{یک یادگیری کارساز}

بوتکمپ استخدامی

ASP.NET Core

Web API

معرفی معماری REST

معماری REST (Representational State Transfer)یک سبک معماری است که برای طراحی و توسعه APIهای وب استفاده می‌شود . این معماری بر اساس اصول و محدودیت‌هایی توسعه یافته که شامل استفاده از منابع یکتا، روش‌های HTTPاستاندارد (مانند GET، POST، PUT، DELETE)، و انتقال اطلاعات با فرمت‌های ساده نظیر JSONیا XMLمی‌باشد.

نکات کلیدی:

- منابع (Resources)در RESTبا URLها نمایش داده می‌شوند.
- از (HTTP Methodsمانند GET، POST، PUT، DELETEبرای عملیات CRUDاستفاده می‌شود.
- معماری REST سبک است و به فناوری خاصی وابسته نیست.

اصول طراحی RESTful API

RESTful APIباید بر اساس اصول مشخصی طراحی شود که این اصول تعامل ساده و مقیاس‌پذیر بین سیستم‌ها را تضمین می‌کنند .مهمترین اصول شامل استفاده از منابع یکتا، روش‌های HTTPاستاندارد، بدون وضعیت بودن (Stateless)، کش‌گذاری (Caching)، و داشتن رابط یکنواخت (Uniform Interface)است.

- اصل Stateless: هر درخواست باید به صورت مستقل پردازش شود.
- اصل Caching: امکان ذخیره‌سازی درخواست‌ها، زمانی که امکان‌پذیر است.
- اصل Interface یکنواخت: ساخت URLساده و قابل فهم.
- اصل ارتباط لایه‌ای (Layered System): جداسازی بین کلاینت و سرور.

منابع (Resources) در REST

در REST هر منبع نماینده‌ای از داده است که می‌توان آن را از طریق URLیکتا شناسایی کرد. منابع حالت ندارند و همانند objectهای مستقل عمل می‌کنند .منابع معمولاً توسط فرمت‌های استاندارد مانند JSONیا XMLارسال می‌شوند.

- منابع باید به روشنی توسط آدرس‌های URLتعریف شوند.
- منابع می‌توانند نماینده‌های مختلفی داشته باشند (مثلاً JSONیا XML).
- عملیات روی منابع از طریق HTTP Methodsانجام می شود.

```
GET /api/products
GET /api/products/1
POST /api/products
PUT /api/products/1
DELETE /api/products/1
```

روش‌های HTTP در RESTful API

RESTful APIاز روش‌های HTTPبرای انجام عملیات مختلف بر روی منابع استفاده می‌کند .پنج روش اصلی شامل GET، POST، PUT، DELETEو PATCHهستند که به ترتیب برای خواندن، ایجاد، به‌روزرسانی، حذف و اصلاح استفاده می‌شوند.

GET: برای دریافت داده‌ها استفاده می‌شود.

POST: برای ایجاد یک منبع جدید استفاده می‌شود.

PUT: برای به‌روزرسانی کل یک منبع استفاده می‌شود.

DELETE: برای حذف یک منبع استفاده می‌شود.

PATCH: برای به‌روزرسانی قسمتی از یک منبع استفاده می‌شود.

```
GET /api/products
POST /api/products
PUT /api/products/1
DELETE /api/products/1
PATCH /api/products/1
```

مزایا و چالش‌های معماری REST

استفاده از معماری RESTمنجر به طراحی یک APIساده، استاندارد، و مقیاس‌پذیر می‌شود .این معماری از قابلیت‌های HTTPبرای انتقال اطلاعات استفاده می‌کند .اما چالش‌هایی نیز وجود دارد، مانند پیچیدگی در مدیریت خطاها و نیاز به طراحی دقیق برای state management.

مزایا:

- سادگی و مقیاس‌پذیری.
- استفاده از استانداردهای HTTPو وب.
- عدم وابستگی به پلتفرم خاص.

چالش‌ها:

- مدیریت stateدر یک سیستم stateless.
- مشکلات امنیتی در ارتباطات باز.
- نیاز به استانداردهای مستندسازی API.

مقدمه‌ای بر API و RESTful

یک API رابطی است که به سیستم‌ها اجازه می‌دهد با یکدیگر تعامل کنند. RESTful API ها از اصول معماری REST برای توسعه APIها استفاده می‌کنند که شامل استفاده از HTTP، منابع و عملیات استاندارد است.

اصول طراحی RESTful API

طراحی یک RESTful API بر پایه استفاده از URLهای منظم و مفهومی، متدهای HTTP استاندارد (مانند GET، POST، PUT، DELETE) و استفاده از فرمت JSON یا XML برای پاسخ‌ها است.

- اصول URL Designing؛
- استفاده از متدهای HTTP؛
- فرمت استاندارد پاسخ.

ایجاد پروژه Web API در ASP.NET Core

از طریق Visual Studio یا CLI می‌توان یک پروژه جدید Web API ایجاد کرد .این پروژه‌ها به طور پیش‌فرض شامل تنظیماتی برای مسیریابی، کنترلرها و پاسخ‌دهی JSON هستند.

نکات کلیدی:

استفاده از Visual Studio یا CLI برای ایجاد پروژه؛ فایل‌های کلیدی مثل Program.cs و appsettings.json.

```
dotnet new webapi -n MySampleAPI
```

ساخت اولین کنترلر

کنترلرها مسئول دریافت درخواست‌ها و بازگردانی پاسخ‌ها هستند .کنترلرها باید از کلاس BaseControllerارث‌بری کنند و می‌توانند از Attributeهایی مثل [Route]و [HttpGet]استفاده کنند.

```
public class ProductsController : ControllerBase {
[HttpGet]
public IActionResult GetProducts() {
 return Ok(new List<string> { "Product1", "Product2" });
}
}
```

عملیات CRUD در RESTful API

برای عملیات CRUD (Create, Read, Update, Delete) باید از متدهای HTTP مرتبط مانند POST، GET، PUT و DELETE استفاده شود .هر متد رابطه مستقیم با عملیات پایگاه داده دارد.

