

Лекция №11. ASP.NET

ASP.NET Core — это современная, кросс-платформенный фреймворк для создания высокопроизводительных веб-приложений и сервисов. Разработанная компанией Microsoft.

ASP.NET Core поддерживает создание различных типов приложений:

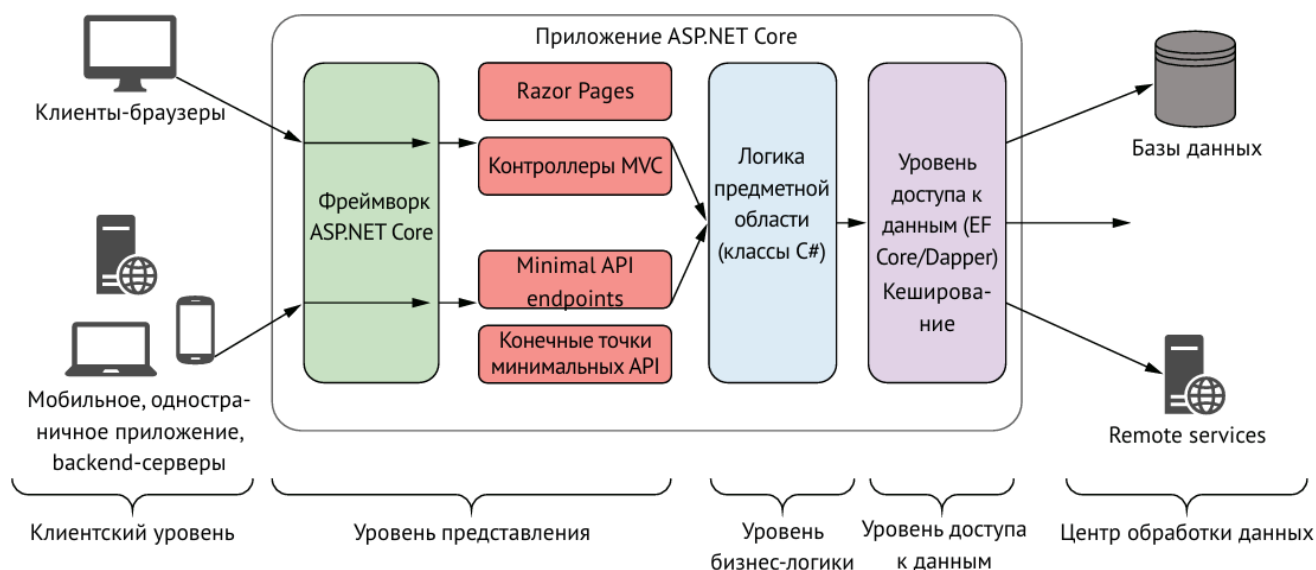
- **Веб-приложения** (MVC, Razor Pages).
- **API** (RESTful, GraphQL).
- **Микросервисы**.
- **Реалтайм-приложения** (SignalR).

Приложения ASP.NET Core могут обслуживать клиентов на основе браузера или могут предоставлять API-интерфейсы для мобильных и других клиентов

Код фреймворка ASP.NET обрабатывает низкоуровневые запросы и вызывает «обработчики» контроллеров Razor Pages и веб-API

Вы пишете эти обработчики, используя примитивы, предоставляемые фреймворком. Обычно они вызывают методы в логике предметной области

Предметная область может использовать внешние сервисы и базы данных для выполнения своих функций и сохранения данных



Современные веб-архитектуры

Эволюция веб-разработки

Раньше доминировал **server-side рендеринг** (MVC, Razor Pages), где сервер генерировал HTML и отправлял его в браузер.

1. Пользователь запрашивает веб-страницу по URL-адресу



5. Браузер отображает HTML-код на странице



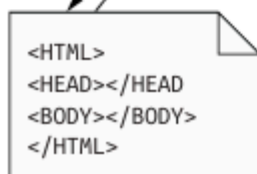
2. Браузер отправляет HTTP-запрос на сервер



4. Сервер отправляет HTML-код в ответ браузеру



3. Сервер интерпретирует запрос и генерирует соответствующий HTML-код



Но с ростом сложности интерфейсов и мобильных приложений появилась необходимость в:

- **Отделении фронтенда от бэкенда.**
- **Едином API** для всех клиентов (веб, мобильные, IoT).

Это привело к популярности **SPA (Single Page Application)** — приложений, где страница загружается один раз, а дальнейшее взаимодействие происходит через API без перезагрузки (например, Gmail, Facebook).

SPA и API

- **SPA** — это фронтенд-приложение (на React, Vue, Angular), которое работает в браузере и общается с сервером только через API.

- **Роль WebAPI:**
 - Предоставляет данные в формате JSON/XML.
 - Обработывает бизнес-логику, авторизацию, работу с БД.
 - Позволяет использовать один бэкенд для разных клиентов (веб, мобильные приложения).

Пример:

Когда вы открываете Instagram:

1. Браузер загружает SPA (статический JS/CSS).
2. SPA запрашивает через API список постов (GET /api/posts).
3. Сервер (ASP.NET Core) возвращает JSON с данными.
4. SPA рендерит посты на стороне клиента.

Сравнение WebAPI и MVC/Razor Pages

- **MVC/Razor Pages** подходят для:
 - Server-side рендеринга (полезно для SEO).
 - Простых проектов с минимумом динамики (блоги, лендинги).
- **WebAPI** выбирают, когда:
 - Фронтенд написан на React/Angular/Vue.
 - Нужен единый бэкенд для сайта, мобильного приложения и партнерских интеграций.
 - Требуется высокая производительность и масштабируемость.

Ключевое отличие:

- MVC возвращает HTML, WebAPI — данные (JSON).
- WebAPI не привязан к интерфейсу, что упрощает поддержку и развитие.

