Современные платформы прикладной разработки

ПЛАТФОРМА ASP.NET, СТРУКТУРА ПРИЛОЖЕНИЯ

Freeman Adam - Pro ASP.NET Core 6: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages – Apress, 2022, 1267p.

https://learn.microsoft.com/ru-ru/aspnet/core/?view=aspnetcore-7.0

Платформа ASP.NET, структура приложения

Тема лекции

Платформа ASP.NET

Приложение ASP.NET Core MVC

Цели и задачи

Знакомство с платформой ASP.Net.

Знакомство с шаблоном проектирования MVC

Знакомство с реализацией шаблона MVC в ASP.NET

Знакомство с приложением ASP.NET Core MVC

Технология ASP.NET

ASP.NET (**Active Server Pages**) — технология создания веб-приложений и веб-сервисов от компании Майкрософт. Она является составной частью платформы Microsoft .NET

ASP.NET Core выполняется в среде Common Language Runtime (CLR), которая является основой всех приложений Microsoft .NET Core. Код для ASP.NET, может использовать практически любые языки программирования, входящие в комплект .NET Core

Технология ASP.NET

ASP.NET была выпущена в 2002г., и была представлена в виде технологии Web Forms.

В Web Forms сайт представляется набором страниц, где каждая страница на сайте представлена в виде физического файла (называемых Web Form) и доступна с помощью имени этого файла.

Технология ASP.NET MVC

ШАБЛОН ПРОЕКТИРОВАНИЯ MVC

Шаблон проектирования MVC

MVC (Model-view-controller, «модель-представление-контроллер») — схема использования нескольких шаблонов проектирования, с помощью которых модель данных приложения, пользовательский интерфейс и взаимодействие с пользователем разделены на три отдельных компонента таким образом, чтобы модификация одного из компонентов оказывала минимальное воздействие на остальные.

Шаблон проектирования MVC

Основная цель применения концепции MVC состоит в разделении бизнес-логики (модели) от её визуализации (представления, вида).

За счет такого разделения повышается возможность повторного использования. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть те же самые данные одновременно в различных контекстах и/или с различных точек зрения.

Преимущества концепции MVC

- к одной модели можно присоединить несколько представлений, при этом не затрагивая реализацию модели.
- не затрагивая реализацию представлений, можно изменить реакции на действия пользователя (нажатие мышью на кнопке, ввод данных), для этого достаточно использовать другой контроллер.
- возможно добиться раздельного (независимого) проектирования бизнес-логики (модели) и интерфейса (представления)
- модульное тестирование

Основные понятия шаблона MVC

Модель: набор классов, предоставляющих данные и правила бизнес-логики для управления этими данными.

Представление: отвечает за визуальное отображение модели.

Контроллер: управляет логикой приложения и является координатором между представлением и моделью

