Razvoj Web Aplikacija

Predavanje 4

Danas

- Ubacivanje zavisnosti (engl. Dependency Injection)
- Međuprogrami (engl. Middleware)
- Konfiguracija

Dependency Injection

Arhitektura temeljena na uslugama

- Arhitektura koja se temelji na usluzi kao temeljnoj građevnoj jedinici
  - Rješavanje usluga može se postići korištenjem tehnike dependency injection 
- Usluga = enkapsulacija funkcionalnosti
- Primjer: "SMS message sender"
- Modularne i samostalne, a mogu ovisiti o drugim uslugama
  - npr. kripto-provider – usluga računa sama
  - Pošiljatelj poruka – usluga ovisi o "message gateway" servisu
- Reusable (višekratno iskoristiva usluga)
- Pruža dobro poznatu funkcionalnost, npr. putem sučelja (interface) - *IMessageSender*
- Normalno je opisati ju pomoću **sučelja** (opis funkcionalnosti) i **klase** (definiranje funkcionalnosti) - *IMessageSender* i *MessageSender*

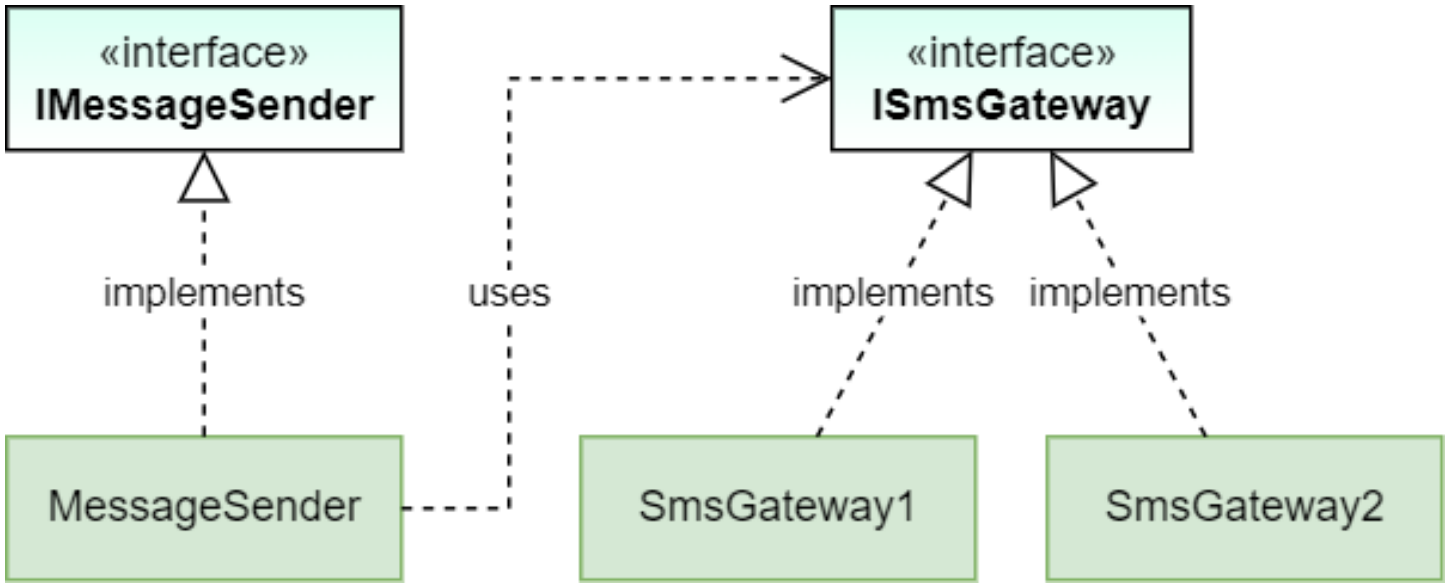
Zavisnosti

- **Inverzija kontrole** (Inversion of Control, IoC)
  - Nešto kontrolira tok programa umjesto nas
  - Primjer koji nije IoC: biblioteka (library), mi zovemo metode biblioteke
  - Primjer koji jest IoC: programski okvir (framework), on zove metode koje smo mi implementirali
  - Druga svojstva: skrivanje funkcionalnosti (npr. programiranje socketa), instanciranje objekata umjesto nas itd.
- **Princip inverzije zavisnosti** (Dependency Inversion Principle, DIP)
  - Jednostavna implementacija zavisnosti: klasa A treba od klase B funkcionalnost, pa instancira klasu B i poziva njene metode
  - Klasa A je klijent, klasa B je zavisnost
  - Ako se promijeni B, promjenu mora implementirati i A pa kažemo da A zavisi od B
  - Ideja: fiksirati sučelje klase B (*class B : IB*)
  - Sada A diktira kako izgleda IB, pa implementacija B zavisi od toga što A zahjteva i zato to nazivamo inverzija zavisnosti

