IO.Stream Compress(Archive archive) {

var zip = ArchiveFactory.Create(ArchiveType.Zip);

zip.AddAllFromDirectory(archive.Folder.Path);

IO.MemoryStream ms = new IO.MemoryStream();

zip.SaveTo(ms, CompressionType.Deflate);

ms.Position = 0;

return ms;

}  
 public Archive Uncompress(

string pathArchive, string directory) {}

}

Метод “Uncompress” содержит довольно таки сложную логику. Так как должна быть возможность получения доступа к отдельным элементам архива. При разработке задаче были приняты следующие решения (в представленных ниже пунктах речь идет о сущностях базы данных):

* полный путь к архиву на сервере прописан только в корневом каталоге куда будет помещен архив;
* в остальных каталогах хранится путь относительно корневого каталога;
* каждый каталог имеет идентификатор архива из которого был извлечен. Это позволит без лишних запросов к базе данных, проверить возможность получения доступа к данному каталогу;
* в файле хранится имя и идентификатор каталога в котором расположен файл;

Во время работы метода “Uncompress” данные попутно добавляются в базу данных. Это было сделано для того, чтоб дополнительно не тратить время на исследование каталогов и файлов, и занесения в базу данных отдельно.

Листинг 2.1.6.3 – Реализация метода “Uncompress”

public Archive Uncompress(string pathArchive, string directory) {

var archive = ArchiveFactory.Open(pathArchive);

var result = new Archive();

result = archiveService.Create(result);

var dir = new Folder { Path = "", Archive = result };

result.Folder = dir;

dir = folderService.Create(dir);

foreach (var entry

in archive.Entries.Where(e => !e.IsDirectory)) {

var split = entry.Key.Split('\\');

var curDir = dir;

var length = split.Length; int i = 0;

while (length > 1) {

var nextDir = curDir.Folders.FirstOrDefault(

d => String.Equals(split[i], d.Name));

if (nextDir == null) {

nextDir = new Folder {

Path = IO.Path.Combine(curDir.Path, split[i]),

Archive = result };

curDir.Folders.Add(nextDir);

nextDir.Parent = curDir;

folderService.Create(nextDir);

}

curDir = nextDir;

length--;

i++;}

entry.WriteToDirectory(directory,

new ExtractionOptions() {

ExtractFullPath = true,

Overwrite = true });

var curFile = new File { Folder = curDir, Name = split[i] };

curDir.Files.Add(curFile);

fileService.Create(curFile);}

archive.Dispose();

dir.Path = directory;

folderService.Update(dir);

return result;}

Следующие сервисы предназначены для управления сущностями базы данных.

Интерфейс “IArchiveService” представляет сервис управления загруженными архивами.

Листинг 2.1.6.4 – Определение “ IArchiveService ”

public interface IArchiveService {

Archive Create(Archive archive);

Archive Update(Archive archive);

void Delete(Archive archive);

Archive Find(string key);

}

Интерфейс “IFileService” представляет сервис управления файлами.

Листинг 2.1.6.5 – Определение “IFileService”

public interface IFileService {

File Create(File file);

File Update(File file);

void Delete(File file);

File Find(int id);

}

Интерфейс “IFolderService” представляет сервис управления каталогами.

Листинг 2.1.6.6– Определение “IFolderService”

public interface IFolderService {

Folder Create(Folder folder);

Folder Update(Folder folder);

void Delete(Folder folder);

FolderDTO GetEntry(int id);

Folder Find(int id);

}

Имеет смысл рассмотреть реализацию одного из интерфейсов, так как все реализации очень похожи за исключение мелких деталей.

Класс “FileServiceImpl” реализует интерфейс “IFileService”. Он получает в конструкторе контекст и уже используя его управляет сущностью “File”.