مدیریت خطاها

مدیریت خطاها در Web API اهمیت دارد .کدهای وضعیت HTTP مثل 404 (یافت نشد) یا 500 (خطای سرور) باید بازگردانده شوند.

```
public IActionResult GetProductById(int id) {
 if(id <= 0)
 return BadRequest("Invalid ID");

var product = _productService.GetById(id);

if(product == null)
return NotFound();

return Ok(product);

}
```

استفاده از Swaggerبرای مستندسازی APIها

Swaggerابزار استاندارد صنعتی برای مستندسازی RESTful APIها است .در ASP.NET Coreاز Swashbuckleیا NSwagبرای ادغام Swaggerاستفاده می‌شود.

```
services.AddSwaggerGen();
app.UseSwagger();
app.UseSwaggerUI();
```

گام‌های نهایی و تست API

بعد از ساخت API، باید عملکرد آن را تست کنیم .ابزارهایی مثل Postmanیا فاز تست خود Visual Studioبرای این منظور مفید هستند.

```
GET http://localhost:5000/api/products
```

معرفی Web API در ASP.NET Core

Web APIابزاری است برای ساخت سرویس‌های HTTPکه از طریق پروتکل‌های RESTfulعمل می‌کنند .این سرویس‌ها می‌توانند برای تبادل داده با برنامه‌های کلاینت مانند وب‌سایت‌ها، اپلیکیشن‌های موبایل و ابزارهای دیگر استفاده شوند.

- Web API در ASP.NET Coreابزاری مقیاس‌پذیر برای ساخت سرویس‌های RESTfulاست.
- طراحی ساده و تفکیک واضح مفهومی میان کنترلرها و View.
- پشتیبانی از فرایندهایی مانند Serializationو Middleware.

ساخت Controllerبرای دریافت لیست

در این بخش یاد می‌گیریم که چگونه یک کنترلر Web APIبرای دریافت لیست بسازیم .نمونه لیست می‌تواند شامل داده‌های کاربران، محصولات، یا هر دیتاست دیگری باشد.

- نامگذاری مناسب کنترلر بر اساس مدل دیتا.
- استفاده از Routeبرای تنظیم مسیر .API
- بازگشت لیست آیتم‌ها با استفاده از .IActionResult

```
[ApiController]
[Route("api/[controller]")]
public class ItemsController : ControllerBase
{
    private static readonly List<string> Items = new List<string> { "Item1", "Item2", "Item3" };

    [HttpGet]
    public IActionResult GetItems()
    {
        return Ok(Items);
    }
}
```

کار با Endpointبرای دریافت آیتم‌ها

اضافه کردن Endpointبه کنترلر برای فراخوانی سرویس و برگرداندن لیست آیتم‌ها.

```
[ApiController]
[Route("api/[controller]")]
public class ItemsController : ControllerBase
{
    private readonly IItemService _itemService;

    public ItemsController(IItemService itemService)
    {
        _itemService = itemService;
    }

    [HttpGet]
    public IActionResult GetItems()
    {
        var items = _itemService.GetItems();
        return Ok(items);
    }
}
```

نکات کلیدی:

- استفاده از Injectکردن وابستگی‌ها در Constructor.
- فراخوانی سرویس از Data Layer.
- بازگشت داده‌ها به صورت JSONاز طریق IActionResult.

افزودن Validationو مدیریت خطا

اعتبارسنجی داده‌های ورودی برای Endpointها و مدیریت خطاها برای بازگشت پاسخ‌های متناسب به کلاینت.

- مدیریت پاسخ‌های مختلف (مثلاً 404و 500).
- اعتبارسنجی داده‌های خروجی.
- استفاده از بلوک Try-Catchبرای مدیریت استثناها.

```
[HttpGet]
public IActionResult GetItems()
{
    try
    {
        var items = _itemService.GetItems();
        if (items == null || items.Count == 0)
        {
            return NotFound("No items found.");
        }
        return Ok(items);
    }
    catch (Exception ex)
    {
        return StatusCode(500, "Internal server error: " + ex.Message);
    }
}
```

ایجاد API برای دریافت جزئیات

در این درس یاد می‌گیریم که چگونه یک API برای دریافت جزئیات یک موجودیت (مانند محصول، کاربر، یا سفارش) پیاده‌سازی کنیم.

1. استفاده از [HttpGet("{id}")] برای دریافت پارامتر از URL.
2. بررسی مقادیر ورودی و بازگرداندن NotFound اگر داده‌ای موجود نباشد.
3. استفاده از LINQ برای جستجوی داده‌ها از پایگاه‌داده.

```
[HttpGet("{id}")]
public IActionResult GetDetail(int id)
{
  var product = _context.Products.SingleOrDefault(p => p.Id == id);
  if (product == null)
    return NotFound();

  return Ok(product);
}
```

مدیریت درخواست‌ها و پاسخ‌های API

مدیریت درست درخواست‌ها و پاسخ‌ها، بخش ضروری در طراحی APIاست .بررسی موفقیت‌آمیز بودن درخواست‌ها و بازگرداندن خطاهای مناسب تضمین‌کننده تجربه کاربری بهتر خواهد بود.

1. استفاده از BadRequestبرای مواردی که ورودی‌ها نامعتبر هستند.
2. NotFoundبرای داده‌هایی که وجود ندارند.
3. Ok برای بازگرداندن پاسخ موفق.

.

```
[HttpGet("{id}")]
public IActionResult GetDetail(int id)
{
  if (id <= 0)
    return BadRequest("Invalid ID");

  var product = _context.Products.SingleOrDefault(p => p.Id == id);
  if (product == null)
    return NotFound("Product not found");

  return Ok(product);
}
```

بهینه‌سازی پاسخ‌های API

برای ارائه بهترین تجربه به کاربران، نیاز است که پاسخ‌های APIرا به صورت بهینه و مطابق نیاز کاربر تنظیم کنیم .می‌توانیم فیلتر، صفحه‌بندی و یا تبدیل داده‌ها را انجام دهیم.

1. کاهش داده‌های ارسال شده با استفاده از LINQ.
2. بازگرداندن تنها خصوصیات مورد نیاز از مدل‌ها.
3. افزایش سرعت پاسخ با محدود کردن داده‌ها.