Zavisnosti (nastavak)

- **Ubacivanje zavisnosti** (Dependency Injection, DI)
  - Tehnika gdje se stvaranje zavisnosti B odigrava izvan klase A koja tu zavisnost treba - **IoC**
  - Klasa A diktira sučelje IB, a vanjska komponenta za IB nalazi implementaciju B (jer *class B : IB*) - **DIP**
  - Da bi to radilo, mora postojati komponenta koja je svjesna mapiranja između IB i B - **DI kontejner**
  - Mogućnosti ubacivanja: preko konstruktora, preko metode, preko svojstva (mora implementirati *set*)
  - Mi koristimo ubacivanje **preko konstruktora**
  - Na taj način su konfiguracija, konstrukcija i korištenje odvojeni koncepti
- **Servisni lokator** (Service Locator, SL)
  - Slično kao DI, ali ubacivanje zavisnosti se radi "ručno"
  - Obrnuto od **IoC**, ali je i dalje implementacija **DIP**
  - Svaka klasa koja to koristi ovisi o posebnoj Service Locator komponenti koja zna mapiranje između sučelja i implementacije

Apstrakcija / zavisnost

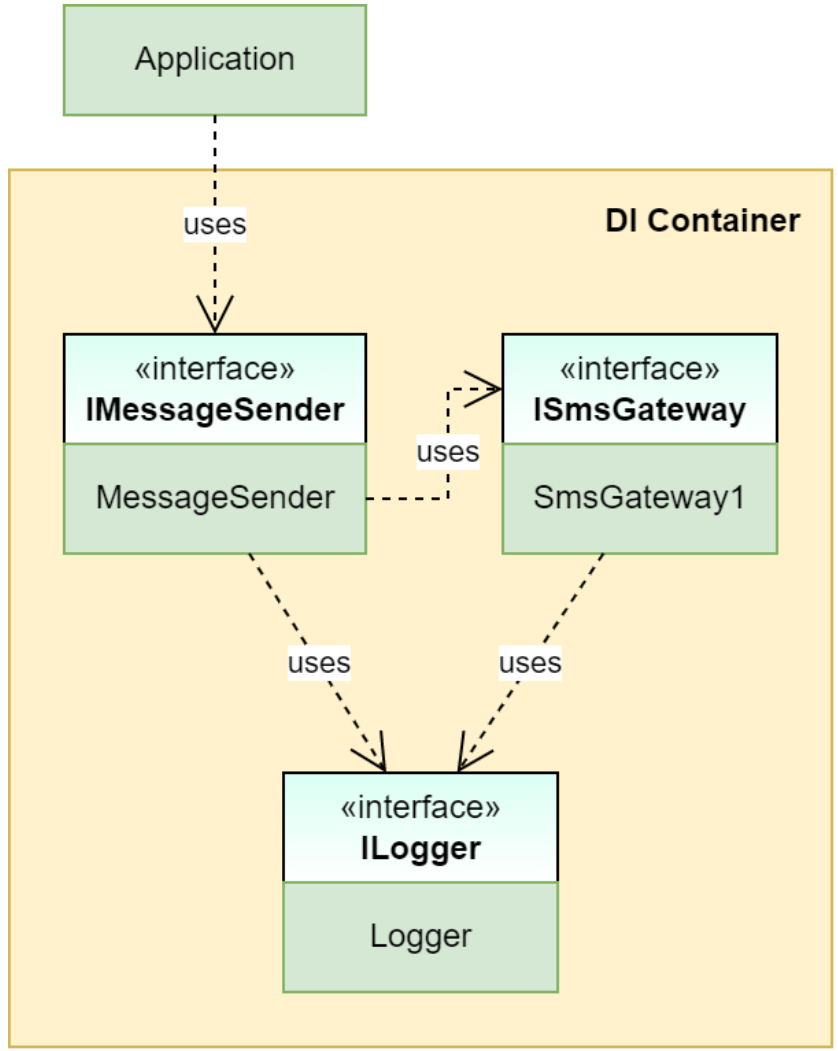


Dependency Injection vs. Service Location

- Kada aplikacija treba određenu uslugu, ona se obraća njenom **sučelju** (apstrakcija) za dohvaćanje objekta **klase** (implementacija)
  - "Molim te, daj mi implementaciju IMessageSendera" ➔ "Evo, uzmi ovaj MessageSender"
- Tu funkcionalnost možemo koristiti putem **Service Locatora** ili **Dependency Injectiona**
  - Service Locator (lokator servisa) je posebna komponenta koju možemo koristiti za dobivanje usluge MessageSender baš tada kada nam zatreba
  - Dependency Injection (ubacivanje zavisnosti) je obrazac koji kaže da će MessageSender usluga biti pružena njenom korisniku kada se korisnik instancira
  - Koji je pristup bolji?
  - **Service Locator** može odgoditi dobivanje usluge do trenutka kada je ona stvarno potrebna, ali u tom slučaju moramo ručno kontrolirati kada ona više nije potrebna i kada se radi dispose
  - **Dependency Injection** ima dobro definiranu/poznatu kontrolu životnog vijeka, ali učitava usluge odmah u trenutku kada se stvara korisnika usluge

Dependency Injection (IoC)

- Postoji konsenzus da Dependency Injection (DI) ima mnoge prednosti u odnosu na Service Location (SL)
- Komponenta koja brine o dependency injection naziva se **DI Container**
- Web API ima **DI Container** implementiran "ispod haube"
- Također se pojednostavljeno naziva inverzija kontrole, IoC ili **IoC container** – ne kontroliramo instancirajuću uslugu, nego kontejner to radi za nas