Схема реализации шаблона MVC

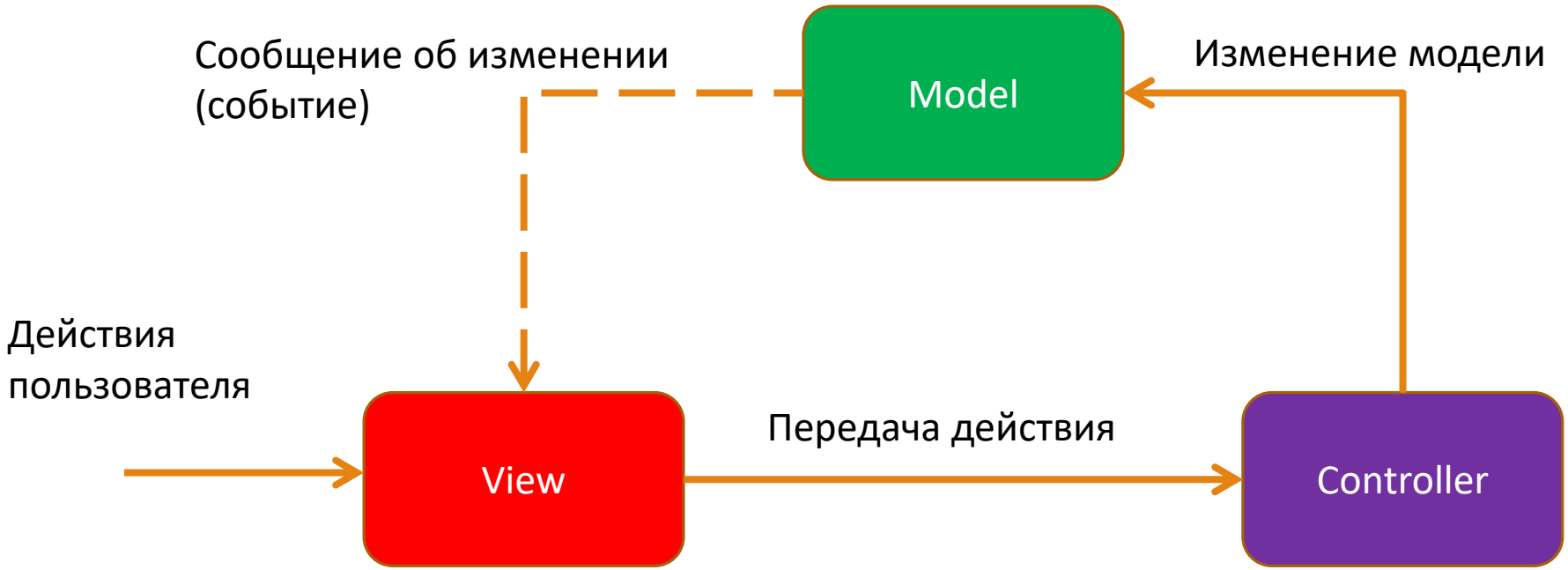
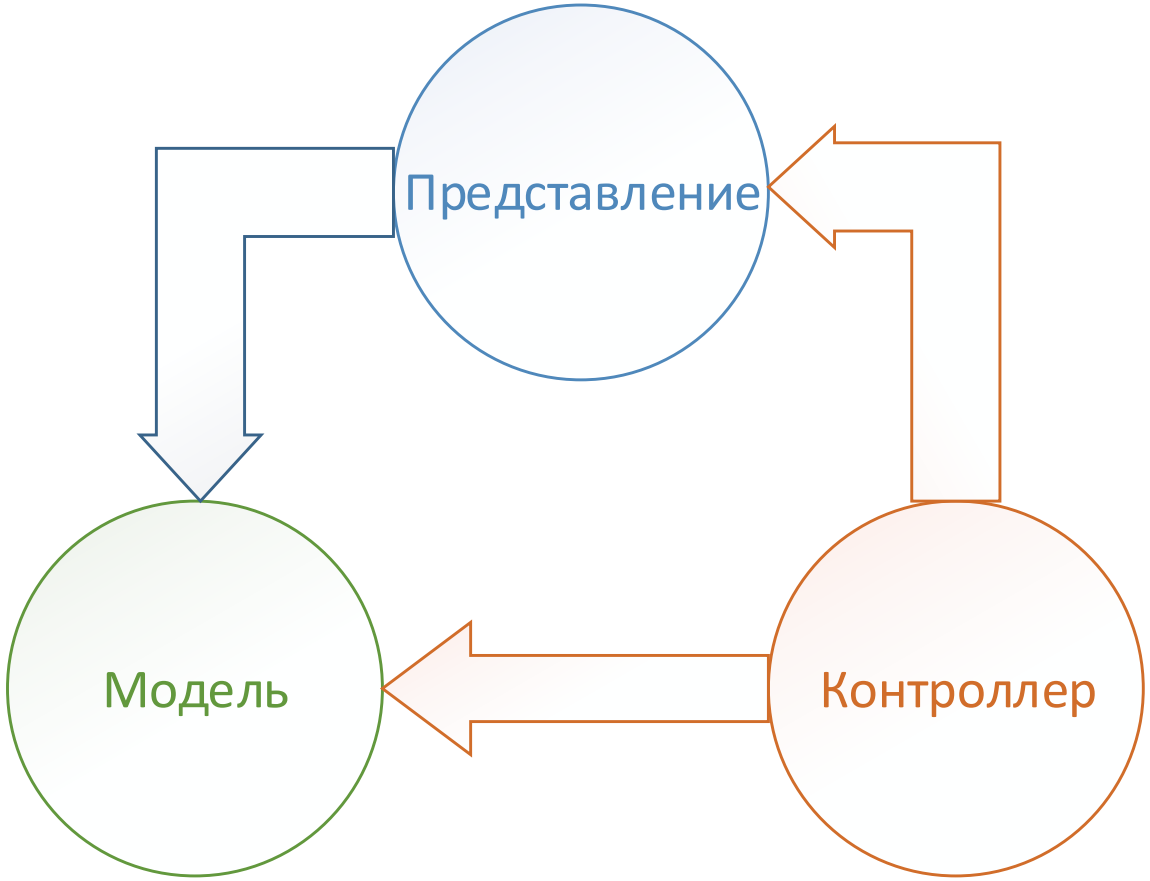