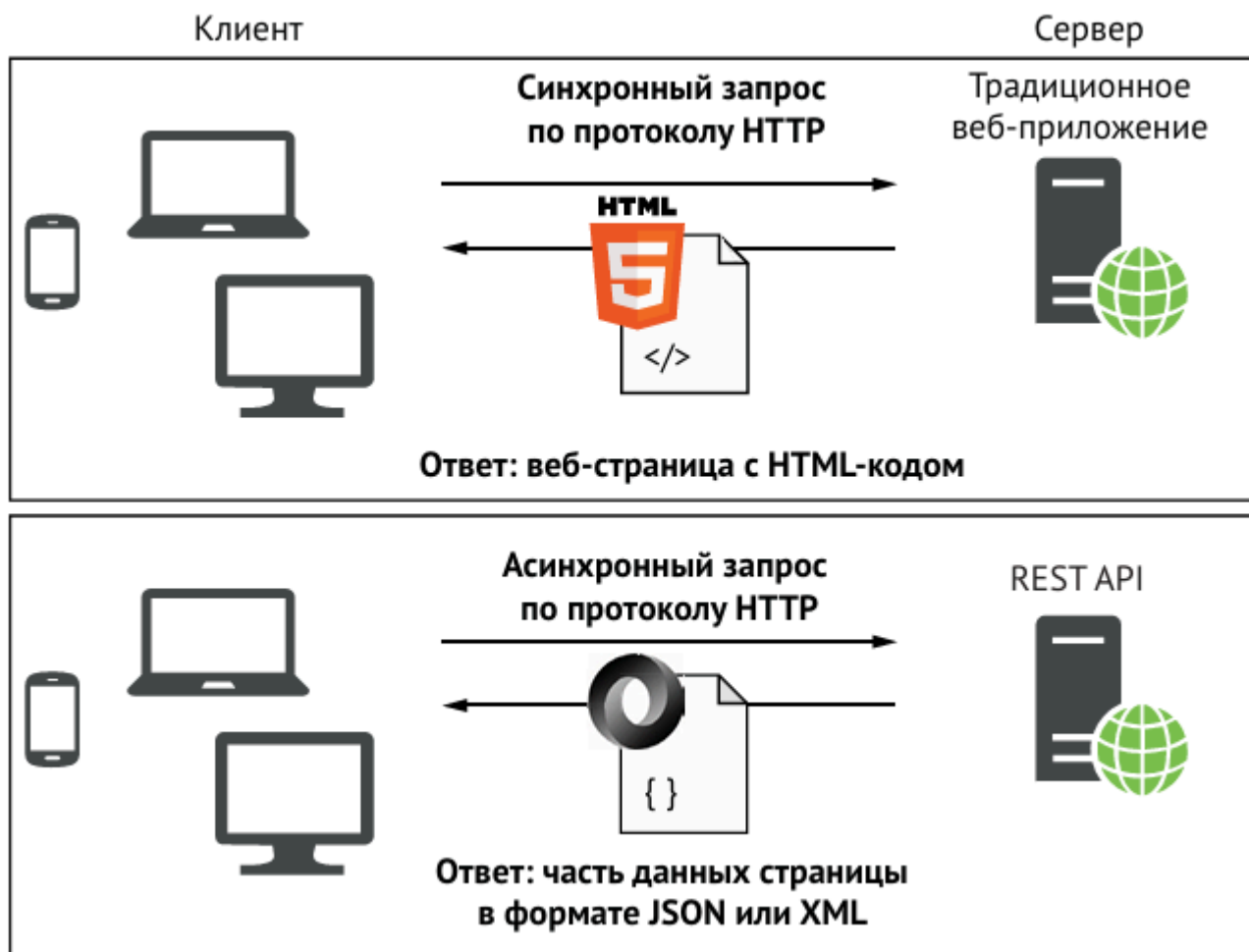
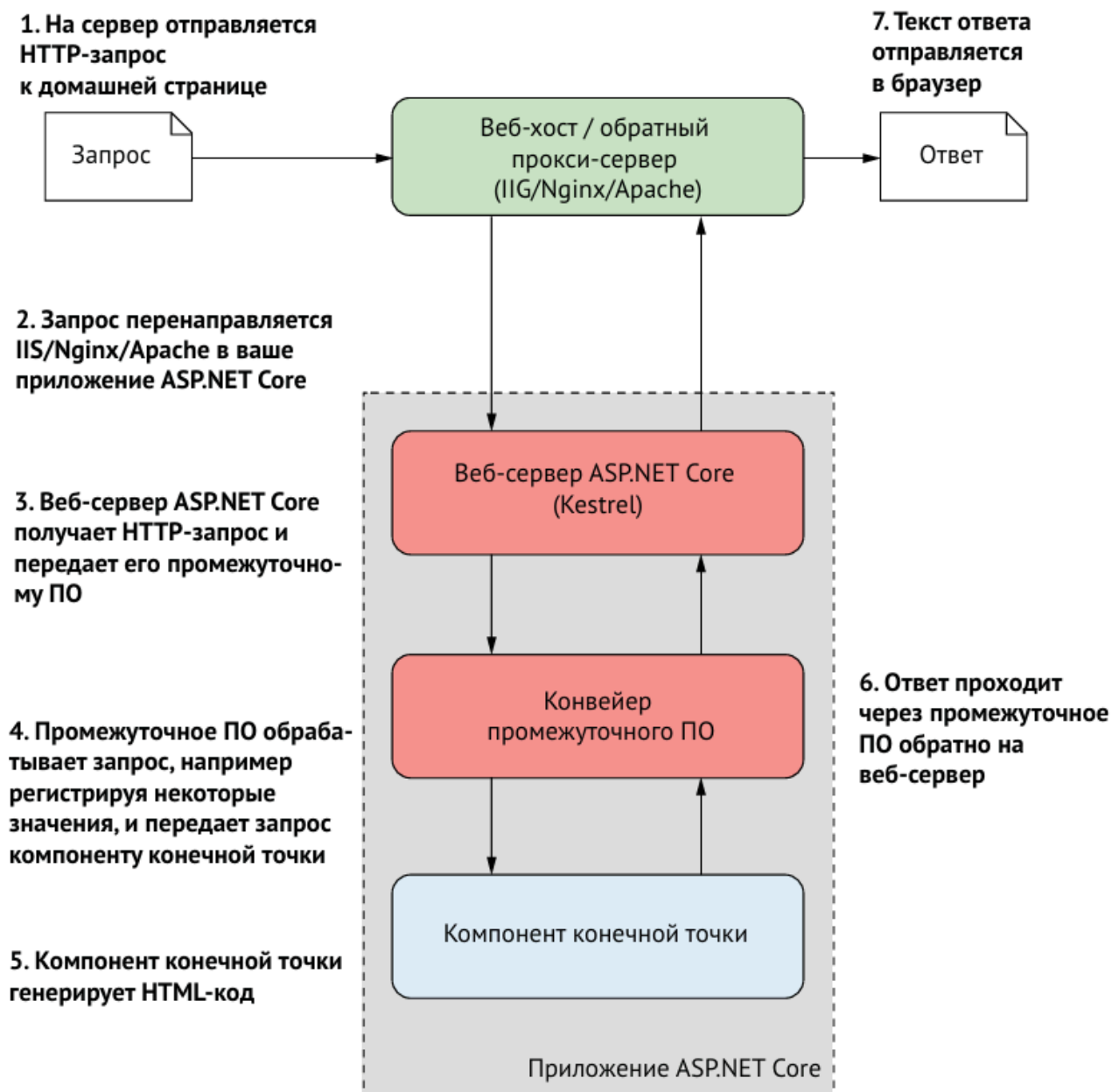


Схема работы приложения ASP.NET

ASP.NET Core приложение работает по принципу **конвейера запросов**, где каждый компонент выполняет свою часть работы. Вот как это выглядит:



Kestrel — это встроенный кросс-платформенный веб-сервер, обрабатывающий HTTP-запросы. Он выполняет следующие задачи:

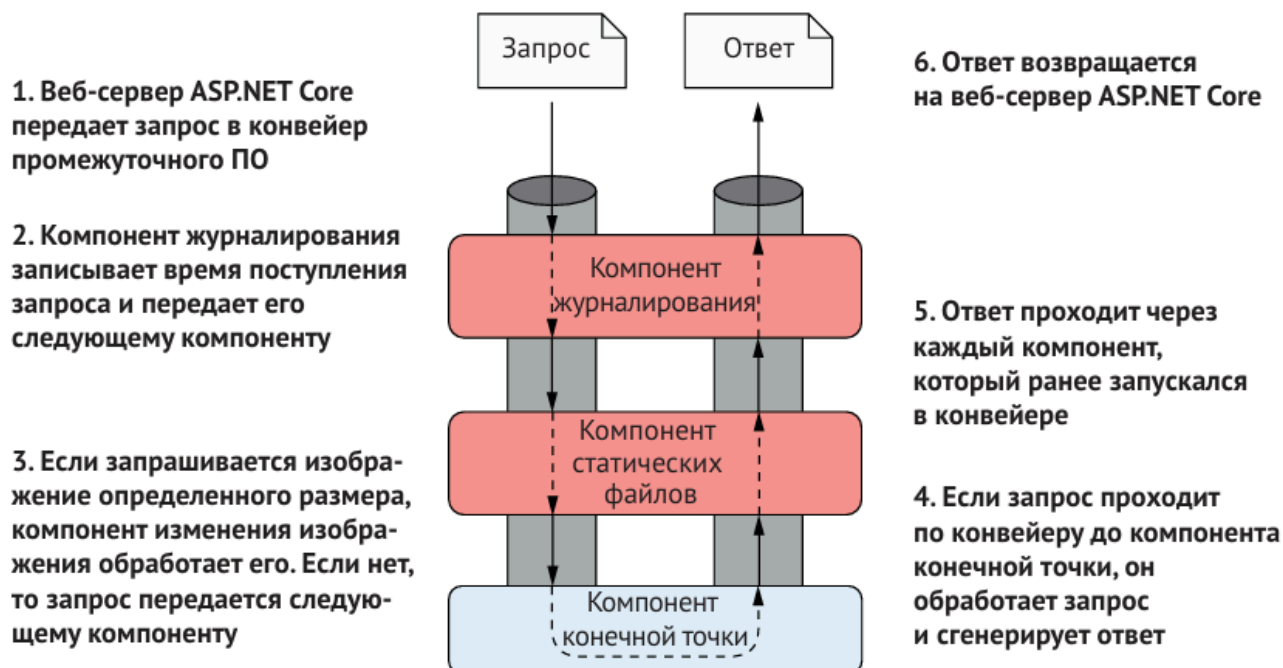
- Принимает входящие запросы.
- Управляет жизненным циклом HTTP (парсинг заголовков, формирование ответов).

Nginx — это обратный прокси сервер. Он используется из-за того, что Kestrel не предназначен для прямого доступа из интернета и работает только локально. Nginx выполняет много важных задач:

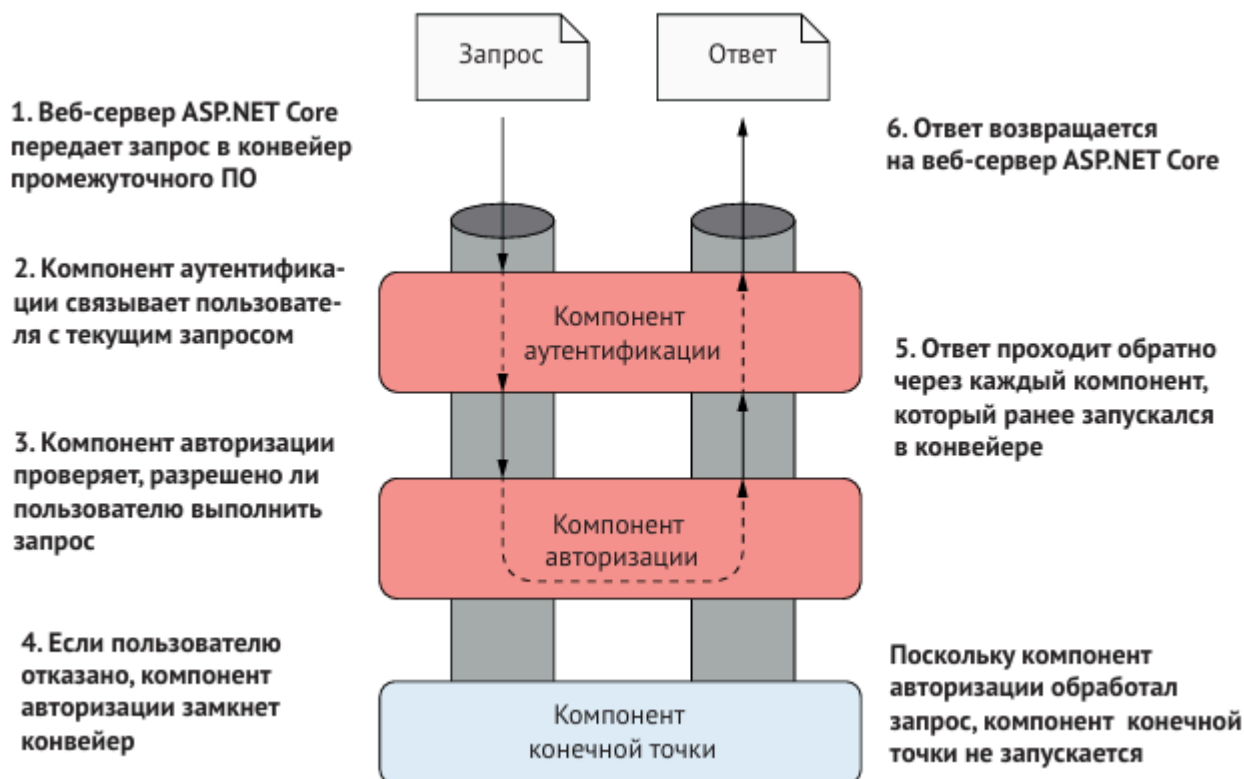
- **SSL-терминация:** Обрабатывает HTTPS-шифрование.
- **Балансировка нагрузки:** Распределяет запросы между несколькими экземплярами Kestrel.
- **Обслуживание статики:** Отдает CSS/JS/изображения без участия ASP.NET.
- **Защита:** Фильтрация DDoS-атак, ограничение запросов.

Middleware — это компоненты, которые обрабатывают запрос и ответ **последовательно**, как конвейер. Например:

- `UseHttpsRedirection()` – перенаправляет HTTP → HTTPS.
- `UseStaticFiles()` – отдает статические файлы (CSS, JS).
- `UseAuthentication()` – проверяет аутентификацию.



