



Казанский федеральный  
УНИВЕРСИТЕТ

ВЫСШАЯ ШКОЛА  
информационных технологий  
и информационных систем

# Введение в ASP.NET Core

# ASP.NET Core

Кросс-платформенный, шустрый,  
open-source фреймворк для создания  
web-приложений MVC, сервисов,  
мобильных и SPA бэкендов и т.д.

# Кросс-платформенный

- Работает поверх .NET Core на Windows, Mac OS, Linux
- Нет привязки к IIS, можно развертывать в качестве отдельного процесса

# Фреймворк

Программная платформа, определяющая структуру программной системы, облегчающая разработку и объединение её компонентов

- Фреймворк – каркас, к которому дописывается функциональность
- Библиотека – готовый модуль, используемый в разрабатываемом коде

# Проект ASP.NET Core

- Приложение ASP.NET Core – консольное приложение, создающее сервер (WebHost) в Main

# Вспомним Builder Pattern

- Порождающий паттерн проектирования
- Создание и сборка частей составного объекта с помощью Builder'a
- Позволяет изменять внутреннее представление объекта
- Позволяет контролировать процесс конструирования

# WebHost Builder

```
using Microsoft.AspNetCore;
using Microsoft.AspNetCore.Hosting;

namespace webApp
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost
            (string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
                .Build();
    }
}
```

Контроль над созданием сервера из составных частей

## Default Builder:

- Использует Kestrel
- Устанавливает ContentRootPath в текущую директорию
- Загружает конфигурацию из appsettings.json и assetsettings.[EnvironmentName].json
- Задействует логирование в консоль и отладочный вывод
- Задействует возможность интеграции с IIS,
- Добавляет страницу исключений разработчика для среды Development

# Content & Web root path

- **Content root** – путь к содержимому приложения (в первую очередь, к представлениям)
- **Web root** – путь к публичным, статическим ресурсам (CSS, javascript, картинкам)



# Окружения

- ASP.NET Core считывает переменную среды **ASPNETCORE\_ENVIRONMENT** при старте приложения, сохраняя значение в `IHostingEnvironment.EnvironmentName`
- Поддерживаются **Development, Staging, Production**
- Можно управлять работой приложения в зависимости от окружения

# Примеры работы с окружением

## Включение или отключение:

- Страницы исключений разработчика
- Элементов (разметки) страниц

## Изменение:

- URL приложения
- Строки подключения к СУБД

# Startup

Класс, определяющий  
функционирование приложения

может называться как угодно,  
в приложении могут быть несколько  
классов с одним именем в разных  
namespace'ах

# Содержимое Startup

- Опциональный метод **ConfigureServices**
  - вызывается до Configure
- **Обязательный метод Configure**
- Поля и конструктор для внедрения зависимостей из WebHost'a

# Convenience методы

- Convenience методы `ConfigureServices` и `Configure` из класса `WebHostBuilder` позволяют управлять сервисами и `middleware pipeline` во время создания сервера (хоста) без определения `Startup` класса

# Сервисы

- В методе `ConfigureServices` описываются сервисы, используемые приложением
- Все сервисы добавляются во встроенный IoC контейнер `IServiceCollection` и могут использоваться при работе приложения (в `Configure`)

# Инъекция зависимостей

- Сервисы интенсивно используются в разных частях приложения
- Нужно ослаблять зависимости между объектами
- Не создавая зависимости напрямую и не используя статические ссылки
- Можно принимать интерфейсные ссылки в конструкторе (как стратегии)

# IoC контейнеры

- Для работы с зависимостями большого количества классов полезно иметь специальный класс, создающий экземпляры классов с указанием зависимостей
- Inversion of Control (IoC) контейнеры – фабрики, ответственные за создание зависимых объектов вместе с их зависимостями и управление временем жизни зависимостей
- ASP.NET Core содержит встроенный IoC контейнер