Схема реализации шаблона MVC



Технология ASP.NET MVC

ШАБЛОН ПРОЕКТИРОВАНИЯ MVC И ASP.NET

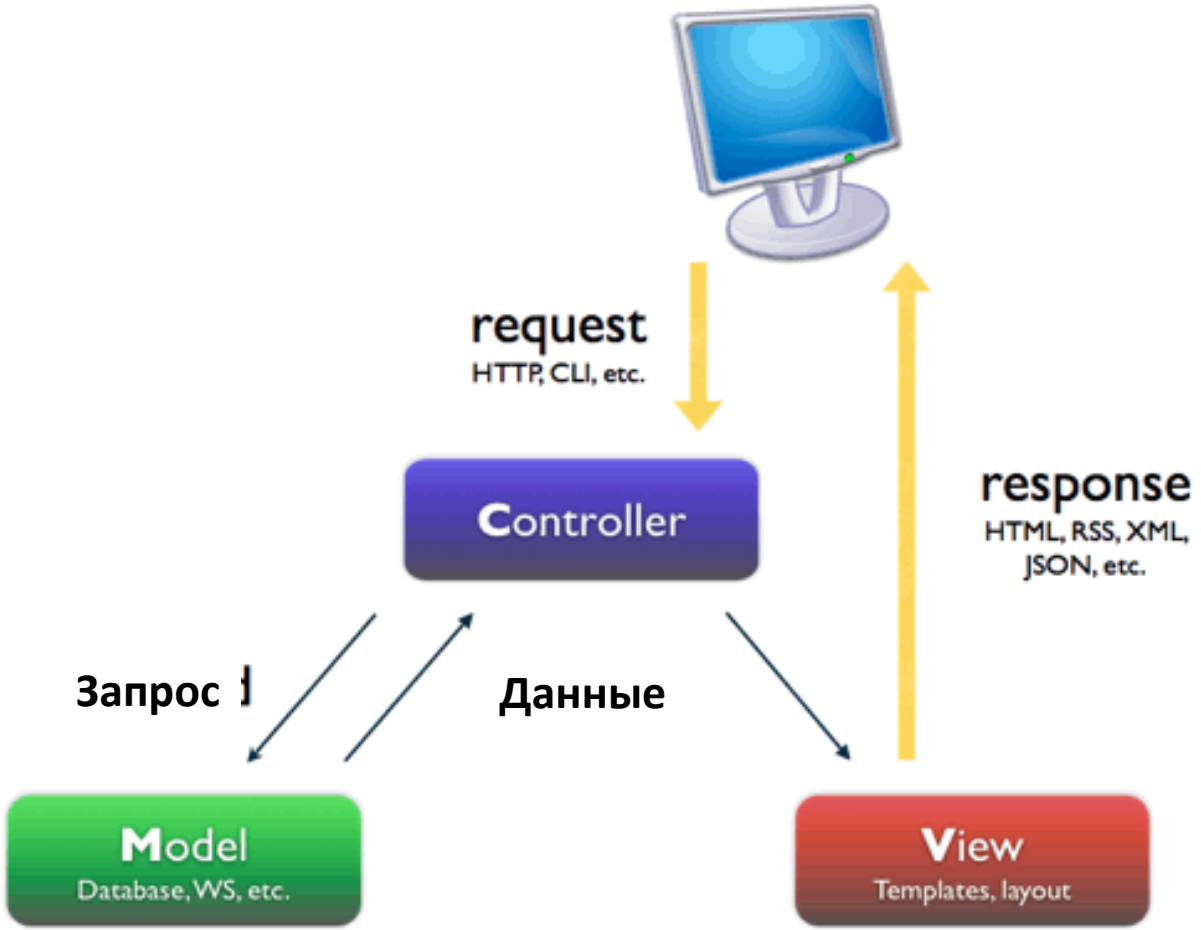
MVC в ASP.NET

Модель: набор классов, инкапсулирующих данные, хранящиеся в базе данных, или передаваемых представлению, а также код, манипулирующий этими данными.