Листинг 2.1.6.7 – Определение “IFolderService”

public class FileServiceImpl : IFileService {

ApplicationContext context;

public FileServiceImpl(ApplicationContext context) {

this.context = context;

}

public void Delete(File file) {

File fileDB = context.Files

.FirstOrDefault(f => f.Id == file.Id);

if(fileDB != null) {

context.Files.Remove(fileDB);

context.SaveChanges();

}

}

public File Find(int id) {

return context.Files

.Include(f=>f.Folder)

.FirstOrDefault(f=>f.Id == id);

}

public File Create(File file) {

context.Files.Add(file);

context.SaveChanges();

return file;

}

public File Update(File file) {

context.Entry(file).State = EntityState.Modified;

context.SaveChanges();

return file;

}

}

### Добавление контроллеров

В архитектуре MVC входящие запросы обрабатываются контроллерами. В ASP.NET MVC контроллеры являются простыми C#. Каждый открытый метод в контроллере известен как метод действия, то есть его можно вызвать из “Интернет” через некоторые URL, чтобы выполнить действие.

Приложение будет включать в себя следующие контроллеры:

* ArchivesControlles. Предоставляет доступ к загрузке и скачиванию архивов. Данный контроллер обрабатывает запросы по адресу “api/archives”.
* FilesController. Предоставляет доступ к файлам архива. Обрабатывает запросы по адресу “api/archives/{key}/files”, где key – это идентификатор архива.
* FoldersController. Предоставляет доступ к каталогам архива. Обрабатывает запросы по адресу “api/archives/{key}/folders”, где key – это идентификатор архива.

Достаточно рассмотреть контроллер “ArchiveController”, так как в остальных контроллерах используются аналогичные подходы.

Чтобы добавить контроллер в проект, щелчком правой кнопкой мыши по папке “Controllers” и выбрать пункт “Добавить” во всплывающем меню. Далее следует указать имя контроллера.

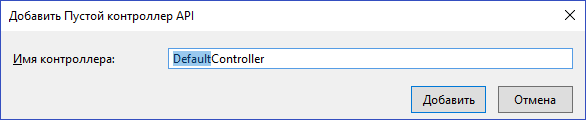


Рисунок 2.1.7.1 – Указание имени контроллера

После нажатия кнопки «Добавить» Visual Studio создаст новый файл с C# кодом с названием “ArchiveController.cs” в папке “Controllers” и откроет его для редактирования.

Листинг 2.1.7.1 – Содержимое файла ”ArchiveControlle.cs”

[Route("api/archives")]

public class ArchivesController : Controller {

IHostingEnvironment appEnv;

IArchiveManager archiveManager;

IArchiveService archiveService;

IFolderService folderService;

public ArchivesController(IHostingEnvironment appEnv,

IArchiveManager archiveManager,

IArchiveService archiveService,

IFolderService folderService) {

this.appEnv = appEnv;

this.archiveManager = archiveManager;

this.archiveService = archiveService;

this.folderService = folderService;

}

[HttpPost]

public async Task<IActionResult> Post(IFormFile file) {…}

[HttpGet("{key}")]

public IActionResult GetEntry(string key) {…}

[HttpGet("{key}/download")]

public IActionResult Download(string key) {…}

}

Существуют различные способы настройки роутинга в приложении. В данном случае он настраивается с использованием атрибута “Route”. Данный атрибут принимает шаблон, с которым будет сравниваться строка запроса. В фигурных скобках можно указывать ключевые слова, которые сопоставятся с реальным значением из строки запроса.

Так же можно наложить ограничения на методы действия, указывая какой тип HTTP запроса он будет обрабатывать. C помощью таких атрибутов возможно дополнить карту маршрутизации. В данном случае, некоторые действия, будут соответствовать запросу, только если в строке запроса присутствует идентификатор архива.

Метод “GetEntry” отправляет клиенту содержимое корневого каталога архива.

Листинг 2.1.7.2– Реализация метода действия“GetEntry”

[HttpGet("{key}")]

public IActionResult GetEntry(string key) {

int? folderId = archiveService.Find(key)?.Folder.Id;

if (folderId != null) {

return Ok(folderService.GetEntry(folderId.Value));

}

return NotFound();

}

При обработке запроса, на определенной стадии, происходит привязка данных. В данном случае платформа ASP.NET передаст в метод действия значение “key”, которое получится из строки запроса.