در مثال زیر، فقط اطلاعاتی که کاربر نیاز دارد بازگردانده می‌شود:

```
[HttpGet("{id}")]
public IActionResult GetDetail(int id)
{
  var product = _context.Products
    .Where(p => p.Id == id)
    .Select(p => new { p.Name, p.Price, p.Description })
    .SingleOrDefault();

  if (product == null)
    return NotFound();

  return Ok(product);
}
```

افزودن مستندات Swagger

Swaggerبه شما اجازه می‌دهد مستندات APIرا برای تیم‌های توسعه یا مشتریان به صورت خودکار ایجاد کنید .با استفاده از بسته Swashbuckle، این امر بسیار آسان است.

1. نصب و اضافه کردن بسته Swashbuckleبه پروژه.
2. استفاده از فایل‌های Swaggerبرای مستندات خودکار .API
3. فراهم کردن تست آسان‌تر APIبرای توسعه‌دهندگان.

نمونه کدی برای افزودنSwagger:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddControllers();
  services.AddSwaggerGen();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
  app.UseSwagger();
  app.UseSwaggerUI(c => { c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1"); });
}
```

ساختار یک Web API

Web APIها شامل endpointهای مختلفی هستند که هر یک با استفاده از روش‌های مختلف HTTP (GET, POST, PUT, DELETE) امکان ارتباط با سرور را فراهم می‌کنند.

```
GET /api/products --> بازگرداندن لیست محصولات
POST /api/products --> ثبت اطلاعات محصول جدید
```

طراحی مدل داده

مدل داده، ساختار اشیا یا داده‌هایی که در APIمدیریت می‌شوند را تعریف می‌کند .معمولاً مدل‌ها به صورت کلاس‌های C#طراحی می‌شوند.

```
public class Product {
 public int Id { get; set; }
 public string Name { get; set; }
 public decimal Price { get; set; }
}
```

اتصال به بانک اطلاعاتی

برای ذخیره اطلاعات ارسال‌شده توسط کاربر، اتصال به بانک اطلاعاتی ضروری است .با استفاده از Entity Framework، این اتصال ساده‌تر می‌شود.

تنظیم کانکشن‌استرینگ،

استفاده از EF Coreبرای مدیریت داده

```
services.AddDbContext<ApplicationDbContext>(options =>
 options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

پیاده‌سازی Apiثبت اطلاعات جدید

فرآیند ثبت اطلاعات جدید در پایگاه داده با استفاده از یک متد POST:

استفاده از async/awaitبرای ذخیره اطلاعات،

بازگشت پاسخ مناسب با HTTP 201

```
public class ProductsController : ControllerBase {
 [HttpPost]
 public async Task<IActionResult> Create(Product product) {
 await _context.Products.AddAsync(product);
 await _context.SaveChangesAsync();
 return CreatedAtAction("Get", new { id = product.Id }, product);
 }
```

اعتبارسنجی ورودی‌ها

برای اطمینان از صحت داده‌های ورودی در متد ویرایش، می‌توان از ویژگی‌های Data Annotationو همچنین Fluent Validationاستفاده کرد.

نکات کلیدی:

- استفاده از [Required]برای اجباری بودن مقدار.
- استفاده از [Range]برای تعیین محدودهٔ عددی.
- نقش Fluent Validationدر توسعه معتبرتر.

```
public class Product
{
    [Required]
    public string Name { get; set; }

    [Range(1, 1000)]
    public decimal Price { get; set; }
}
```

پاسخ‌های مناسب به کلاینت

یک APIباید پاسخ‌های HTTPدرست و قابل فهم به کلاینت بازگرداند .این شامل استفاده از کدهای وضعیت(Status Codes) مانند 200، 404و 400می‌شود.

- استفاده از NotFoundبرای مواردی که داده موجود نیست.
- استفاده از Okبرای بازگرداندن پاسخ موفق.
- قابلیت شخصی‌سازی پیام‌های خطا.

```
if (product == null)
{
    return NotFound(new { Message = "محصول یافت نشد." });
}
return Ok(product);
```

مروری بر HATEOASو RESTful APIs

HATEOAS (Hypermedia as the Engine of Application State) بخشی کلیدی از طراحی RESTful APIs است که به کلاینت‌ها اجازه می‌دهد بدون نیاز به دانش اولیه در مورد Endpointsسیستم، تعامل کنند.

RESTful APIs شامل سطوح مختلف بلوغ در مدل RMM (Richardson Maturity Model)است که سطح 3به استفاده از HATEOASاشاره دارد.

مدل بلوغ ریچاردسون (RMM)

مدل بلوغ ریچاردسون استانداردی برای طراحی RESTful APIs است که به 4 سطح تقسیم‌بندی می‌شود:

سطح 0 (Endpoints ساده)

سطح1 (استفاده از منابع)

سطح2 (استفاده از HTTP Methods)

سطح 3 (HATEOAS)

مفاهیم اصلی HATEOAS

در سطح 3 RMM، هر پاسخ APIمی‌تواند شامل لینک‌هایی (به نام Hypermedia Links) به منابع مرتبط یا اقدامات بعدی باشد .این رویکرد از وابستگی کلاینت به ساختار داخلی APIجلوگیری می‌کند.

- لینک‌های Hypermediaدر پاسخ
- کاهش وابستگی ساختاری کلاینت به API
- انعطاف‌پذیری بیشتر در توسعه

```
HTTP/1.1 200 OK
{
  "id": 123,
  "name": "کاربر نمونه",
  "links": [
    { "rel": "self", "href": "/users/123" },
    { "rel": "orders", "href": "/users/123/orders" }
  ]
}
```

پیاده‌سازی HATEOAS در ASP.NET Core

برای افزودن HATEOAS به API، شما باید لینک‌های Hypermediaرا به پاسخ‌های JSONخود اضافه کنید .از کلاس‌های انتخابگر لینک و تزریق وابستگی در ASP.NET Coreاستفاده کنید.

- مدیریت لینک‌ها با یک مدل جداگانه
- اضافه کردن لینک‌ها به پاسخJSON
- ایجاد منابع انعطاف‌پذیر

```
public class UserResource
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Link> Links { get; set; } = new List<Link>();
}

[HttpGet("api/users/{id}")]
public IActionResult GetUser(int id)
{
    var user = _userService.GetUserById(id);
    if (user == null) return NotFound();

    var resource = new UserResource
    {
        Id = user.Id,
        Name = user.Name,
        Links = new List<Link>
        {
            new Link { Rel = "self", Href = $"/api/users/{id}" },
            new Link { Rel = "orders", Href = $"/api/users/{id}/orders" }
        }
    };

    return Ok(resource);
}
```

مروری بر Status Codeهای HTTP

Status Codeهای HTTPکدهایی هستند که به پاسخ‌های سرویس‌دهنده وب به درخواست کلاینت‌ها توصیف می‌کنند .این کدها معمولاً بیانگر وضعیت موفقیت‌آمیز یا خطای درخواست‌ها هستند و در ۵دسته اصلی قرار می‌گیرند:

1xx:برای اطلاعات اضافی استفاده می‌شود

2xx :درخواست با موفقیت پردازش شده است.