Представление: шаблон для динамического генерирования HTML-кода

Контроллер: класс, управляющий взаимодействием представления и модели. Он принимает запрос от клиента, общается с моделью и решает, какое представление нужно отправить клиенту в данный момент.

Схема реализации шаблона MVC в ASP.NET



Задачи модели

- Хранить данные предметной области
- Содержать логику для чтения, добавления, удаления и редактирования данных предметной области
- Предоставлять простой интерфейс для работы с данными

Модель НЕ ДОЛЖНА

- Показывать детали доступа к данным
- Содержать логику для преобразования данных, основываясь на взаимодействии с пользователем (это задача контроллера)
- Содержать логику для отображения данных пользователю (это задача представления)

Задачи контроллера

Содержать методы для манипулирования данными на основании действий пользователя

Контроллер НЕ ДОЛЖЕН

- Содержать логику, управляющую отображением данных (это задача представления)
- Содержать логику, управляющую сохраняемыми данными (это задача модели)

Задачи представления

Содержать логику и разметку для отображения данных пользователю

Представление НЕ ДОЛЖНО

- Содержать сложную логику (это лучше предоставить контроллеру)
- Содержать логику для сохранения, создания или изменения доменной модели

Подготовка к работе

ОБЩИЕ СВЕДЕНИЯ

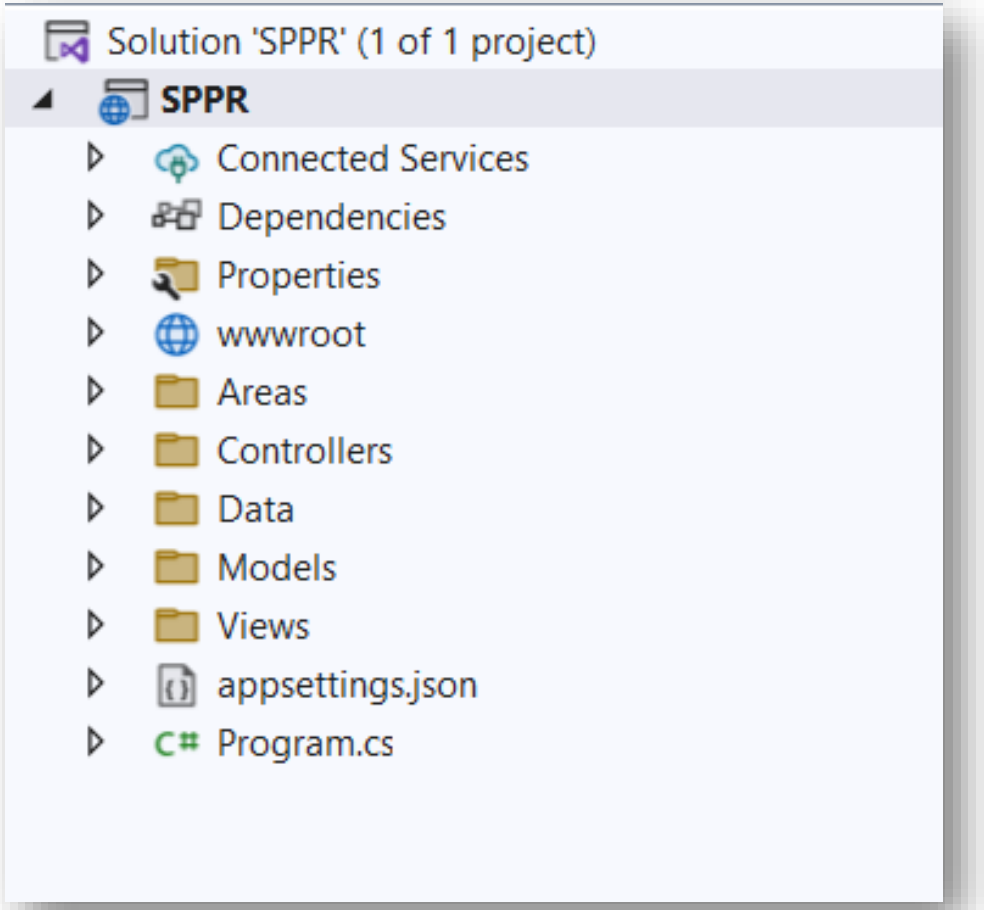
Для работы с .NET Core нужно:

1. Visual Studio 2022 Community
   или Visual Studio Code
   или Rider
2. Установить .NET Core SDK

Структура приложения

ОБЩИЕ СВЕДЕНИЯ

Структура приложения



Структура приложения

Приложение ASP.NET Core – это обычное консольное приложение, которое создает веб сервер в методе main

Content Root

Content Root - это основной путь к любому контенту, используемому в приложении, например, его представления и веб контент.

По умолчанию Content Root - это то же самое, что и основной путь приложения для выполняемого хостинга

Web Root

Web Root - это директория для открытых ***статических*** ресурсов, таких как файлов css, js и файлов изображений.

Связующее ПО статических файлов по умолчанию отрабатывает файлы **только** из этой директории (и поддиректорий).

Путем директории является *<content root>/**wwwroot***

В пустом проекте данная папка отсутствует.

Структура приложения

Connected Services: подключенные сервисы из Azure или сервисы WCF и др.

Dependencies: используемые в проект пакеты и библиотеки

wwwroot: этот узел (на жестком диске ему соответствует одноименная папка) предназначен для хранения **статических** файлов - изображений, скриптов javascript, файлов css и т.д., которые используются приложением.).

Структура приложения

Program.cs: главный файл приложения, с которого и начинается его выполнение.

Структура приложения

В классе **Program** определяется конвейер (pipeline – дословно трубопровод) обработки запросов (конвейер Middleware), а также задается конфигурация сервисов, необходимых для работы приложения