Обработка ошибок с помощью конвейера промежуточного ПО:

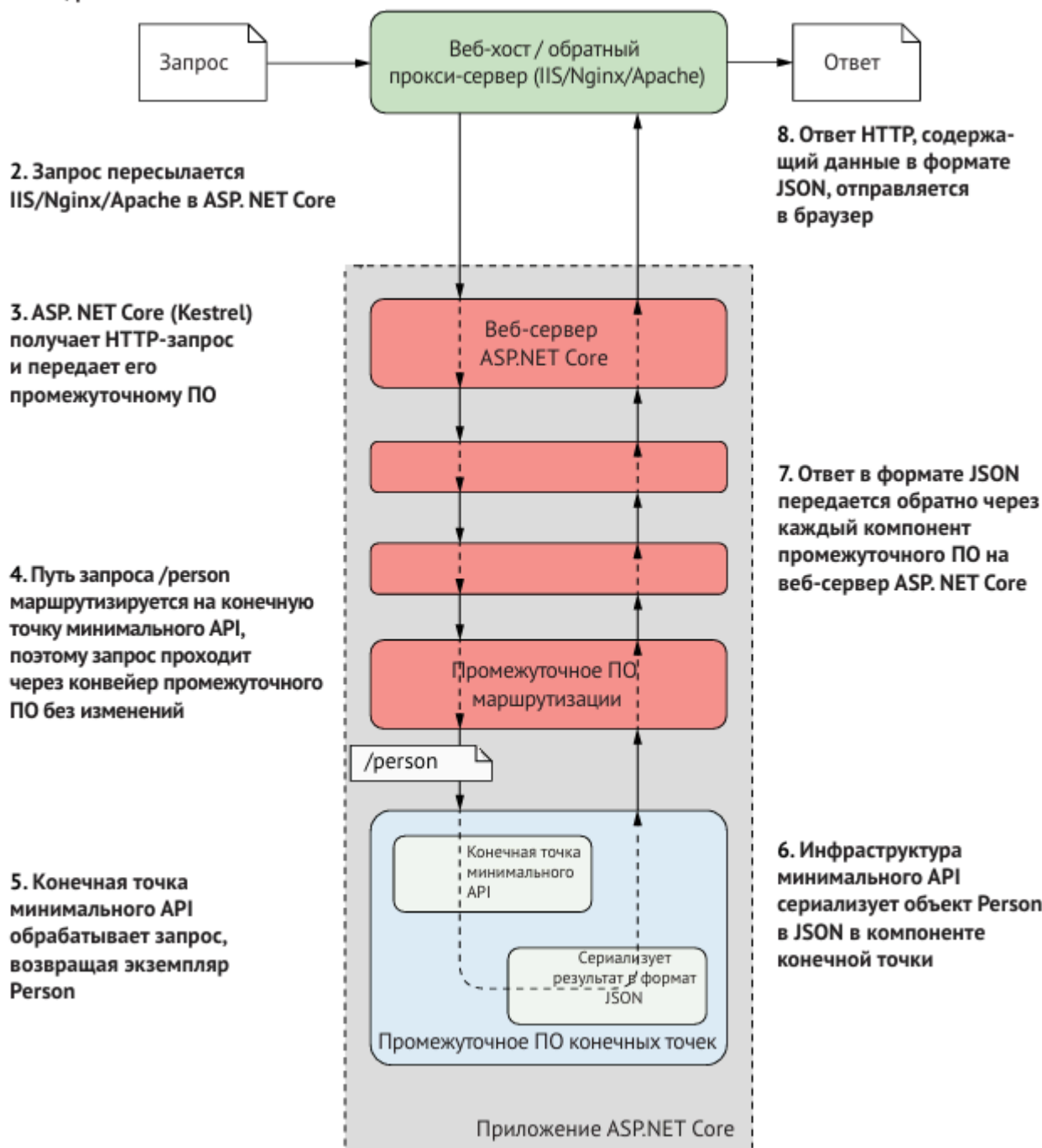


Из-за того, что Middleware работает последовательно, порядок, в котором были созданы компоненты критически важен.

Endpoints — это логические адреса приложения, которые обрабатывают запросы (например, `/api/users`). Они играют следующую роль:

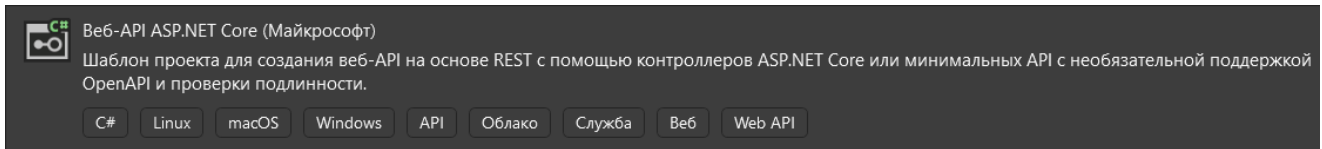
- Выполняют бизнес-логику.
- Возвращают данные (JSON, файлы, HTML).

1. Выполняется HTTP-запрос к URL/person

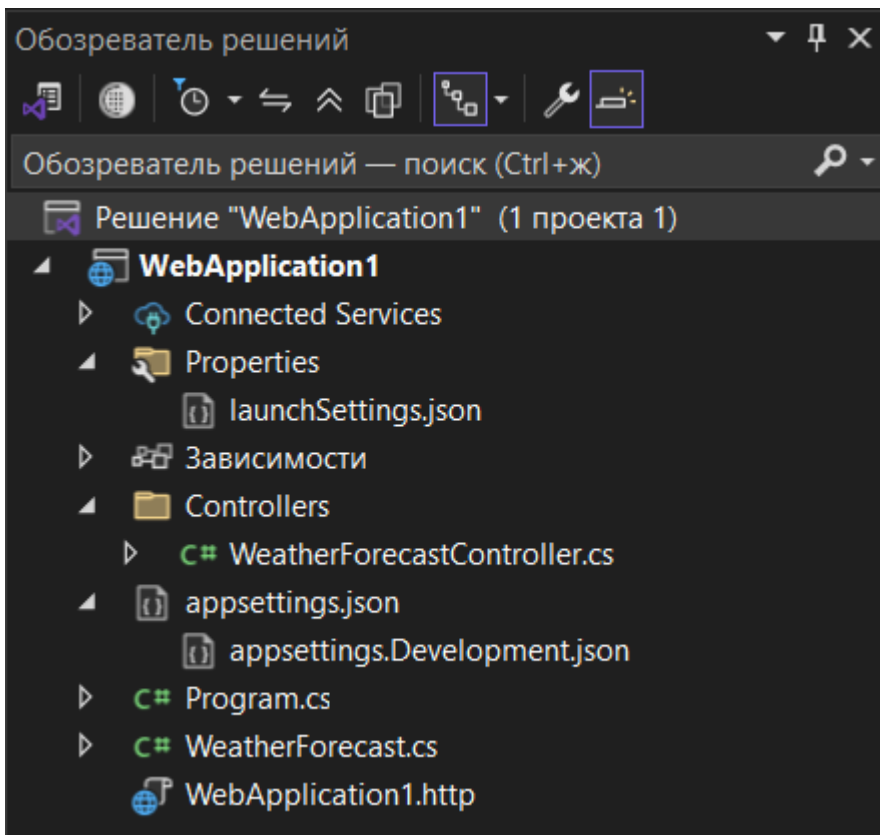


Создание и структура WebAPI-проекта

Давайте создадим проект WebAPI и рассмотрим его структуру. Для этого выберем нужный шаблон в VisualStudio и создадим приложение.



В результате получим такой проект, который сможет давать прогноз погоды.



Изучим структуру проекта:

- **appsettings.json**: Конфигурация (строки подключения, настройки).
- **Controllers/**: Папка для API-контроллеров.
- **Program.cs**:
 - Точка входа приложения.
 - Настройка сервера (Kestrel), Middleware, маршрутизация, Swagger и т.д.