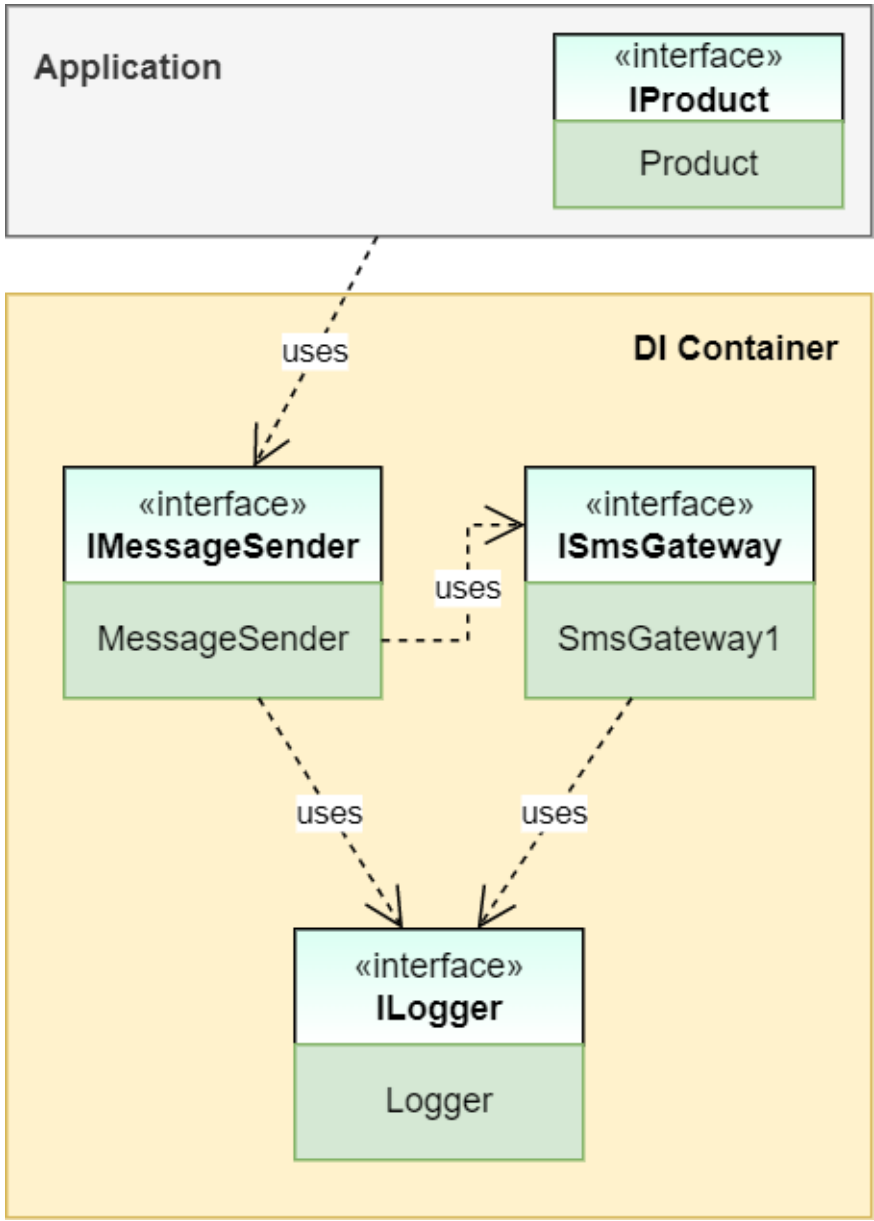


DI registracija kontejnera

- U stvarnom životu bilo koje sučelje za implementaciju ne omogućuje rješavanje odgovarajuće implementacije objekta klase
- Samo s registriranim parovi poput *(IMessageSender, MessageSender)* će DI spremnik rukovati
- Par *(IProduct, Product)* ovdje nije registriran od DI/IoC kontejnera i s njim se ne postupa kao da je registriran

Primjer registracije usluge u DI spremniku:

```
builder.Services.AddScoped<IMessageSender, MessageSender>();
```



Životni vijek zavisnosti

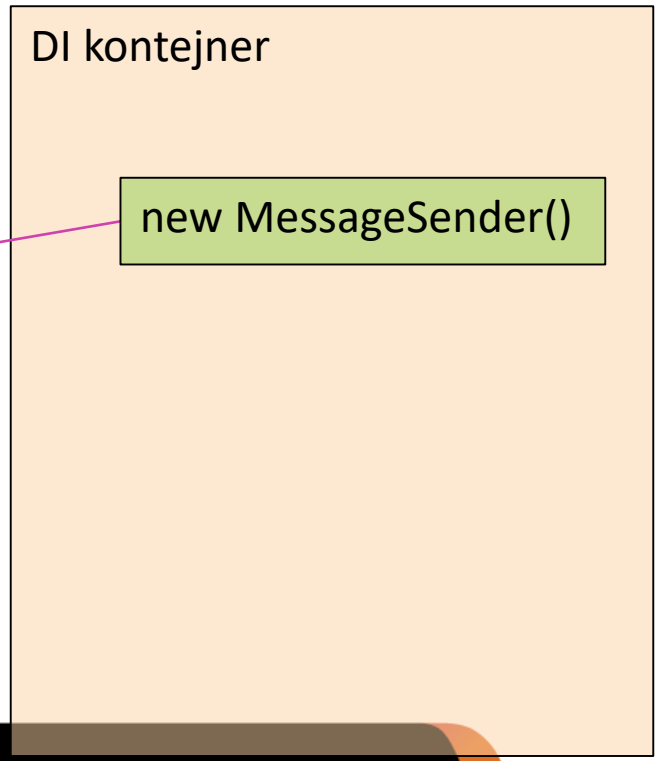
- Eng. dependency lifetime
- Kada se radi odlaganje (dispose) usluge?
- Usluge se mogu registrirati za jedan od sljedećih vijekova trajanja:
  - **Transient** – **svaki put kada se usluga zatraži**, kontejner vraća novu uslugu; odlaže se (dispose) kada **zahtjev završava**
  - **Scoped** – **svaki put kada stigne HTTP zahtjev**, kontejner stvara novu uslugu; odlaže se (dispose) kada **zahtjev završava**
  - **Singleton** – **prvi put kada se usluga zatraži** iz kontejnera, kontejner stvara novu uslugu; odlaže se (dispose) kada **se aplikacija gasi**
  - Metode registracije:

```
builder.Services.AddTransient<ILogger, Logger>();
builder.Services.AddScoped<IMessageSender, MessageSender>();
builder.Services.AddSingleton<ISmsGateway, SmsGateway>();
```

Rješavanje zavisnosti

- Eng. dependency resolution
- Kada DI/IoC kontejner radi instancu (i ubacivanje) potrebnih komponenata (zavisnosti)?
  - ➔ prije instanciranja komponente koja ih treba
  - ➔ zatim se prosljeđuju konstruktoru te komponente!