# Инъекция через конструктор

- Для получения зависимостей с помощью конструктора у большого количества классов полезно иметь специальный класс, создающий экземпляры классов с указанием зависимостей
- IoC контейнеры – фабрики, ответственные за создание зависимых объектов вместе с их зависимостями и управление временем жизни зависимостей
- ASP.NET Core содержит простой встроенный Inversion of Control (IoC) контейнер `IServiceProvider`

# Инъекция в *Startup*

Класс **Startup** сам может принимать зависимости от *WebHost*'а через конструктор

- **IHostingEnvironment** – для настройки сервисов в зависимости от окружения
- **Iconfiguration** – для конфигурирования приложения в *Startup*

# Создание сервисов

Определим интерфейс сервиса

```
public interface IMessageSender
{
    string Send();
}
```

И реализующий этот интерфейс класс

```
public class EmailMessageSender : IMessageSender
{
    public string Send() => "Sent by Email";
}
```

# Регистрация и инъекция сервисов

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        Сопоставим интерфейс с реализацией
        services.AddTransient<IMessageSender, EmailMessageSender>();
    }

    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env, IMessageSender messageSender)
    {
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync(messageSender.Send());
        });
    }
}
```

# Создание сервисов

При создании сервисов и добавлении их в IoC контейнер *можно не выделять явно интерфейс и реализацию сервиса,*  
**но лучше так не делать**

Для удобства можно реализовать методы расширения IServiceCollection для более удобного добавления сервисов в контейнер

# Время жизни сервиса

- **Transient** – объект сервиса создаётся при каждом обращении
- **Scoped** – объект создаётся для каждого запроса
- **Singleton** – создаётся при первом обращении

# Замена контейнера

- Встроенный контейнер удовлетворяет базовым потребностям, но может возникнуть потребность его замены
- Для этого нужно изменить тип возвращаемого значения метода `ConfigureServices` с `void` на `IServiceProvider` и воспользоваться `ContainerBuilder`'ом

# Замена на autofac

Подключаем пакеты

- Autofac
- Autofac.Extensions.DependencyInjection

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Add other framework services

    // Add Autofac
    var containerBuilder = new ContainerBuilder();
    containerBuilder.RegisterModule<DefaultModule>();
    containerBuilder.Populate(services);
    var container = containerBuilder.Build();
    return new AutofacServiceProvider(container);
}
```



# Альтернативный доступ к сервисам

- Кроме инъекции зависимостей можно обращаться к сервисам

- Через `ServiceScope`

```
using (var serviceScope =
app.ApplicationServices.GetService<IServiceScopeFactory>().CreateScope())
{
    var context = serviceScope.ServiceProvider.GetRequiredService<AppDbContext>();
    context.Database.Migrate();
}
```

- Через `HttpContext`

```
var db = context.RequestServices.GetService<AppDbContext>();
```

# Middleware

- Метод `Configure` определяет как обрабатываются HTTP запросы
- Используется `ApplicationBuilder`, komponующий компоненты `middleware` в цепочку (pipeline)
- `ApplicationBuilder` создается хостом и передается напрямую в `Configure`

# Методы Use и Run

- Лежат в основе компонент middleware
- Run – завершает обработку
  - Принимает RequestDelegate (`Func<HttpRequest, Task>`)
- Use – выполняет действия и может передать обработку следующей компоненте
  - Принимает RequestDelegate и ссылку на следующий компонент в цепочке (`Func<Task> next`), выполняет действия и вызывает `next.Invoke()`

# Методы Use и Run

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        await context.Response.WriteAsync("<p>Hello world!</p>");
        await next.Invoke();
    });

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("<p>Good bye, World...</p>");
    });
}
```

# Методы Map и MapWhen

Сопоставляют путь запроса с делегатом-обработчиком

```
public void Configure(IApplicationBuilder app)
{
    app.MapWhen(context => context.Request.Query.ContainsKey("id") &&
                context.Request.Query["id"] == "5", HandleId);
    app.Map("/about", About);