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);
        builder.Services.AddControllers();
        builder.Services.AddEndpointsApiExplorer();
        builder.Services.AddSwaggerGen();
        var app = builder.Build();
    }
}
```



```

        if (app.Environment.IsDevelopment())
        {
            app.UseSwagger();
            app.UseSwaggerUI();
        }
        app.UseHttpsRedirection();
        app.UseAuthorization();
        app.MapControllers();
        app.Run();
    }
}

```

Основные компоненты WebAPI

1. Контроллеры (Controllers):

- Классы, обрабатывающие HTTP-запросы.
- Являются конечными точками (Endpoints).
- Наследуются от `ControllerBase`.
- По умолчанию данные передаются в JSON.

Пример:

```

[ApiController]
[Route("api/[controller]")]
public class WeatherForecastController : ControllerBase
{
    // ...

    [HttpGet(Name = "GetWeatherForecast")]
    public IEnumerable<WeatherForecast> Get()
    {
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .ToArray();
    }
}

```

2. Атрибуты маршрутизации:

- `[Route("api/[controller]")]`:
 - `[controller]` заменяется на имя контроллера.

- [HttpGet] , [HttpPost] : Определяют метод HTTP.
- [FromBody] , [FromQuery] : Указывают источник данных (тело запроса, query-параметры).

3. Жизненный цикл запроса:

- Запрос → Kestrel → Middleware (например, CORS, аутентификация) → Контроллер → Ответ.
- **Middleware в Program.cs:**


```
app.UseHttpsRedirection(); // Перенаправление на HTTPS
app.UseAuthorization();
app.MapControllers();
```

4. Swagger UI:

Подключается в Program.cs:

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
// ...
// Только в режиме разработки
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Позволяет тестировать эндпоинты прямо в браузере.

 Swagger
OPEN API SPECIFICATION

Select a definitionWebApplication1 v1

WebApplication11.0OAS 3.0
<http://localhost:5120/swagger/v1/swagger.json>

WeatherForecast

Ok! /api/WeatherForecast

Parameters

No parameters

Cancel

ExecuteClear

Response

Curl

curl -X 'GET' \n'http://localhost:5120/api/WeatherForecast' \n-H 'accept: text/plain'

Request URL

http://localhost:5120/api/WeatherForecast

Server response

CodeDetail

200

Response body

```
[
  {
    "date": "2023-04-04",
    "temperatureC": 10,
    "temperatureF": 50,
    "summary": "Sunny"
  },
  {
    "date": "2023-04-05",
    "temperatureC": 10,
    "temperatureF": 50,
    "summary": "Sunny"
  },
  {
    "date": "2023-04-06",
    "temperatureC": 10,
    "temperatureF": 50,
    "summary": "Sunny"
  },
  {
    "date": "2023-04-07",
    "temperatureC": 10,
    "temperatureF": 50,
    "summary": "Sunny"
  },
  {
    "date": "2023-04-08",
    "temperatureC": 10,
    "temperatureF": 50,
    "summary": "Sunny"
  },
  {
    "date": "2023-04-09",
    "temperatureC": 10,
    "temperatureF": 50,
    "summary": "Sunny"
  }
]
```

Response headers

```
content-type: application/json; charset=utf-8
date: Mon, 04 Apr 2023 08:18:00 GMT
server: Kestrel
transfer-encoding: chunked
```

Response

CodeDescriptionLink

200OKNo link

Media type

text/plain

Content Accept header

Example Value | Schema

```
[
  {
    "date": "2023-04-04",
    "temperatureC": 10,
    "temperatureF": 50,
    "summary": "string"
  }
]
```

Schemas

WeatherForecast {

date> [...]

temperatureC> [...]

temperatureF> [...]

summary> [...]

}

Так же для тестирования API можно использовать программу Postman, которая обладает большим функционалом.

http://localhost:5133/api/WeatherForecast

GET http://localhost:5133/api/WeatherForecast

Send

Params Authorization Headers (6) Body Scripts Tests Settings Cookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

Body Cookies Headers (4) Test Results

200 OK • 9 ms • 528 B

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "date": "2025-04-15",
4     "temperatureC": -14,
5     "temperatureF": 7,
6     "summary": "Bracing"
7   },
8   {
9     "date": "2025-04-16",
10    "temperatureC": 50,
11    "temperatureF": 121,
12    "summary": "Cool"
13  }
```

Работа с данными и Entity Framework Core

Теперь научимся подключать базу данных, реализовывать CRUD-операции, валидировать данные и внедрять базовую аутентификацию.

Для начала переделаем исходный шаблонный проект, теперь вместо погоды он будет обрабатывать данные о товарах.

Подключение Entity Framework Core

1. Установка пакетов:

```
dotnet add package Microsoft.EntityFrameworkCore
# Будем работать с SQLite:
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

2. Создание контекста базы данных:

```
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
    base(options) {
        // Создаем БД при первом обращении к контексту
        Database.EnsureCreated();
    }

    public DbSet<Product> Products { get; set; }
}
```

3. Регистрация контекста в Program.cs:

```
builder.Services.AddDbContext<AppDbContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("Default"
)));
```

4. Настройка строки подключения в appsettings.json:

```
"ConnectionStrings": {
  "Default": "Data Source=app.db;"
}
```

Реализация CRUD-операций

Создадим полноценное API для управления продуктами.

1. Обновление контроллера:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private readonly AppDbContext _context;
    public ProductsController(AppDbContext context) => _context =
context;

    [HttpGet]
    public async Task<IActionResult> GetAll() => Ok(await
_context.Products.ToListAsync());

    [HttpGet("{id}")]
    public async Task<IActionResult> GetById(int id)
    {
        var product = await _context.Products.FindAsync(id);
        return product == null ? NotFound() : Ok(product);
    }

    [HttpPost]
    public async Task<IActionResult> Create(Product product)
    {
        await _context.Products.AddAsync(product);
        await _context.SaveChangesAsync();
        return CreatedAtAction(nameof(GetById), new { id = product.Id },
product);
    }
}
```

```
}  
}
```

Использование DTO и AutoMapper

DTO (Data Transfer Object) — это шаблон проектирования, который используется для передачи данных между слоями приложения. Основная цель — изолировать модель домена от слоя представления или API, чтобы изменения в одной части не влияли на другую. Например, в веб-API мы не хотим напрямую использовать сущности базы данных в ответах, чтобы не раскрывать внутреннюю структуру или лишние данные.

1. Создание DTO (Data Transfer Object):

```
public class ProductDto  
{  
    public string Name { get; set; }  
    public decimal Price { get; set; }  
}
```

AutoMapper автоматизирует маппинг между сущностями.

Маппинг — это процесс преобразования данных из одного формата в другой. Он связывает объекты или структуры данных между разными слоями приложения, чтобы они могли "понимать" друг друга.