3xx :کلاینت نیاز به اقدامات بیشتر (مانند ریدایرکت) دارد.

4xx: خطاهای مربوط به کلاینت (مانند 404 .(Not Found

5xx: خطاهای سمت سرور (مانند 500 (Internal Server Error

```
HTTP/1.1 200 OK
Content-Type: application/json
{"message": "درخواست موفقیت‌آمیز بود"}
```

دسته‌بندی Status Codeهای HTTP

1. کدهای xx1: اطلاعاتی Informational

100 Continue : درخواست اولیه دریافت شده و کلاینت می‌تواند ادامه دهد.

101 Switching Protocols: سرور درخواست تغییر پروتکل را پذیرفته است.

2. کدهای xx2: موفقیت Successful

200 OK: درخواست با موفقیت پردازش شده و پاسخ در بدنه موجود است.

201 Created: درخواست با موفقیت پردازش شده و منبع جدیدی ایجاد شده است.

204 No Content: درخواست موفقیت‌آمیز بوده، اما هیچ محتوایی برای بازگشت وجود ندارد.

دسته‌بندی Status Codeهای HTTP

3. کدهای 3xx: هدایت Redirection

Moved Permanently 301: منبع به طور دائمی به آدرس جدیدی منتقل شده است.

Found 302: منبع به طور موقت به آدرس جدیدی منتقل شده است.

Not Modified 304: منبع تغییر نکرده و کلاینت می‌تواند از کش محلی استفاده کند.

4. کدهای 4xx: خطاهای کلاینت Client Errors

Bad Request 404: درخواست به دلیل خطا در نحوه ساختار آن قابل پردازش نیست.

Unauthorized 401: درخواست نیاز به احراز هویت دارد.

Forbidden 403: سرور درخواست را رد کرده و کلاینت مجاز به دسترسی به منبع نیست.

Not Found 404: منبع مورد نظر پیدا نشد.

دسته‌بندی Status Codeهای HTTP

5. کدهای xx5: خطاهای سرور Server Errors

Internal Server Error500 : خطای عمومی در سرور.

Bad Gateway 502: سرور به عنوان دروازه یا پروکسی، پاسخ نامعتبری از سرور بالادستی دریافت کرده است.

Service Unavailable 503: سرور در حال حاضر قادر به پردازش درخواست نیست (معمولاً به دلیل بار زیاد یا نگهداری).

معرفی Versioning در Web API

یکی از ویژگی‌های مهم در توسعه Web API، مدیریت نسخه‌ها (Versioning)است .این قابلیت به توسعه‌دهندگان اجازه می‌دهد تا تغییرات جدیدی را بدون از بین بردن سازگاری با نسخه‌های قدیمی‌تر اعمال کنند .این رویکرد برای پروژه‌های مقیاس بزرگ و سیستم‌های در حال توسعه حیاتی است.

- اهمیت استفاده از Versioning
- قابلیت مدیریت نسخه‌ها در حفظ سازگاری
- روش‌های مختلف پیاده‌سازی Versioning

روش‌های پیاده‌سازی Versioning در Web API

سه روش اساسی برای پیاده‌سازی نسخه‌بندی در APIها وجود دارد:

1. نسخه‌بندی مبتنی بر URI
2. نسخه‌بندی مبتنی بر Header
3. نسخه‌بندی مبتنی بر Query String

پیاده‌سازی نسخه‌بندی مبتنی بر URI

در این روش، نسخه APIبا اضافه کردن نسخه در مسیر URIمشخص می‌شود .برای مثال /api/v1/products :

نکات کلیدی:

1. روش ساده و قابل فهم برای کاربران API
2. مناسب برای پروژه‌هایی که نسخه‌بندی اولیه نیاز دارند
3. ممکن است مدیریت زیاد مسیرها در پروژه‌های بزرگ دشوار شود

```
app.MapGet("/api/v1/products", () => "Version 1 API");
```

پیاده‌سازی نسخه‌بندی مبتنی بر Header

در این روش، نسخه APIاز طریق Headerدرخواست مشخص می‌شود .این روش معمولاً انعطاف‌پذیرتر است ولی نیازمند پیکربندی بیشتری می‌باشد.

1. مناسب برای APIهای پیشرفته و کاربران حرفه‌ای
2. ساختار درخواست ساده باقی می‌ماند
3. کاربران نیازمند تنظیمات اضافی در درخواست خود هستند

```
services.AddApiVersioning(options => {
    options.ApiVersionReader = new HeaderApiVersionReader("x-api-version");
});
```

پیاده‌سازی نسخه‌بندی مبتنی بر Query String

در این روش، نسخه APIاز طریق پارامتر Query Stringدر URLدرخواست تعیین می‌شود.

برای مثال: /api/products?api-version=1

1. مناسب برای پروژه‌هایی که کاربران عمومی و گسترده دارند
2. راحت در استفاده توسط کاربران و توسعه‌دهندگان
3. ممکن است URLهای طولانی و پیچیده ایجاد شود

```
services.AddApiVersioning(options => {
    options.ApiVersionReader = new QueryStringApiVersionReader("api-version");
});
```

مقایسه و انتخاب بهترین روش Versioning

هر کدام از روش‌های نسخه‌بندی مزایا و محدودیت‌های خاص خود را دارند .انتخاب روش مناسب به نیازهای پروژه، کاربران هدف و الزامات فنی بستگی دارد.

1. مقایسه مزایا و معایب روش‌های URI، Headerو Query String

2 .بررسی نیازهای پروژه برای انتخاب بهترین روش

3 .امکان ترکیب روش‌ها در برخی پروژه‌ها

مقدمه‌ای بر Swagger

Swaggerیک ابزار قدرتمند برای مستندسازی، تولید و تست Web APIها می‌باشد .این ابزار به تیم‌ها کمک می‌کند تا APIها را بهتر مستند کرده و به توسعه‌دهندگان اجازه می‌دهد بدون نیاز به کد واقعی، APIرا تست کنند.