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Page Not Found");
    });
}
```

# Создание компонент `middleware`

- Можем описывать компоненты `middleware` в отдельных классах и добавлять их в конвейер методом `UseMiddleware`
  - (если имя заканчивается на `Middleware`, то просто `UseИмя`)
- Класс должен содержать конструктор, принимающий `RequestDelegate (next)` и содержать метод  
`public async Task InvokeAsync(HttpContext context)`

# Встроенные компоненты middleware

- Authentication
- CORS
- URL Rewriting
- Routing
- Session
- Response Caching
- Response Compression

# Статические файлы

- ASP.NET Core содержит middleware (UseStaticFiles)
  - Для публичных css, js, картинок
- Файлы хранятся в директории `<content_root>/wwwroot`
- Путь к WebRoot можно менять при создании WebHost'a методом UseWebRoot
- Можно управлять заголовками HTTP ответов
- Можно разрешить просмотр файлов в директории



# Конвейер обработки запроса

- Обработка запроса выполняется компонентами в порядке добавления в конвейер
- Рекомендуется придерживаться следующего порядка обработки:
  - Ошибки
  - Статические файлы
  - Аутентификация
  - MVC (формирование данных и представлений)

# Время жизни `middleware`

- Метод `Configure` вызывается один раз при создании экземпляра `Startup`
- Все компоненты `middleware` создаются один раз и живут пока выполняется приложение
- Можно проверить, заведя локальный счетчик в `Configure`, увеличивая и возвращая его значение. Значение не будет сбрасываться на начальное

# Публикация в Azure

- Облачная платформа Microsoft для приложений и данных
  - [azure.microsoft.com](https://azure.microsoft.com)
  - 1 месяц бесплатно
- Опубликовать приложение в Azure можно достаточно просто прямо из IDE
- См. инструкцию <https://docs.microsoft.com/en-us/aspnet/core/tutorials/publish-to-azure-webapp-using-vs>

# Heroku

- Известная PaaS платформа для приложений и данных
- Позволяет развёртывать приложения ASP.NET Core при помощи git (deploy with Heroku git) и соответствующего .NET Core buildpack'a
- Позволяет бесплатно хостить приложения и базы данных (с ограничениями)

# Развёртывание в Heroku

- Регистрируемся в Heroku
- Устанавливаем Heroku CLI
- В Dashboard создаём новый проект, указываем способ публикации Heroku git
- Создаём приложение, добавляем в систему контроля версий (git init)
- Устанавливаем Heroku build pack  
heroku buildpacks:set <https://github.com/jincod/dotnetcore-buildpack> -a [AppName]
- Связываем репозиторий с Heroku  
heroku git:remote -a [AppName]
- Публикуем проект  
git push heroku master

# Heroku PostgreSQL

- Создаём в Heroku Dashboard новый проект и добавляем в Resources базу данных Heroku PostgreSQL
- В настройках базы данных узнаём Credentials
- Указываем их в connection string

# Ссылки

- <https://github.com/aspnet/>
- <https://metanit.com/sharp/aspnet5/>
- <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>
- <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/index?tabs=aspnetcore2x>
- <https://heroku.com>
- <https://github.com/jincod/dotnetcore-buildpack>
- <https://docs.microsoft.com/en-us/aspnet/core/tutorials/publish-to-azure-webapp-using-vs>

# Темы и предложения для разбора

- MVC, Razor и другие полезные инструменты
- Аутентификация и авторизация
- WebAPI
- Тестирование
- Загрузка файлов. Облачное хранение данных
- Функциональность реального времени
- Развёртывание на Linux с Nginx
- Микросервисы на ASP.NET Core
- Контейнерная виртуализация
- Одностраничные приложения (SPA)
- Web-приложения ASP.NET Core на F#





Казанский федеральный  
УНИВЕРСИТЕТ

ВЫСШАЯ ШКОЛА  
информационных технологий  
и информационных систем

# Введение в ASP.NET Core