Конвейер Middleware

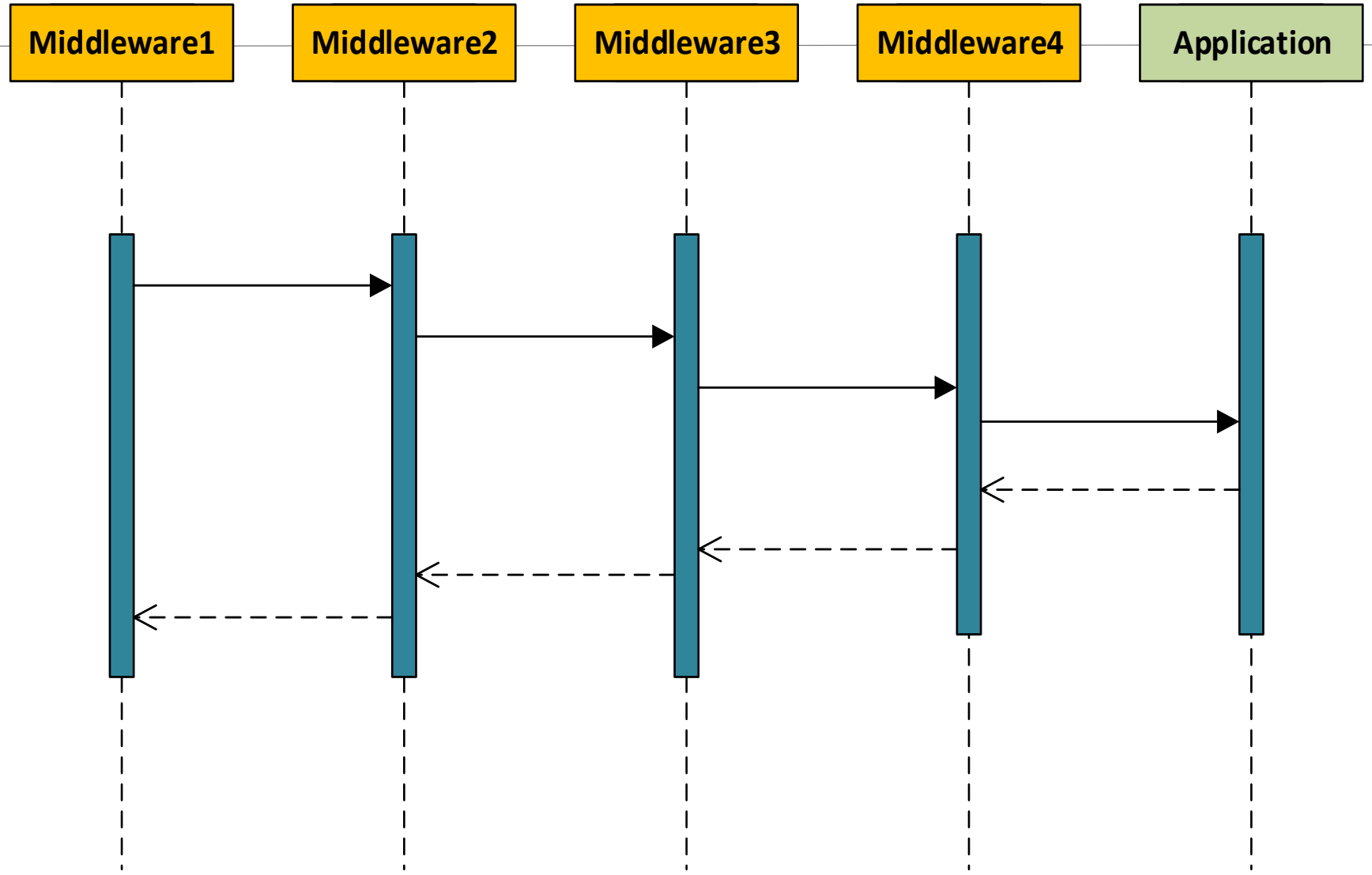
Конвейер Middleware

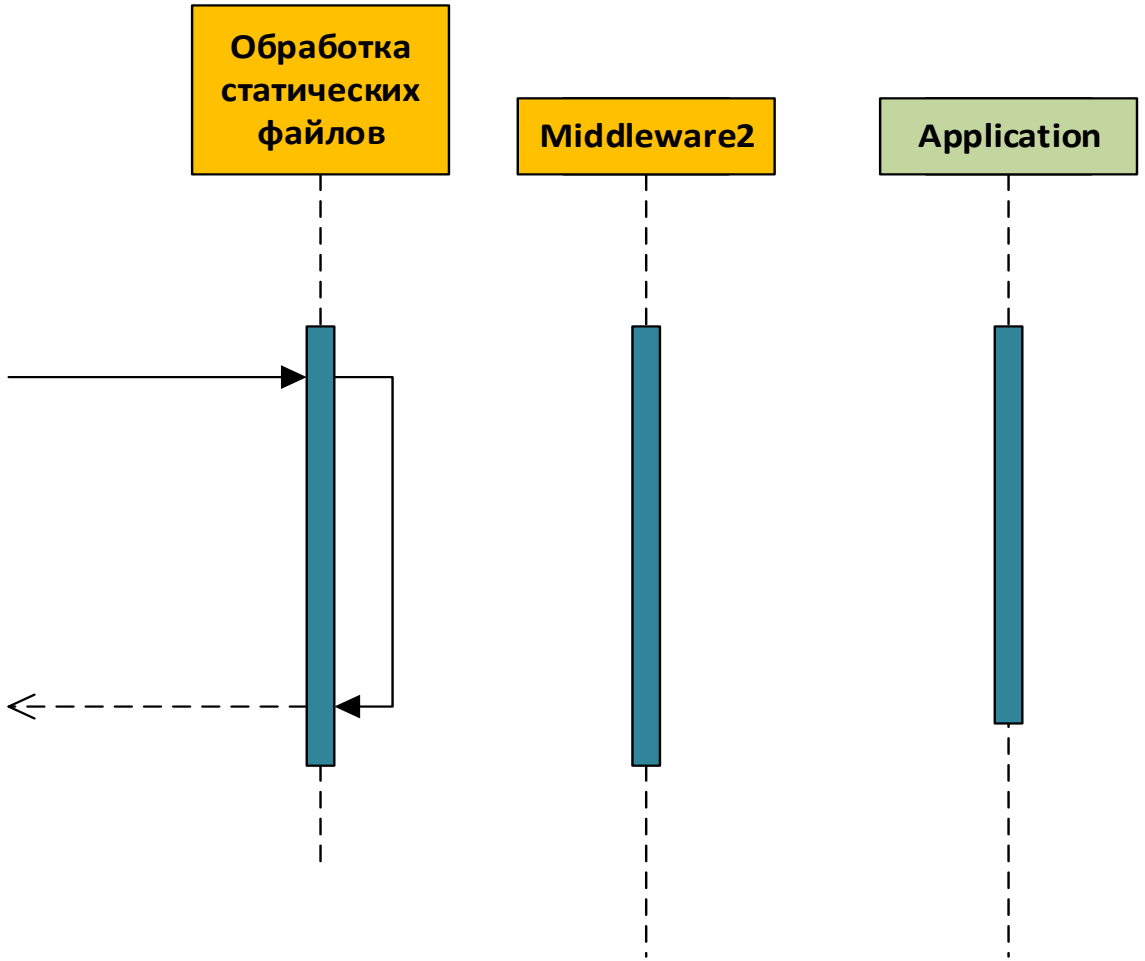
Middleware - это программное обеспечение, которое собирается в конвейер приложения для обработки запросов и ответов.