Листинг 2.1.7.3 – Реализация метода действия“ Download ”

[HttpGet("{key}/download")]

public IActionResult Download(string key) {

Archive archive = archiveService.Find(key);

IO.Stream fs = archiveManager.Compress(archive);

return File(fs, "application/zip");

}

Метод действия “Post” реализует загрузку архива на сервер и отправки клиенту идентификатора загруженного архива клиенту.

Листинг 2.1.7.4 – Реализация метода действия“ Post ”

[HttpPost]

public async Task<IActionResult> Post(IFormFile file) {

if (file == null) {

return BadRequest();

}

//Идентификатор данного архива

string guid = Guid.NewGuid().ToString();

// путь к папке Temp

string pathDirTemp

= IO.Path.Combine(appEnv.ContentRootPath, "Temp");

string pathArchive = IO.Path.Combine(pathDirTemp, guid);

// сохраняем файл в папку Temp

IO.Directory.CreateDirectory(pathDirTemp);

using (var fileStream = IO.File.Create(pathArchive)) {

await file.CopyToAsync(fileStream);

}

//путь к каталогу Archives

var dirArchive

= IO.Path.Combine(appEnv.ContentRootPath, "Archives", guid);

Archive archive = null;

try {

//получение структуры архива

archive = archiveManager.Uncompress(pathArchive, dirArchive);

} catch {

return BadRequest("Не удалось разобрать архив");

}

//Удаление временного файла

IO.File.Delete(pathArchive);

archive.Key = guid;

archive = archiveService.Update(archive);

return Ok(archive.Key);

}

### Конфигурация приложения

За конфигурацию приложения отвечает класс “Startup”. В нем указываются настройки для внедрения зависимостей, конвейера обработки запросов.

Так же необходимо добавить в проект файл конфигурации “appsettings.json”, в котором будут храниться различные настройки для приложения.

Листинг 2.1.8.1 – Содержимое файла “appsettings.json”

{

"ConnectionStrings": {

"DefaultConnection": "Data Source=ArchiveManager.db"

}

}

Класс “Sturtup” принимает в конструкторе реализацию интерфейса “IConfiguration”, и данный класс будет содержать в себе данные из файла “appsettings.json”.

Листинг 2.1.8.2 – Класс “Startup”

public class Startup {

public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration) {

Configuration = configuration;

}

public IServiceProvider ConfigureServices(

IServiceCollection services){…}

public void Configure(

IApplicationBuilder app, IHostingEnvironment env){…}

}

Метод “ConfigureServices” вызывается платформой для настройки внедрения зависимостей. В данном методе сопоставляются интерфейсы и их реализации.

Листинг 2.1.8.3 – Метод “ConfigureServices”

public IServiceProvider ConfigureServices(

IServiceCollection services) {

string connection =

Configuration.GetConnectionString("DefaultConnection");

services.AddDbContext<ApplicationContext>(

options => options.UseSqlite(connection));

services.AddMvc();

services.AddSingleton<IArchiveManager, ArchiveManagerImpl>();

services.AddTransient<IFileService, FileServiceImpl>();

services.AddTransient<IFolderService, FolderServiceImpl>();

services.AddTransient<IArchiveService, ArchiveServiceImpl>();

return services.BuildServiceProvider();}

Метод “Configure” вызывается для настройки конвейера обработки запроса.

Листинг 2.1.8.4 – Метод “ConfigureServices”

public void Configure(

IApplicationBuilder app, IHostingEnvironment env) {

app.UseStaticFiles();

app.UseMvc();

}

### Миграция данных

После написание основного функционала и компиляции проекта. Необходимо синхронизировать доменную модель с базой данных.

Необходимо выполнить следующие действия:

* открыть “Консоль диспетчера пакетов”
* выполнить команду “Add-Migrations Initial”.Где “Initial” - это имя миграции.
* выполнить команду “Update-Database”

После выполнения данных действий, в базе данных появятся соответствующие таблицы и ограничения.

## Клиентская часть web-приложения “ArchiveManager”

### Постановка задачи

Необходимо разработать приложение с использованием библиотеки React. Данное приложение должно предоставлять следующий функционал:

* загружать архив на сервер
* просмотр содержимого архива

### Создание проекта

Разработку клиента было принято решение сделать в том же проекте, где и серверное приложение. Это дает такие преимущества как, разработка единого приложения, удобство отладки, браузер не будет прибегать к кросс-браузерным запросам.