- مستندسازی API
- تست و بررسی Endpointها
- اعتبارسنجی ورودی‌ها و خروجی‌ها

نصب و اضافه کردن Swaggerبه پروژه

برای استفاده از Swaggerباید کتابخانه Swashbuckle.AspNetCoreرا نصب کنیم .این کتابخانه به سادگی Swaggerرا به پروژه ASP.NET Coreما اضافه می‌کند.

- نصب با دستور Nugetیا Package Manager
- اطمینان از اضافه شدن صحیح Dependency
- همخوانی نسخه کتابخانه با ASP.NET Core

```
Install-Package Swashbuckle.AspNetCore
```

پیکربندی اولیه Swagger

بعد از نصب، باید Swaggerرا در متد Startup.csپیکربندی کنیم .این شامل افزودن سرویس Swaggerدر متد ConfigureServicesو استفاده از آن در متد Configureمی‌باشد.

- استفاده از متد AddSwaggerGen
- تعریف داکیومنت با OpenApiInfo
- راه اندازی Swagger UI

```
services.AddSwaggerGen(c => {
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });
});
...
app.UseSwagger();
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API v1");
});
```

ساختار Swagger UI

Swagger UIبه صورت خودکار مستندات APIشما را تولید می‌کند و یک رابط گرافیکی ارائه می‌دهد که قابل استفاده و تعامل است.

- دسترسی به Swagger UIبا مسیر پیش‌فرض /swagger
- قابلیت مشاهده و تست تمامی Endpointها در محیط گرافیکی
- نمایش پارامترها و توضیحات مربوط به هر Endpoint

سفارشی‌سازی Swagger

Swaggerقابلیت‌های سفارشی‌سازی مختلفی ارائه می‌دهد .می‌توانید نام مستند را تغییر داده، فیلتر اضافه کنید یا توضیحات بیشتری برای Endpointها ارائه دهید.

- تغییر عنوان و توضیحات در تنظیمات SwaggerDoc
- افزودن توضیحات دقیق برای Controllersو Actions
- استفاده از Attributeهایی مانند [ApiExplorerSettings]

```
c.SwaggerDoc("v1", new OpenApiInfo {
    Title = "Customized API",
    Version = "v1",
    Description = "This is a custom API documentation example."
});
```

معرفی Json Web Token (JWT)

Json Web Token (JWT)یک استاندارد باز برای ایجاد توکن‌هایی است که اطلاعات را به صورت JSONبین دو طرف به صورت امن انتقال می‌دهند .این توکن‌ها معمولاً برای تأیید هویت و احراز دسترسی کاربران در برنامه‌های وب و APIها استفاده می‌شوند.

۱ JWT .متشکل از سه بخش اصلی Header، Payload، و Signatureاست .

۲. ایمن‌سازی دیتا از طریق امضای توکن صورت می‌گیرد .

۳. JWTبه فرم Base64URLکدگذاری می‌شود.

ساختاری به صورت زیر دارد JSON Web Token:
Header.Payload.Signature
مثال یک توکن واقعی:
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiO
iIxMjM0NTY3ODkwIiwibmFtZSI6IkpvbiBEb2UiLCJpYXQ
iOjE1MTYyMzkwMjJ9.SflKxwRJSMeKKF2QT4fwpMeJf36P
Ok6yJV_adQssw5c
```

ساختار JWT

JWTاز سه بخش Header، Payload، و Signatureتشکیل شده است .

۱ :Header .تعیین الگوریتم امضا مانند HS256یا RS256و نوع توکن.

۲ :Payload .شامل اطلاعات اصلی یا Claims (موارد اختیاری اما کلیدی مانند شناسه کاربری، زمان انقضا و غیره).

۳. :Signatureبرای تأیید صحت JWTو جلوگیری از تغییر داده‌ها.

- Header و Payloadبه صورت Base64URLکدگذاری می‌شوند .
- Signatureبرای تأیید عدم تغییر داده‌ها استفاده می‌شود .
- هر سه بخش با نقطه (.)جدا می‌شوند.

مثال Header:

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

مثال Payload:

```
{
 "sub": "1234567890",
 "name": "John Doe",
 "iat": 1516239022
}
```

امضای JWTبا الگوریتم HMAC-SHA256:

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

یادداشت‌های ارائه دهنده:

به طور دقیق ساختار هر بخش را توضیح دهید و روی کاربرد واقعی هر کدام از آن‌ها تمرکز داشته باشید.

مزایا و معایب JWT

مزایا :

۱ .ساده بودن فرمت و جابجایی آسان بین سرویس‌ها.

۲ .قابلیت عدم وابستگی به سرور در ذخیره وضعیت احراز هویت.

۳ .امکان تنظیم تاریخ انقضا و کنترل بهتر دسترسی.

معایب:

۱ .گاهی حجم بالا به دلیل اجزای اطلاعات ذخیره شده .

۲ .عدم قابلیت لغو اعتبار JWTپیش از انقضا مگر با استفاده از لیست سیاه.

۳ .پیچیدگی در مدیریت موارد امنیتی.

مزایا از دید کدنویسی، ارسال JWTدر Headerدرخواست‌ها :

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

نکته :بررسی زمان انقضای توکن پیش از قبول درخواست کاربر:

```
if (jwt.IsExpired()) {
    return 401;
}
```

تولید و استفاده از JWTدر ASP.NET Core

در ASP.NET Coreمی‌توان با استفاده از کتابخانه Microsoft.AspNetCore.Authentication.JwtBearerمعتبرسازی JWTرا به صورت ساده پیاده کرد .ابتدا باید یک Secret Keyو الگوریتم امضا تعیین شود و سپس JWTتولید و در هنگام درخواست بعدی کاربر اعتبارسنجی شود.

۱ .از Microsoft.AspNetCore.Authentication.JwtBearerبرای احراز هویت JWTاستفاده کنید .

۲ Secret Key .باید قوی و طولانی باشد .

۳ .تاریخ انقضا را به دقت تنظیم کنید.

اضافه کردنJWT به Startup.cs:

```
services.AddAuthentication(options => {
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options => {
    options.TokenValidationParameters = new TokenValidationParameters {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = "yourdomain.com",
        ValidAudience = "yourdomain.com",
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKey"))
    };
});
```

نصب کتابخانه‌های مورد نیاز

برای کار با JWT در ASP.NET Core نیاز به نصب بسته‌های JWT از طریق NuGet داریم. این بسته‌ها شامل Microsoft.AspNetCore.Authentication.JwtBearer هستند.

۱. از NuGet استفاده کنید.

۲. بسته‌های مورد نیاز باید در بخش مدیریت بسته‌ها نصب شوند.