2. Настройка AutoMapper:

- Установка пакета:

```
dotnet add package  
AutoMapper.Extensions.Microsoft.DependencyInjection
```

- Регистрация в Program.cs:

```
builder.Services.AddAutoMapper(typeof(Program));
```

- Профиль маппинга:

```
public class ProductProfile : Profile  
{  
    public ProductProfile() => CreateMap<ProductDto, Product>
```

```
() .ReverseMap();  
}
```

3. Обновление метода Create:

```
[HttpPost]  
public async Task<IActionResult> Create(ProductDto productDto)  
{  
    var product = _mapper.Map<Product>(productDto);  
    await _context.Products.AddAsync(product);  
    await _context.SaveChangesAsync();  
    return CreatedAtAction(nameof(GetById), new { id = product.Id },  
product);  
}
```

Валидация данных

Для обеспечения корректности входящих запросов необходимо настроить валидацию. Для этого нужно выполнить следующие шаги:

1. Аннотации данных в DTO:

```
public class ProductDto  
{  
    [Required, StringLength(100)]  
    public string Name { get; set; }  
  
    [Range(0.01, 10000)]  
    public decimal Price { get; set; }  
}
```

2. Проверка валидации в контроллере:

```
if (!ModelState.IsValid)  
    return BadRequest(ModelState);
```

3. Глобальная обработка ошибок:

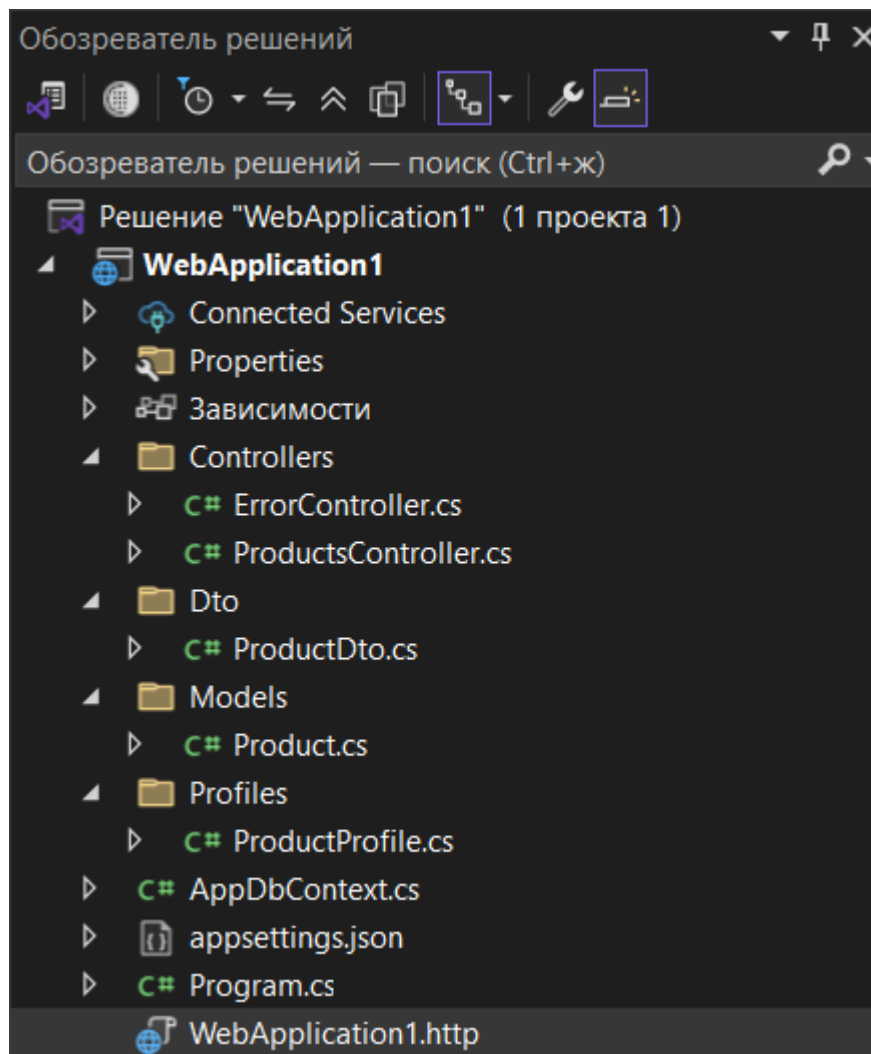
Добавить в Program.cs:


```
app.UseExceptionHandler("/error");
```

Создать контроллер для обработки ошибок:

```
[ApiController]
public class ErrorController : ControllerBase
{
    [HttpGet("/error")]
    public IActionResult HandleError() => Problem();
}
```

В результате получим проект с такой структурой:



 Swagger
powered by SMARTBEAR

Select a definitionWebApplication1 v1

WebApplication1 1.0 OAS 3.0
http://localhost:5133/swagger/v1/swagger.json

Error ^

GET /error v

Products ^

GET /api/Products v

POST /api/Products v

GET /api/Products/{id} v

Schemas ^

ProductDto >

Базовая аутентификация с JWT

Токены на предъявителя (токены носителя, **bearer tokens**) – это строки, содержащие сведения об аутентификации пользователя или приложения. Они могут быть или не быть зашифрованы, но обычно подписываются, чтобы избежать подделки. **JWT (JSON Web Token)** – наиболее распространенный формат токена для безопасной передачи данных между клиентом и сервером. Это строка, состоящая из трех частей:


```
Header.Payload.Signature
```

- **Header** – алгоритм подписи и тип токена (например, `HS256`).
- **Payload** – полезные данные (claims), например, идентификатор пользователя, роли, срок действия.
- **Signature** – подпись, гарантирующая целостность токена.


JSON Web Tokens - jwt.io

https://jwt.io

Not syncing

 JWT

Debugger Libraries Introduction Ask

Crafted by  auth0 by Okta

Encoded PASTE A TOKEN HERE

Decoded EDIT THE PAYLOAD AND SECRET

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bm1ldWVfYmFtZSI6ImFuZlZhd2xvY2submV0QGdtYWlsLmNvbSI6ImFuZlZhd2xvY2submV0QGdtYWlsLmNvbSI6Imp0aSI6Ijg4OWEzMDFhIiwid2Vic2l0ZSI6Imh0dHBzOi8vYW5kcmV3bG9jay5uZXQlLCJhdWQiOi01siaHR0cDovL2xvY2FsaG9zdDo1MDczIiwiaHR0cHM6Ly9sb2NhbgG9zQ6NzExMjJdLCJzY29wZSI6WyJyZWZkIiwid3JpdGUxXSwibmJmIjojY2NDQXNjE5LCJleHAiOjE2NzQzOTA0MTksImh0dCI6MTY2NjQ0MTYyMSwiaXNzIjoizG90bmV0LXVzZXItandocyJ9.haz66_V8PBzW4vUV4htHnDYddegdIQqsM56hmD3eNno
```

Каждая часть JWT закодирована в base64 и разделена символом "."

Подпись JWT вычисляется на основе заголовка, полезной нагрузки и ключа подписи

HEADER: ALGORITHM & TOKEN TYPE

```
{  "alg": "HS256",  "typ": "JWT"}
```

Заголовок описывает токен

PAYLOAD: DATA

```
{  "unique_name": "andrewlock.net@gmail.com",  "sub": "andrewlock.net@gmail.com",  "jti": "889a301a",  "website": "https://andrewlock.net",  "aud": [    "http://localhost:5073",    "https://localhost:7112"  ],  "scope": ["read", "write"],  "nbf": 1666441619,  "exp": 1674390419,  "iat": 1666441621,  "iss": "dotnet-user-jwts"}
```

Пользовательское утверждение

Стандартные утверждения

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  QOVONJSMDyQXFjHUg6m1I)
```