В качестве сборщика проекта будет использоваться Webpack. Чтобы его сконфигурировать, в проект нужно добавить файл “webpack.config.js”.

В данном файле можно выделить пару важных моментов.

* “entry” указывает точку входа с которой нужно начинать сборку;
* “output” указывает место куда будет записан собранный файл;
* в “module” указывается какие расширения файлов каким загрузчиком должны обрабатываться;

Листинг 2.2.2.1 – Содержимое файла “webpack.config.js”

const path = require('path');

const bundleFolder = "./wwwroot/bundle/";

const srcFolder = "./WebApp/"

module.exports = {

mode: 'development',

entry: [

srcFolder + "index.jsx"

],

devtool: "source-map",

output: {

filename: "bundle.js",

publicPath: 'bundle/',

path: path.resolve(\_\_dirname, bundleFolder)

},

module: {

rules: [

{

test: /\.jsx$/,

exclude: /(node\_modules)/,

loader: "babel-loader",

query: {

presets: ["env", "react"]

}

}

]

},

plugins: [

],

resolve: {

modules: ['node\_modules', srcFolder],

extensions: ['.js', '.jsx'],

mainFiles: ["index"]

}

}; (

Так же необходимо включить в классе “Startup” использование Webpack, чтобы сервер запускал сборку клиента.

Листинг 2.2.2.2 – Реализация метода ”Configure” в классе “Startup”

public void Configure(

IApplicationBuilder app, IHostingEnvironment env) {

if (env.IsDevelopment()) {

app.UseWebpackDevMiddleware();

}

app.UseStaticFiles();

app.UseMvc((routes) => {

routes.MapSpaFallbackRoute("spa",

new { controller = "Home", action = "Index" });

});

}

Также, был добавлен маршрут, чтобы все запросы перенаправлялись на один контроллер, и можно было воспользоваться системой маршрутизации React.

### Реализация логики приложения

Для разделения логики обработки данных и отображение, использовалась библиотека Reflux. Основу Reflux составляют три компонента:

* действия, набор применяемых действий в приложении;
* представление, в данной роли выступают компоненты React;
* хранилища, содержат данные и логику приложения.

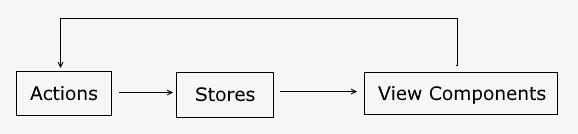


Рисунок 2.2.3.1 – Взаимодействие между компонентами Reflux

Хранилища подписываются на действия, связываются с представлениями. При активации действий хранилища выполняют логику и обновляют представление.

Листинг 2.2.3.1 – Содержимое файла “ArchiveActions.js”

import Reflux from "reflux";

var ArchiveActions = Reflux.createActions(

[

"Upload"

]

);

export default ArchiveActions;

Листинг 2.2.3.1 – Содержимое файла “FolderAction.js”

import Reflux from "reflux";

var FolderAction = Reflux.createActions(

[

"Load","StartLoad"

]

);

export default FolderAction;

Хранилище “AchiveStore” содержит логику для загрузки архива на сервер.

В листинге 2.2.3.2 опущена реализация метода “onUpload”, так как реализацией является обычная отправка запроса на сервер и обновление поля “key”.

Листинг 2.2.3.2 – Содержимое файла “AchiveStore.js”

import Reflux from "reflux";

import $ from "jquery";

import ArchiveActions from "actions/ArchiveActions";

import { urls } from "tools/const";

class AchiveStore extends Reflux.Store {

constructor() {

super();

this.state = { key:null };

this.listenTo(ArchiveActions.Upload, this.onUpload);

}

onUpload(data) {…}

}

export default Reflux.initStore(AchiveStore);

Класс “FoldersStore” содержит логику для загрузки содержимого каталога.