```
Install-Package Microsoft.AspNetCore.Authentication.JwtBearer
```

تنظیمات JWTدر فایل Startup.cs

جهت استفاده از JWTباید تنظیماتی در فایل Startup.csانجام شود .این تنظیمات شامل افزودن Middlewareاحراز هویت و مشخص کردن پارامترهای امنیتی است.

۱ Middleware .احراز هویت اضافه شود.

۲ .کلید محرمانه برای تطبیق توکن باید قوی باشد.

۳ .تنظیمات مربوط به صادر کننده (Issuer)و مخاطب (Audience)ضروری است.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "https://yourdomain.com",
            ValidAudience = "https://yourdomain.com",
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKey"))
        };
    });
```

چگونگی ایجاد یک JWT Token

توکن JWTشامل اطلاعات کاربر است که با استاندارد JSONرمزنگاری شده است .برای تولید یک JWTاز توابع کتابخانه Microsoft.IdentityModel.Tokensاستفاده می‌کنیم.

۱ .رمزنگاری اطلاعات کاربران ضروری است.

۲ .زمان اعتبار توکن مشخص شود.

۳ SigningCredentials .مقداردهی شود.

```
var tokenHandler = new JwtSecurityTokenHandler();
var key = Encoding.ASCII.GetBytes("YourSecretKey");
var tokenDescriptor = new SecurityTokenDescriptor
{
    Subject = new ClaimsIdentity(new Claim[]
    {
        new Claim(ClaimTypes.Name, "UserId")
    }),
    Expires = DateTime.UtcNow.AddHours(2),
    SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key), SecurityAlgorithms.HmacSha256Signature)
};
var token = tokenHandler.CreateToken(tokenDescriptor);
var tokenString = tokenHandler.WriteToken(token);
```

استفاده از JWT Token در درخواست‌ها

JWT Tokenپس از تولید می‌تواند به عنوان Bearer Tokenدر درخواست‌های HTTPپاس داده شود .سمت سرور توکن را بررسی کرده و اهراز هویت انجام می‌دهد.

۱ .توکن باید در Headerدرخواست قرار گیرد.

۲ .مقدار Authorizationبه فرم Bearerباشد.

۳ .سمت سرور توکن اعتبارسنجی شود.

```
fetch('https://yourapi.com/api/data', {
    method: 'GET',
    headers: {
        'Authorization': 'Bearer ' + tokenString
    }
})
.then(response => response.json())
.then(data => console.log(data));
```

استفاده از Tokenدر Headerدرخواست‌ها

ارسال Token (مانند JSON Web Token) از طریق Headerدرخواست‌های HTTP(مانند Authorization) یکی از رایج‌ترین روش‌ها برای احراز هویت کاربران است.

- از پروتکل HTTPSبرای ایمن کردن ارسال Tokenاستفاده کنید.
- زمان انقضای Tokenرا برای امنیت بیشتر در نظر بگیرید.

```
//ارسال Tokenدر Header
fetch('https://api.example.com/protected-resource', {
  method: 'GET',
  headers: {
    'Authorization': 'Bearer your-token-here',
    'Content-Type': 'application/json',
  },
}).then(response => response.json())
  .then(data => console.log(data));
```

تنظیم Refresh Tokenو مدیریت Sessions

Refresh Tokenبرای تمدید دسترسی کاربران بدون درخواست مجدد نام کاربری و رمز عبور استفاده می‌شود .این Tokenمعمولاً طول عمر بیشتری نسبت به Access Tokenدارد و باید با روش مطمئنی ذخیره شود.

- استفاده از Refresh Tokensبرای جلوگیری از خستگی کاربر هنگام ورود مکرر.
- اطمینان حاصل کنید که Refresh Tokenبه صورت امن ذخیره شود.
- از رویدادهای امنیتی مانند لغو دستی Refresh Tokenپشتیبانی کنید.

```
app.MapPost("/refresh-token", async (HttpContext context) => {
    var refreshToken = context.Request.Cookies["refresh_token"];
    if (string.IsNullOrEmpty(refreshToken)) {
        return Results.Unauthorized();
    }

    //  بررسیRefresh Token
    var isValid = ValidateRefreshToken(refreshToken);
    if (!isValid) {
        return Results.Unauthorized();
    }

    //  ایجادAccess Token جدید
    var newAccessToken = GenerateAccessToken();
    return Results.Json(new { accessToken = newAccessToken });
});
```

جمع‌بندی ذخیره Token

در این بخش یاد گرفتید که چطور Tokenرا به روش‌های مختلف ذخیره کنید و برای امنیت بیشتر از روش‌های امن مثل Cookiesیا Header Authorizationاستفاده نمایید.

مقدمه Refresh Token

در این بخش با مفهوم Refresh Tokenو دلایل استفاده از آن در معماری احراز هویت آشنا می‌شویم. Refresh Token به ما کمک می‌کند تا بدون نیاز به ارسال دوباره اطلاعات حساس کاربر، دسترسی طولانی‌مدت به سیستم داشته باشیم.

۱. Refresh Tokenمعمولاً در کنار Access Tokenاستفاده می‌شود.

2. برای افزایش امنیت، Refresh Tokenباید در سمت سرور مدیریت شود.

3. طول عمر Refresh Tokenبیشتر از Access Tokenاست.

ساختار Refresh Token

در Refresh Tokenبه مواردی مانند شناسایی کاربر، زمان انقضا و کد امنیتی خاص نیاز است .این ساختار معمولاً در سمت سرور تولید و مدیریت می‌شود.