☒ secret base64 encoded

Signature Verified

SHARE JWT

Настройка JWT в ASP.NET Core

1. Установка пакетов

```
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

2. Конфигурация в Program.cs

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

// Добавление аутентификации через JWT
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidIssuer = builder.Configuration["Jwt:Issuer"], // Кто
```

```

выдал токен
        ValidateAudience = true,
        ValidAudience = builder.Configuration["Jwt:Audience"], // Для
кого предназначен
        ValidateLifetime = true, //
Проверка срока действия
        IssuerSigningKey = new SymmetricSecurityKey( // Ключ
подписи
            Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]!)
        )
    };
});

var app = builder.Build();
app.UseAuthentication(); // Включение middleware аутентификации
app.UseAuthorization();

```

3. appsettings.json

```

{
  "Jwt": {
    "Issuer": "MyAuthServer",
    "Audience": "MyClientApp",
    "Key": "supersecretkey123!", // Минимум 16 символов
    "LifetimeHours": 1
  }
}

```

Генерация JWT-токена

Пример метода Login:

```

[HttpPost("login")]
public IActionResult Login([FromBody] LoginRequest request)
{
    // Проверка учетных данных (заглушка)
    var user = _userService.Authenticate(request.Username,
request.Password);
    if (user == null) return Unauthorized("Неверный логин или пароль");

    // Создание claims (утверждений)
    var claims = new[]
    {
        new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),
        new Claim(ClaimTypes.Name, user.Username),
        new Claim(ClaimTypes.Role, user.Role)
    };
}

```

```
// Генерация токена
var token = new JwtSecurityToken(
    issuer: _config["Jwt:Issuer"],
    audience: _config["Jwt:Audience"],
    claims: claims,
    expires: DateTime.UtcNow.AddHours(_config.GetValue<double>
("Jwt:LifetimeHours")),
    signingCredentials: new SigningCredentials(
        new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]!)),
        SecurityAlgorithms.HmacSha256
    )
);

return Ok(new
{
    token = new JwtSecurityTokenHandler().WriteToken(token),
    expires = token.ValidTo
});
}
```

Ответ сервера:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "expires": "2024-05-20T12:00:00Z"
}
```

Использование токена в запросах

Клиент отправляет токен в заголовке `Authorization`:

```
GET /api/secure-data HTTP/1.1
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Используйте атрибут `[Authorize]` для ограничения доступа по токenu.

Пример контроллера:

```
[ApiController]
[Route("api/[controller]")]
[Authorize] // Все методы требуют аутентификации
public class SecureController : ControllerBase
{
    [HttpGet("user-info")]
    public IActionResult GetUserInfo()
    {
    }
```

```
// Извлечение данных из токена
var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
var userName = User.Identity?.Name;
var role = User.FindFirstValue(ClaimTypes.Role);

return Ok(new { userId, userName, role });
}

[HttpGet("admin-data")]
[Authorize(Roles = "Admin")] // Только для администраторов
public IActionResult GetAdminData() => Ok("Секретные данные для админов");
}
```

Работа с файлами в ASP.NET Core и клиенте на C#

При разработке веб-приложений важно правильно организовать работу со статическими ресурсами и обеспечить безопасность доступа к ним. Рассмотрим, как настроить сервер для обработки файлов (например, изображений), как клиент может взаимодействовать с сервером для загрузки и получения файлов, а также как ограничить доступ к чувствительным ресурсам.

Статические файлы и папка `wwwroot`

Папка `wwwroot` — специальное место в структуре ASP.NET Core проекта, предназначенное для хранения статических ресурсов, таких как CSS-файлы, JavaScript, изображения и документы.

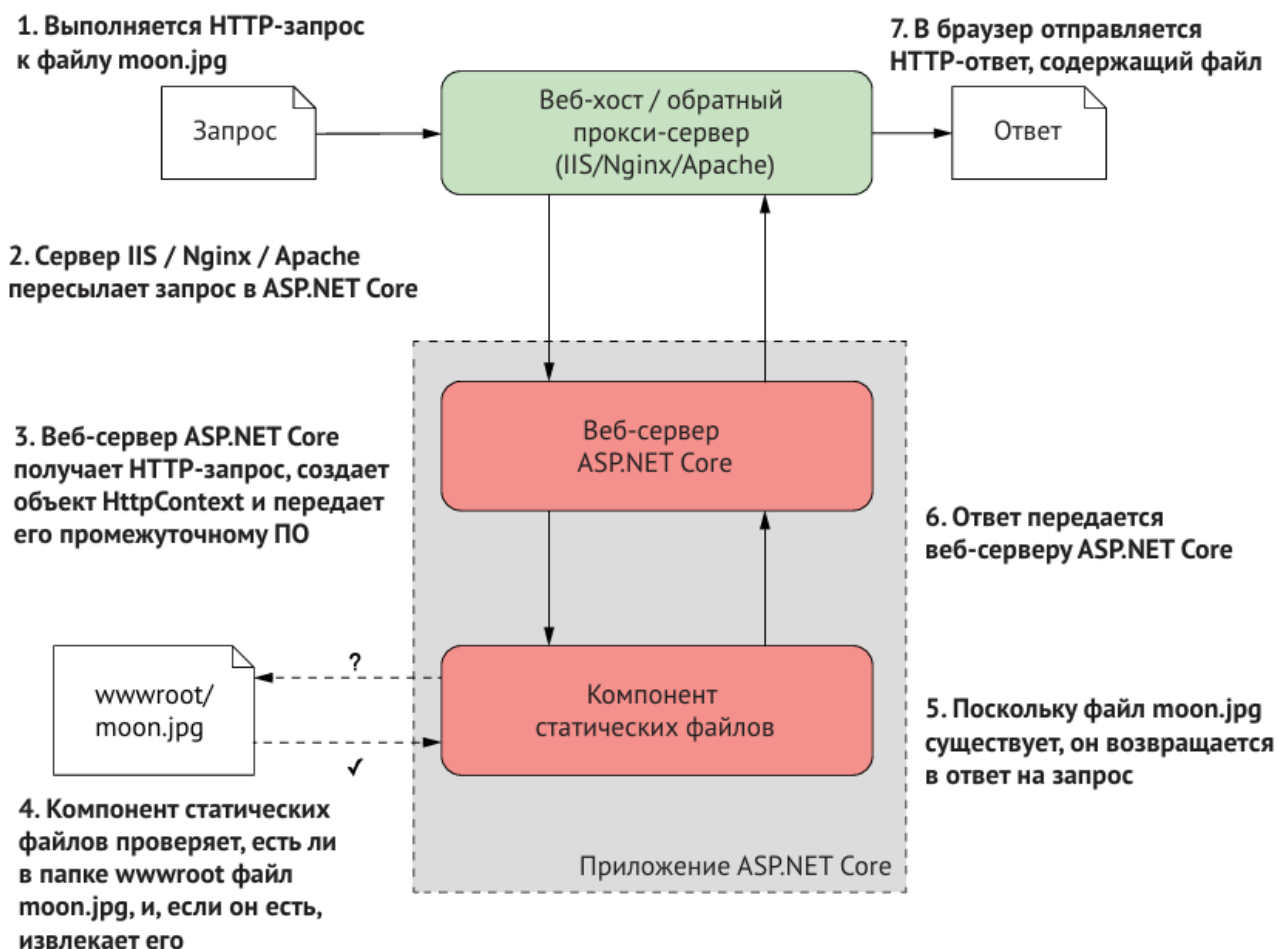
Размещение файлов в этой папке позволяет серверу автоматически обрабатывать HTTP-запросы к ним без дополнительного программного кода. Это упрощает настройку публичного доступа к ресурсам, например, к логотипам или статичным изображениям.