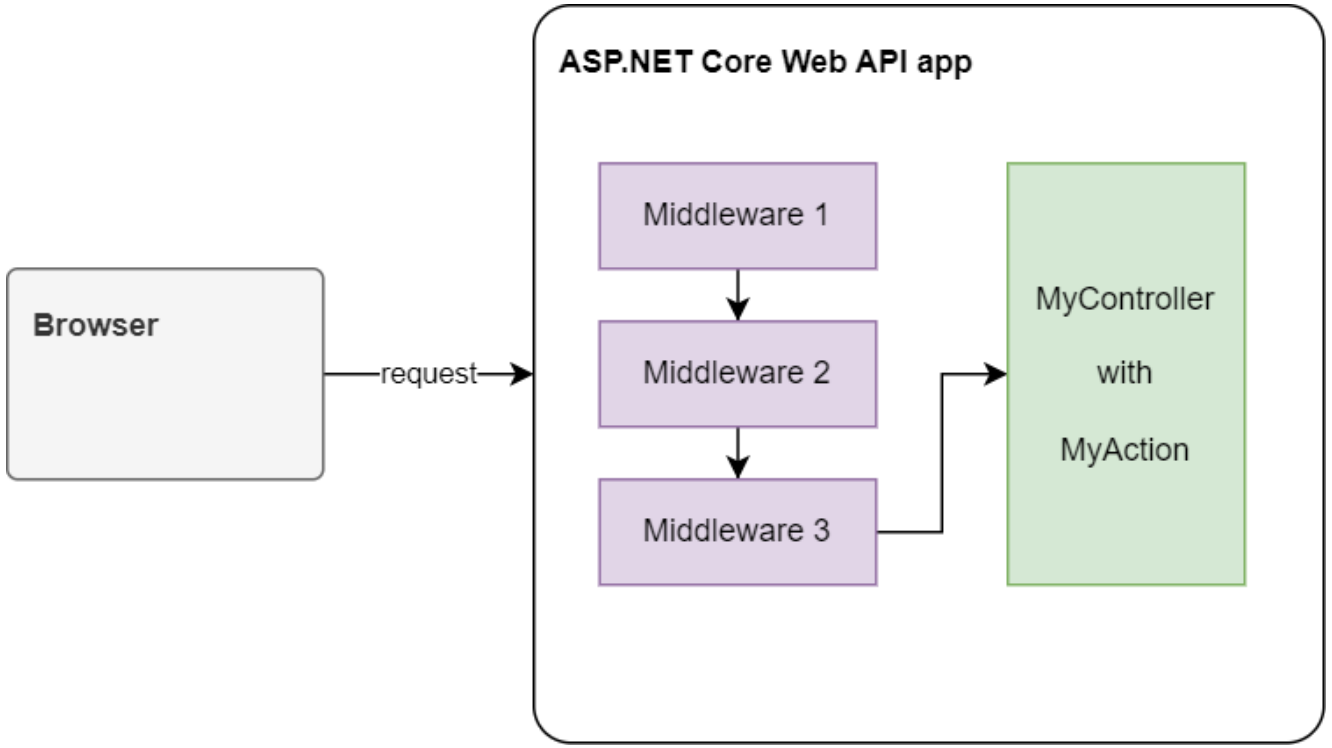
```
[Route("api/[controller]")]
[ApiController]
public class SummaryController : ControllerBase
{
  IMessageSender _sender;

  public SummaryController(IMessageSender sender)
  {
    _sender = sender;
  }

  [HttpPost]
  public Summary AddSummary(Summary summary)
  {
    _sender.Send("Summary added");
    …
```



Middleware

Middleware osnove

- Kada zahtjev stigne, prvo ga obrađuje middleware (a ne usmjeravanje? zašto?)
- Zapravo, komponenta za usmjeravanja također je middleware! 😉



Korištenje middleware-a

- Ima mnogo ugrađenog middleware-a koji je na dohvat ruke u ASP.Net Core Web API
- Iz Program.cs:

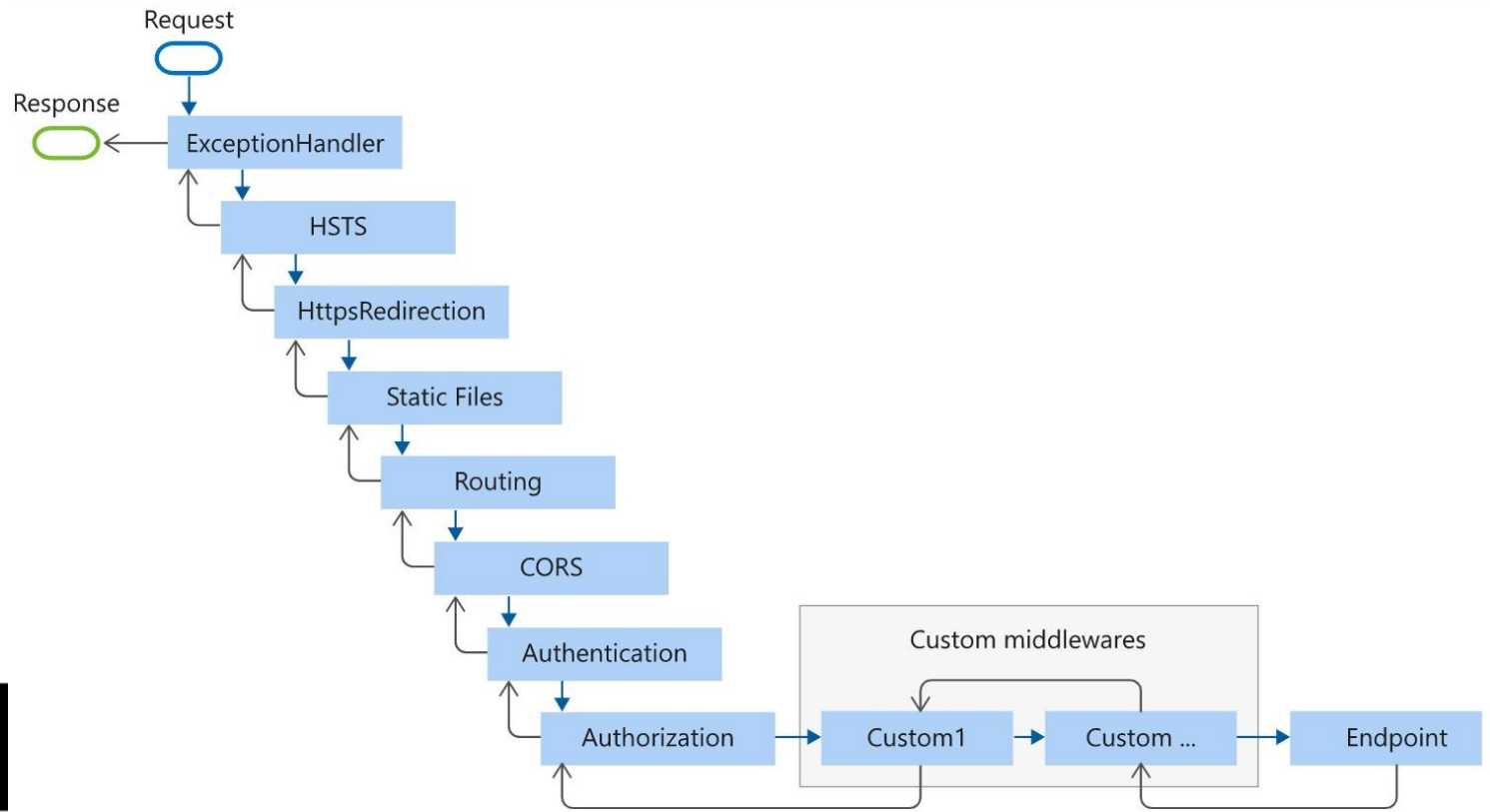
```
app.UseAuthorization(); // Autorizacijski middleware
app.MapControllers(); // Middleware za kontrolere
app.Run(); // Run() završava middleware setup
```