1. Refresh Tokenباید تصادفی و غیرقابل حدس باشد.
2. نیاز به الگوریتمی امن مثل RNGبرای تولید توکن.
3. ذخیره‌سازی امن در پایگاه داده ضروری است.

```
string GenerateRefreshToken()
{
  var randomNumber = new byte[32];
  using (var rng = RandomNumberGenerator.Create())
  {
    rng.GetBytes(randomNumber);
  }
  return Convert.ToBase64String(randomNumber);
}
```

ذخیره و مدیریت Refresh Token

Refresh Tokenباید در یک پایگاه داده امن با اطلاعات کاربر ثبت شود .از هش کردن داده‌ها برای افزایش امنیت استفاده کنید.

۱ .استفاده از تاریخ انقضا برای Refresh Token.

۲ .محدود کردن توکن‌ها به کاربر خاص.

۳ .حذف Refresh Tokenهای منقضی شده به صورت دوره‌ای.

```
public async Task SaveRefreshToken(string userId, string refreshToken)
{
    var token = new RefreshToken {
        UserId = userId,
        Token = refreshToken,
        ExpiryDate = DateTime.UtcNow.AddDays(7)
    };
    _dbContext.RefreshTokens.Add(token);
    await _dbContext.SaveChangesAsync();
}
```

تبادل Refresh Token و Access Token

در فرآیند تبادل، کاربر با ارسال Refresh Token به API، درخواست Access Token جدید می‌کند. سرور با بررسی اعتبار Refresh Token، تصدیق را انجام داده و در صورت معتبر بودن، Access Token جدید بازگردانده می‌شود.

۱. بررسی معتبر بودن Refresh Token.

۲. استفاده از تاریخ انقضا برای جلوگیری از سوءاستفاده.

۳. توزیع Access Token جدید فقط در صورت تایید اعتبار.

```
public async Task<IActionResult> RefreshToken(string token)
{
    var refreshToken = await _dbContext.RefreshTokens.FirstOrDefaultAsync(r => r.Token == token);
    if (refreshToken == null || refreshToken.ExpiryDate < DateTime.UtcNow)
    {
        return Unauthorized();
    }
    var newAccessToken = GenerateAccessToken(refreshToken.UserId);
    return Ok(new { AccessToken = newAccessToken });
}
```

حذف Refresh Tokenهای منقضی شده

به منظور حفاظت از سرور، Refresh Tokenهای منقضی شده باید به صورت دوره‌ای از پایگاه داده حذف شوند.

1. Refresh Tokenهای تاریخ انقضا گذشته نباید در سیستم باقی بمانند.
2. بهینه‌سازی عملکرد پایگاه داده با حذف داده‌های بلااستفاده.
3. پیاده‌سازی مکانیزم‌های اتوماتیک جهت پاکسازی دوره‌ای.

```
public async Task RemoveExpiredTokens()
{
    var expiredTokens = _dbContext.RefreshTokens
        .Where(r => r.ExpiryDate < DateTime.UtcNow).ToList();

    _dbContext.RefreshTokens.RemoveRange(expiredTokens);
    await _dbContext.SaveChangesAsync();
}
```

نکات امنیتی مرتبط با Refresh Token

استفاده از Refresh Tokenباید با رعایت پروتکل‌های امنیتی همراه باشد تا از سوءاستفاده جلوگیری شود.

۱. ذخیره Refresh Tokenدر HttpOnly Cookie برای جلوگیری از دسترسی جاوا اسکریپت.

۲. استفاده از HTTPSبرای رمزنگاری درخواست‌ها.

۳. محدودسازی Refresh Tokenبرای IPو دستگاه مشخص.

معرفی Jwt Token و مفهوم Logout

Jwt Token به طور پیش‌فرض stateful نیست و باید مکانیزم Logout برای مسدودسازی توکن‌های صادرشده پیاده‌سازی شود.

راهکارهای پیاده‌سازی Logoutبرای Jwt Token

برای پیاده‌سازی Logoutمعمولاً از تکنیک‌هایی مانند Blacklistingیا کاهش عمر توکن استفاده می‌شود.

دو راهکار اصلی :Logout

1. Blacklisting توکن‌ها.
2. کوتاه کردن زمان انقضا (TTL)توکن.

پیاده‌سازی Blacklisting برای Jwt Token

Blacklisting شامل ذخیره توکن‌های نامعتبر (تمام یا جزیی) در یک پایگاه داده است.

نیاز به پایگاه داده برای ذخیره توکن‌های مسدود شده.

```
public async Task Logout(string token) {
    await _dbContext.BlacklistedTokens.AddAsync(new BlacklistedToken {
        Token = token,
        ExpirationDate = GetJwtExpirationDate(token)
    });
    await _dbContext.SaveChangesAsync();
}
```

کوتاه کردن مدت زمان انقضای Jwt Token

در این روش مدت زمان انقضای توکن‌ها کاهش پیدا می‌کند و نیازمند Refresh Tokenبرای تمدید است.

کاهش زمان انقضا امنیت بیشتری ایجاد می‌کند اما تجربه کاربری را ممکن است کاهش دهد.

```
services.AddAuthentication(options => {
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options => {
    options.TokenValidationParameters = new TokenValidationParameters {
        ValidateLifetime = true,
        ClockSkew = TimeSpan.FromMinutes(1) // کاهش عمر انقضا
    };
});
```

آشنایی با RestSharp

RestSharpیک کتابخانه قدرتمند برای تعامل با APIهای RESTfulمی‌باشد .این ابزار در کلاینت‌های C#به ما کمک می‌کند درخواست‌های HTTPرا به شکل ساده ارسال و پاسخ‌ها را مدیریت کنیم.

- ساختار خوانا
- قابلیت درخواست HTTPساده (GET, POST, PUT, DELETE)
- مدیریت پاسخ‌های JSONو XMLبه آسانی
- پشتیبانی از احراز هویت

```
var client = new RestClient("https://api.example.com");
var request = new RestRequest("/endpoint", Method.Get);
RestResponse response = await client.ExecuteAsync(request);
Console.WriteLine(response.Content);
```

ساختار اصلی یک درخواست در RestSharp

برای ارسال یک درخواست HTTPبا استفاده از RestSharp، ابتدا باید یک نمونه از RestClientایجاد کنید .سپس با استفاده از RestRequestمشخص کنید که به کدام endpointدرخواست ارسال شود و نوع متد HTTP (GET, POSTو غیره (چه باشد .در نهایت درخواست را با متد Executeارسال کرده و پاسخ مورد نظر را دریافت کنید.