Для настройки работы со статическими ресурсами нужно добавить middleware `app.UseStaticFiles()` в файле `Program.cs`, чтобы активировать раздачу файлов из `wwwroot`.

```
app.UseStaticFiles(); // Разрешает доступ к файлам внутри wwwroot
```

При таком подходе, файл, расположенный по пути `wwwroot/images/photo.jpg`, будет доступен по URL, например:

```
https://example.com/images/photo.jpg
```



Контроллер для загрузки изображений

Контроллер принимает файл, проверяет его наличие, тип и размер, сохраняет файл в определённую папку (например, `wwwroot/images`) и возвращает URL, по которому файл можно будет получить.

Такой подход позволяет централизовать валидацию файлов и гарантировать, что на сервер попадают только допустимые и безопасные файлы. Генерация уникальных имён файлов предотвращает конфликты при сохранении.

```
[ApiController]
[Route("api/[controller]")]
public class FilesController : ControllerBase
{
    [HttpPost("upload")]
    public async Task<IActionResult> UploadImage(IFormFile file)
    {
        // Проверка наличия файла
        if (file == null || file.Length == 0)
            return BadRequest("Файл не выбран");

        // Проверка допустимых форматов
        var allowedExtensions = new[] { ".jpg", ".jpeg", ".png" };
```

```

var extension = Path.GetExtension(file.FileName).ToLowerInvariant();
if (!allowedExtensions.Contains(extension))
    return BadRequest("Недопустимый формат файла");

// Определение папки для загрузки и создание её, если не существует
var uploadsFolder = Path.Combine("wwwroot" "images");
Directory.CreateDirectory(uploadsFolder);

// Генерация уникального имени для файла
var fileName = Guid.NewGuid().ToString() + extension;
var filePath = Path.Combine(uploadsFolder, fileName);

// Сохранение файла на диск
using var stream = new FileStream(filePath, FileMode.Create);
await file.CopyToAsync(stream);

// Формирование URL для доступа к файлу
var fileUrl = $"
{Request.Scheme}://{Request.Host}/images/{fileName}";
return Ok(new { url = fileUrl });
}
}

```

- Контроллер проверяет, что файл был передан и что его формат соответствует разрешённому набору.
- Создаётся папка, если она ещё не существует, чтобы избежать ошибок при сохранении.
- Файл сохраняется, и клиент получает ссылку для дальнейшего использования (например, для отображения загруженного изображения).

Ограничение максимального размера файла

Чтобы избежать чрезмерной загрузки и потенциальных атак, необходимо ограничить максимальный размер загружаемого файла. Это делается с помощью настройки `FormOptions`.

```

builder.Services.Configure<FormOptions>(options =>
{
    options.MultipartBodyLengthLimit = 10 * 1024 * 1024; // Ограничение в 10
    МБ
});

```

Клиентская часть на C# с использованием HttpClient

Клиентское приложение может использовать `HttpClient` для отправки файлов на сервер и последующего их получения. Рассмотрим примеры отправки на сервер и

загрузки изображения с сервера.

Отправка изображения на сервер

Метод клиента создаёт объект `MultipartFormDataContent`, добавляет в него поток файла и отправляет POST-запрос по нужному URL. В ответ клиент получает URL сохранённого изображения.

```
public async Task<string?> UploadPhotoAsync(string filePath)
{
    using var client = new HttpClient();
    using var fileStream = File.OpenRead(filePath);
    using var content = new MultipartFormDataContent();
    content.Add(new StreamContent(fileStream), "file",
        Path.GetFileName(filePath));

    // Отправка запроса на загрузку файла на сервер
    var response = await
client.PostAsync("https://example.com/api/files/upload", content);

    if (!response.IsSuccessStatusCode)
        return null;

    // Десериализация ответа и получение URL изображения
    var result = await response.Content.ReadFromJsonAsync<UploadResponse>();
    return result?.Url;
}

public class UploadResponse
{
    public string Url { get; set; }
}
```

Загрузка и отображение изображения

Метод получает поток данных по URL, преобразует поток в объект `Bitmap` и возвращает его для отображения в приложении.

```
public async Task<Bitmap?> LoadPhotoAsync(string url)
{
    using var client = new HttpClient();
    using var stream = await client.GetStreamAsync(url);
    return await Task.Run(() => Bitmap.DecodeToWidth(stream, 150));
}
```

- Клиент отправляет HTTP GET-запрос к URL, полученному после загрузки файла.

- Полученный поток преобразуется в изображение, которое можно использовать для визуализации в UI приложения.

Ограничение доступа к изображениям

Поскольку по умолчанию все файлы в папке `wwwroot` доступны публично, важно обеспечить защиту чувствительных данных.

Есть два основных подхода:

Хранение изображений вне `wwwroot` и отдача через контроллер

Если изображения содержат конфиденциальные данные, их можно хранить во вне публичной директории (например, в папке `App_Data/Images`). Затем они отдаются через контроллер, где можно провести проверку прав доступа (например, через авторизацию).

```
[Authorize]
[HttpGet("secure-image/{fileName}")]
public IActionResult GetSecureImage(string fileName)
{
    var path = Path.Combine("App_Data", "Images", fileName);
    if (!System.IO.File.Exists(path))
        return NotFound();

    // Поток для чтения файла
    var stream = new FileStream(path, FileMode.Open, FileAccess.Read);
    var mimeType = "image/jpeg"; // можно определять по расширению
    return File(stream, mimeType);
}
```

Клиент получает защищённое изображение:

```
public async Task<Bitmap?> LoadSecureImageAsync(string url, string
bearerToken)
{
    using var client = new HttpClient();
    client.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", bearerToken);
    using var stream = await client.GetStreamAsync(url);
    return await Task.Run(() => Bitmap.DecodeToWidth(stream, 150));
}
```

Ограничение доступа через middleware

Если изображения остаются в папке `wwwroot`, можно добавить промежуточный обработчик (middleware), который будет проверять, авторизован ли пользователь, прежде чем отдавать файлы из определённой папки.

```
app.Use(async (context, next) =>
{
    var path = context.Request.Path.Value;
    if (path != null && path.StartsWith("/images"))
    {
        // Здесь можно реализовать кастомную логику: проверка авторизации,
        // ролей и т.д.
        if (!context.User.Identity.IsAuthenticated)
        {
            context.Response.StatusCode = 403;
            await context.Response.WriteAsync("Access denied");
            return;
        }
    }

    await next();
});

app.UseStaticFiles(); // Раздача статических файлов происходит после
// выполнения middleware
```