- Metoda UseAuthorization() koristi općenitiju metodu **Use()** a isto vrijedi i za MapControllers() i **Map()**

```
app.Use(async (context, next) => {
    // Logic of a custom middleware
    await next.Invoke();
});
```

Ugrađeni middleware

- Kao što je već spomenuto, usmjeravanje je također middleware
- Možete pozicionirati middleware za usmjeravanje, kao i bilo koji drugi middleware u middleware poretku pozivanjem UseRouting(), inače će se prema zadanim postavkama koristiti na odgovarajućem položaju



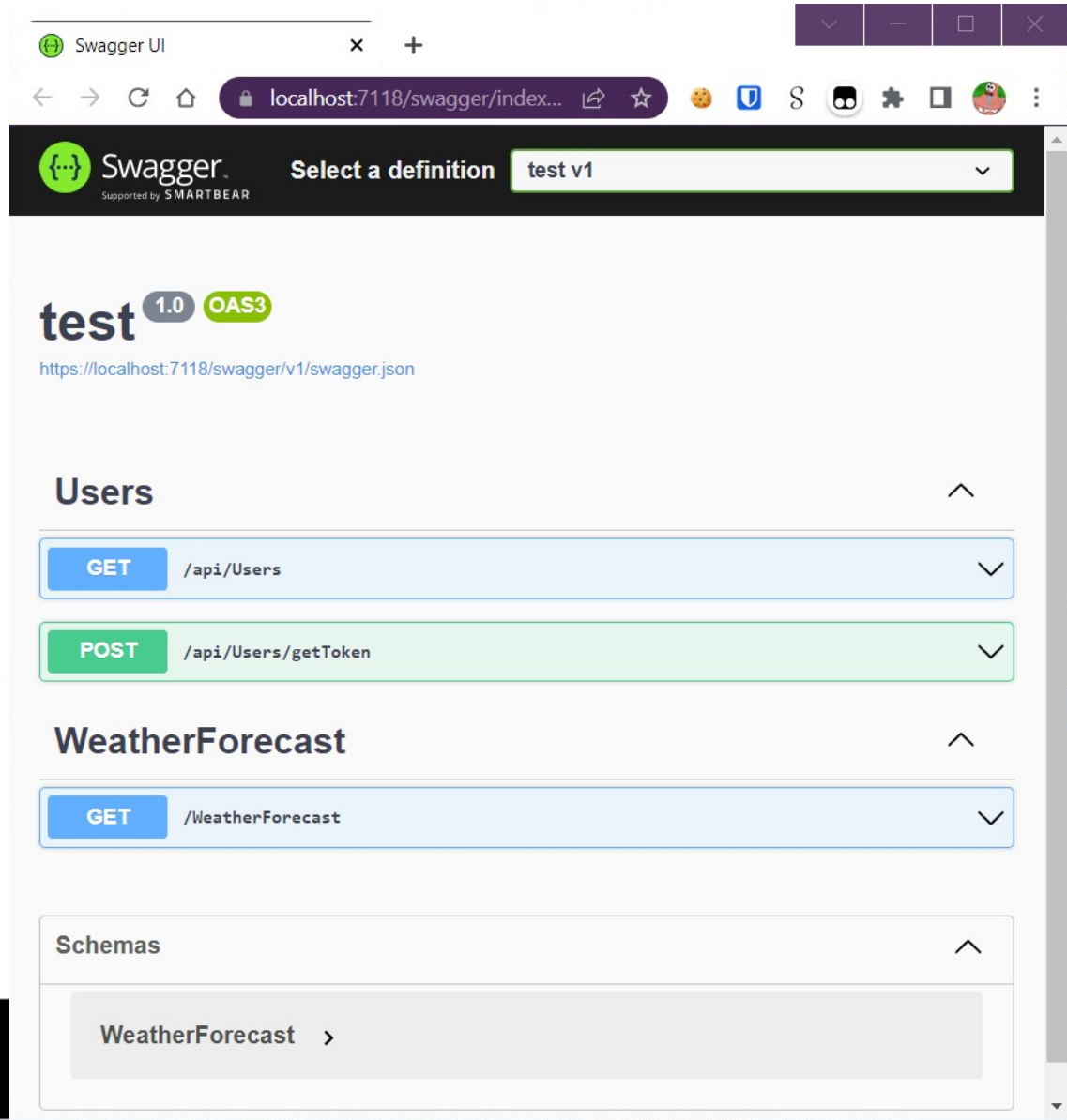
Ugrađeni middleware (nastavak)