1. ابتدا باید یک RestClientبرای ارتباط با APIساخته شود.
2. متد HTTPمشخص می‌شود.
3. هدرها و یا Bodyدرخواست‌ها قابل اضافه شدن هستند.
4. پاسخ دریافت شده قابلیت پردازش دارد.

```
var client = new RestClient("https://api.example.com");
var request = new RestRequest("/data", Method.Post);
request.AddHeader("Authorization", "Bearer token_goes_here");
request.AddJsonBody(new { name = "John", age = 30 });
RestResponse response = await client.ExecuteAsync(request);
Console.WriteLine(response.IsSuccessful);
```

مدیریت پاسخ و ارورها در RestSharp

پاسخ‌ها در RestSharp در قالب RestResponse بازخورد داده می‌شوند .این کلاس شامل اطلاعاتی همچون کد وضعیت HTTP، محتوا (Content)، خطاهای احتمالی و زمان پاسخ‌گویی است.

1. ویژگی IsSuccessful برای صحت ارسال درخواست.
2. مدیریت استثناءها و خطاها.
3. بررسی کد وضعیت .HTTP

```
if (response.IsSuccessful)
{
    Console.WriteLine("Response: " + response.Content);
}
else
{
    Console.WriteLine("Error: " + response.ErrorMessage);
}
```

یکپارچگی RestSharp با Web APIدر پروژه ASP.NET Core

RestSharpمی‌تواند به عنوان یک کلاینت در پروژه ASP.NET Coreبرای کار با دیتابیس خارجی و یا سایر سرویس‌ها استفاده شود .این ابزار کار را ساده کرده و مدیریت ارتباطات شبکه را آسان می‌کند.

1. استفاده از RestSharpدر لایه Servicesپروژه.
2. بهبود خوانایی و کپسوله کردن کد در متدهای مستقل.
3. مدیریت ارتباطات خارجی به شکل ساختار یافته.

```
public async Task<string> GetDataAsync()
{
    var client = new RestClient("https://api.example.com");
    var request = new RestRequest("/fetch-data", Method.Get);
    var response = await client.ExecuteAsync(request);
    return response.IsSuccessful ? response.Content : "Error";
}
```

ساختار Controller در Web API

مسئول Controllerهر .است که درخواست‌های کلاینت را دریافت کرده و پاسخ مناسب برمی‌گرداند Web APIکلاس مرکزی در Controller
.مدیریت یک ماژول خاص است

:نکات کلیدی

1. Controllers مکانیزمی برای ثبت و مدیریت درخواست‌های HTTP.
2. Controllerبرای هر Routeاز مانند پیش‌فرض استفاده می‌شود؛ /api/{controller}.

:یادداشت‌های ارائه دهنده

.باید خوانا و قابل مدیریت باشد Controllerتوجه کنید و اضافه کنید که Controllerبه ساختار ساده

```
public class ProductController : ControllerBase {
   [HttpGet("products")]
   public IActionResult GetAllProducts() {
      return Ok(new List<string> { "Product1", "Product2" });
   }
}
```

مقدمه به MVC

MVC (مدل-نما-کنترل‌گر)یک الگوی معماری است که برای جداسازی سه بخش اصلی برنامه استفاده می‌شود:Model ، View، و Controller.
این الگو به مدیریتی‌تر و ساختاری‌شدن کد کمک می‌کند.

1. جداسازی منطق برنامه (Controller)و داده‌ها (Model)از رابط کاربری .(View)
2. کاهش پیچیدگی و بهبود تست‌پذیری.

ارتباط بین اجزای MVC

MVC به ارتباط موثر بین Model، Viewو Controllerکمک می‌کند .داده‌ها را نمایش می‌دهد، Viewداده‌ها را نگهداری می‌کند و Controllerبه عنوان پلی بین Modelو Viewعمل می‌کند.

1. View برای نمایش داده است ولی مستقیماً با پایگاه داده کار نمی‌کند.
2. Model داده خام را نگهداری می‌کند و به Viewارسال می‌کند.
3. Controller درخواست‌ها را مدیریت می‌کند و داده‌ها را هماهنگ می‌کند.

```
public IActionResult Index() {
   var student = new Student { Name = "Ali", Age = 20 };
   return View(student);
}
```

Minimal APIچیست؟

.NET Minimal API یک روش ساده‌تر برای ساخت Web APIاست که از حداقل کدهای لازم برای ارائه APIها استفاده می‌کند .این مدل بر خلاف مدل‌های پیچیده‌تر MVCنیازمند تنظیمات ابتدایی کمتر است.

سادگی، سبک بودن، مناسب برای پروژه‌های کوچک و نمونه‌سازی

شروع کار با Minimal API

برای ایجاد یک Minimal APIابتدا باید یک پروژه خالی .NET.ایجاد شود .سپس می‌توانید با استفاده از کتابخانه‌های موجود و سینتکس ساده به تعریف APIبپردازید.

شروع با پروژه خالی، ساده‌سازی فرآیند ایجاد یک API

```
dotnet new web -n MyMinimalApi
```

ایجاد اولین Endpoint

یک Endpoint در Minimal API یک متد است که یک درخواست HTTP را مدیریت می‌کند و یک پاسخ تولید می‌نماید. Endpoint ها در فایل Program.cs تعریف می‌شوند.

فراخوانی app.MapGet، مدیریت وضعیت‌های HTTP

```
app.MapGet("/hello", () => "Hello World!");
```

افزودن Middleware

Middlewareبه شما اجازه می‌دهد که عملیات سفارشی بین دریافت یک درخواست و ارسال پاسخ انجام دهید .این یکی از قابلیت‌های قدرتمند ASP.NET Coreاست.

تعریف Middlewareساده، توسعه قابلیت‌ها در Pipelineدرخواست

```
app.Use(async (context, next) => { await next(); });
```

اضافه کردن Dependency Injection

Dependency Injection (DI)یک قابلیت اصلی در ASP.NET Coreاست که به سازماندهی بهتر کد و کاهش وابستگی‌های مستقیم کمک می‌کند.

مزایای DIدر طراحی معماری تمیز (Clean Architecture)

```
builder.Services.AddSingleton<IMyService, MyService>();
```


دانشکار
THANK YOU
FOR YOUR ATTENTION!