Каждый компонент Middleware отвечает за свою узкую часть задачи.

Каждый компонент:

- ✓ выбирает, следует ли передавать запрос следующему компоненту в конвейере.
- ✓ может выполнять работу до и после следующего компонента в конвейере.


Middleware1
Middleware2
Middleware3
Middleware4
Application


Обработка статических файлов
Middleware2
Application

Класс Program

Файл Program.cs

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));

. . .

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseStaticFiles();

. . .

app.Run();
```

Файл Program.cs

В классе Program определяется конвейер обработки запросов, а также задается конфигурация сервисов, необходимых для работы приложения.

appsettings.json

Конфигурация приложения ASP.NET Core

Конфигурация приложения в ASP.NET Core выполняется с использованием одного или нескольких поставщиков конфигурации.

Конфигурация приложения ASP.NET Core

Поставщики конфигурации считывают данные конфигурации из пар ключ-значение, используя различные источники конфигурации:

- Файлы настроек, например appsettings.json.
- Переменные среды
- Хранилище ключей Azure
- Конфигурация приложения Azure
- Аргументы командной строки
- Пользовательские поставщики, установленные или созданные
- Файлы каталогов
- Объекты .NET в памяти

Конфигурация приложения ASP.NET Core

По умолчанию используется файл конфигурации **appsettings.json**. Данный файл содержит параметы в формате JSON. Параметры хранятся в виде пары ключ-значение

Пример файла appsettings.json

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data source = Test.db"
  },
  "ApiData": {
    "Url": "https://somedata.by/api/",
    "ClientName": "dataConsumer"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Получение параметра из конфигурации

```
var apiUrl = builder.Configuration["ApiData:Url"];
```

Или

```
var apiUrl = builder.Configuration
                    .GetSection("ApiData:Url").Value
```

Получение параметра из конфигурации

Для строки подключения к базе данных имеется свой специальный метод **GetConnectionString**.

```
var connectionString = builder
        .Configuration
        .GetConnectionString("DefaultConnection");
```

Получение параметра из конфигурации

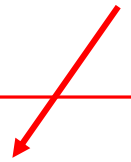
```
public HomeController(IConfiguration configuration,
                      ILogger<HomeController> logger)
{
    _logger = logger;
    var connectionString = configuration
            .GetConnectionString("DefaultConnection");
};
```

Получение параметра из конфигурации

```
{
. . .

"ApiData": {
    "Url": "https://somedata.by/api/",
    "ClientName": "dataConsumer"
  },

. . .
}
```



```
public class ConfigData
{
    public string Uri { get; set; }
    public string ClientName { get; set; }
}
```

Получение параметра из конфигурации

```
var config = new ConfigData();
builder.Configuration
       .GetSection("ApiData")
       .Bind(config);

или

var config = builder.Configuration
                    .GetSection("ApiData")
                    .Get<ConfigData>();
```

IOptions

```
builder.Services
        .Configure<ConfigData>(
                Builder
                    .Configuration
                    .GetSection("ApiData")
            );
```

IOptions

```
Public HomeController(IOptions<ConfigData> config,
                      ILogger<HomeController> logger)
{
    _logger = logger;
    ConfigData cData = config.Value;
}
```