- Stranica za rukovanje iznimkama: **app.UseExceptionHandler()**
  - Koristi se po defaultu kada je okruženje postavljeno na "Development"
- HTTPS preusmjeravanje: **app.UseHttpsRedirection()**
  - Za to treba konfigurirati HTTPS
- CORS (cross-origin resource sharing): **app.UseCors()**
  - Za to je potrebno konfigurirati pravila CORS-a
- Autentifikacija: **app.UseAuthentication()**
- Autorizacija: **app.UseAuthentication()**
- OpenAPI/Swagger: **app.UseSwagger(); app.UseSwaggerUI();**
- Rate-limiting: **app.UseRateLimiter()**
- ...

OpenAPI/Swagger middleware

- Ovaj middleware ima kompletni UI
- Omogućuje slanje POST i ostalih zahtjeva ☺
- Također može omogućiti izvoz kolekcija koje Postman razumije
- Vrlo jednostavno konfiguriranje

```
builder.Services.AddSwaggerGen();
var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```



Konfiguracija

Izvori konfiguracije

- ASP.Net Core Web API čita konfiguraciju iz **appsettings.json** datoteke, ali i od jednog ili više konfiguracijskih providera (davatelja usluga):
  - Environment varijable
  - Azure Key Vault
  - Azure App Configuration
  - Argumenti naredbenog retka (command line)
  - Prilagođeni provideri, instalirani ili stvoreni
  - Datoteke iz mape
  - .NET objekti u memoriji
- Svaka konfiguracija je par ključ-vrijednost

Primjer konfiguracije (appsettings.json)

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  },
  "MyKey": "My appsettings.json Value",
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Konfiguracija u appsettings.json

- Obično je konfiguracija zapisana u **appsettings.json** datoteci

- Datoteke **appsettings.Development.json** i **appsettings.Production.json** koriste se za nadjačavanje konfiguracijskih vrijednosti za određeno okruženje

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Trace"
    }
  }
}
```

- Okolina se određuje iz **ASPNETCORE_ENVIRONMENT** env varijable ("Development", "Staging", "Production"), i po defaultu se prosljeđuje u aplikaciju iz **launchSettings.json**

```
"environmentVariables": {
  "ASPNETCORE_ENVIRONMENT": "Development"
}
```

Pristup konfiguraciji

- Da bismo pristupili konfiguraciji, DI kontejner ubacuje implementaciju IConfiguration

```
[Route("api/[controller]")]
[ApiController]
public class SummaryController : ControllerBase
{
  private readonly IMessageSender _sender;
  private readonly IConfiguration _configuration;

  public SummaryController(IMessageSender sender, IConfiguration configuration)
  {
    _sender = sender;
    _configuration = configuration;
  }

  public List<Summary> GetSummaries()
  {
    var defaultLogLevel = _configuration["Logging:LogLevel:Default"];

    ...
  }
}
```

Konfiguracija i options pattern

- Options pattern se koristi za binding složene konfiguracije s klasom

PositionOptions.cs

```
public class PositionOptions
{
  public const string Position
    = "Position";

  public string Title { get; set; } = "";
  public string Name { get; set; } = "";
}
```

appsettings.js

```
{
  "Position": {
    "Title": "Editor",
    "Name": "Joe Smith"
  }
}
```

Konfiguracija i options pattern (nastavak)

- Umjesto da se bavimo cijelom konfiguracijom, bavimo se samo dijelom

```
[Route("api/[controller]")]
[ApiController]
public class SummaryController : ControllerBase
{
  private readonly IMessageSender _sender;
  private readonly PositionOptions _options;

  public SummaryController(IMessageSender sender, IOptions<PositionOptions> options)
  {
    _sender = sender;
    _options = options.Value;
  }

  public List<Summary> GetSummaries()
  {
    var title = _options.Title;
    var name = _options.Name;
    ...
  }
}
```

Konfiguracija i options pattern (nastavak)

- Da bismo to koristili, prvo moramo konfigurirati
- Konfiguracija se vrši u kodu (Program.cs)

```
builder.Services.Configure<PositionOptions>(
    builder.Configuration.GetSection(PositionOptions.Position));
```

Hvala na pažnji!