Листинг 2.2.3.3 – Содержимое файла “ FoldersStore.js”

import Reflux from "reflux";

import $ from "jquery";

import FolderActions from "actions/FolderActions";

import { urls } from "tools/const";

class FoldersStore extends Reflux.Store {

constructor() {

super();

this.state = {

folder: {

folders: [], files: []

}

};

this.listenTo(FolderActions.StartLoad, this.StartLoad);

this.listenTo(FolderActions.Load, this.Load);

}

StartLoad(key) {…}

Load(id) {…}

}

Достаточно рассмотреть одно представление, так как остальные строятся на таких же принципах.

Листинг 2.1.3.1 – Содержимое файла “Main/index.js”

import Reflux from 'reflux';

import React from 'react';

import ArchiveStore from 'stores/ArchiveStore';

import ArchiveActions from "actions/ArchiveActions";

import { Redirect } from 'react-router-dom';

export default class Main extends Reflux.Component {

constructor() {

super();

this.store = ArchiveStore;

this.onSubmit = this.onSubmit.bind(this);

}

onSubmit(e) {

e.preventDefault();

const file = this.file.files[0];

if (file) {

var formData = new FormData();

formData.append("file", file, file.name);

ArchiveActions.Upload(formData);

}

}

render() {

const { key } = this.state;

return (

<div>

{key && <Redirect to={`/${key}`} />}

<form onSubmit={this.onSubmit}>

<input type="file" ref={e => this.file = e} />

<input type="submit" value="Загрузить" />

</form>

</div>

);

}

}

Как видно, компонент привязывается к хранилищу “ArchiveStore” через свойство “store”. Это означает, что после хранилище после выполнения логики обновит данный компонент. Так же данный компонент вызывает действие “ArchiveActions.Upload”, которое обработает хранилище. А после выполнения действия хранилище обновит данные, и сам компонент перенаправит на другую страницу.

# МАСТЕР-КЛАСС

План мастер-класса:

1. Что такое ASP.NET?
2. Развитие ASP.NET.
3. Что такое ASP.NET Core?
4. Структура проекта.
5. Демонстрация приложения.
6. Рекомендации по разработке web-приложений.

# ЗАКЛЮЧЕНИЕ

Итогом проделанной работы является написание приложения “ArchiveManager” на ASP.NET Core и React.

В ходе прохождение производственной практики были получены углубленные знания в разработке web-приложений с использованием платформы ASP.NET Core, разработке одностраничных web-приложений с использованием JS-библиотеки React, использование пакетного менеджера для загрузки необходимых пакетов, создание гибкой структуры проекта для возможности дальнейшего расширения функциональности приложения.

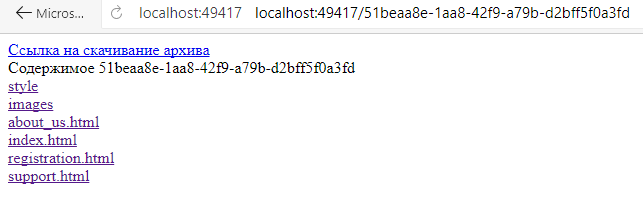
Поставленные задачи на преддепломную практику были выполнены, что позволило закрепить и углубить знания, полученные в университете, их практическое применение под руководством научных руководителей и получить большой опыт в квалификации программиста.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Адам Фримен, ASP.NET MVC 5 с примерами на C# 5.0 для профессионалов. — 5-е. — Санкт-Петербург, 2015. — С.736— ISBN 978-5-8459-1911-3
2. ASP.NET Core - новая эпоха в развитии ASP.NET [Электронный ресурс]. - Режим доступа: https://metanit.com/sharp/aspnet5/1.1.php. Дата доступа: 15.03.2018.
3. ASP.NET MVC | The ASP.NET Site [Электронный ресурс]. - Режим доступа: <https://www.asp.net/mvc> . Дата доступа: 15.03.2018.
4. Андерс Хейлмберг, Мэдс Торгерсен, Язык программирования C#. Классика Computers Science. - 4-е. - Санкт-Петербург, 2011. — С. 784— ISBN 978-5-459-00283-6

# ПРИЛОЖЕНИЕ А

**Пример работы web-приложения**



# ПРИЛОЖЕНИЕ Б

**Содержимое дискового носителя**

1. \ArchiveManager –web-приложения “ArchiveManager”.
2. Отчет.docx - пояснительная записка к практической работе