Bakgrunden till den här bloggserien är att vi ser ett ökat intresse för konsulter med erfarenhet av ASP.NET Core MVC och .NET Core – och vad passar väl då bättre än att Jimi som är en av våra konsulter inom systemutveckling och C# .NET visar hur man bygger en enkel molnbaserad applikation med dessa tekniker?

[Läs gärna vårt första inlägg om .NET core](http://www.agero.se/blogg/asp-dotnet-core-del1-vad-och-varf%C3%B6r)

**Välkommen! – till min introduktion i att bygga en applikation med .NET Core och ASP.NET Core.**

Den här guiden är till för dig som är utvecklare och har lite erfarenhet av C# och .NET men är ny när det gäller .NET Core. I [första delen](http://www.agero.se/blogg/asp-dotnet-core-del1-vad-och-varf%C3%B6r) av den här bloggserien pratade vi lite kort om vad .NET Core är och varför man ska använda det, i den här och följande delar går jag igenom utvecklingen av ett projekt från start till mål och all kod kommer att finnas tillgänglig för nedladdning allt eftersom delarna blir färdiga.

Applikationen vi bygger visar konsultprofiler i från ett konsultregister.

Innehåll i bloggserien:

* [del 1](http://www.agero.se/blogg/asp-dotnet-core-del1-vad-och-varf%C3%B6r) - en introduktion till .NET Core
* del 2 (här och nu) - vi skapar ett Web Api med autentisering och enhetstester/integrationstester.
* del 3 - en administrationssida i ASP.NET Core MVC (tag helpers, routing, autentisering)
* del 4 - en klient i form av en mobilapp i Ionic.
* del 5 - vi refaktoriserar appen, använder Azure DocumentDb för att lagra konsultprofilerna och flyttar konfigurationen av IdentityServer till databasen.
* del 6 - vi driftsätter appen i Azure
* därefter kommer det att implementeras någon typ av administration för att hantera användare och roller, planen var att jag skulle använda mig av [IdentityManager](https://github.com/identitymanager) men det finns ingen färdig implementation för .NET Core Identity ännu så vi får se hur jag löser det, inget är bestämt än och har du tips får du gärna kontakta mig.
* i fortsättningen kommer vi när tid finns att göra en app i React och React Native för administration av innehåll.

*Tips! För att inte missa något är det bäst att du prenumererar på bloggen genom att skriva in din e-postadress här till höger.*

**Min bakgrund inom .NET Core.**

För drygt ett år sedan började jag kolla på Core, jag läste bloggar och kollade på videor, tog en [videokurs på Pluralsight](https://app.pluralsight.com/library/courses/play-by-play-asp-net-core-1-0-on-any-os-john-papa-shayne-boyer/description) och labbade, men bestämde mig för att avvakta eftersom ramverket var  i betastadiet och fortfarande förändrades mycket. Nu är .NET Core lite stabilare så det är alltså dags att ta upp kärnan igen…

*.NET Core 1.0 släpptes i juni 2016 och 1.1 släpptes i november och senare i vår släpps version 2.0.*

Något jag personligen gillar med Core är inbyggd "dependency injection" och hur man konfigurerar projekten, med json-filer för dependencies, tester och byggen. Konfigurationen är ganska lik i hur man konfigurerar javascriptprojekt som t ex Angular och React, så för dig som jobbat i frontendstacken är det här troligen inget nytt.

Ett par saker till som jag gillar är att man kan köra ASP .NET Core som en ”stand alone”-app utan att vara beroende av IIS, väldigt trevligt! Och en sån liten sak som att projektets struktur mappar mot det man ser i Visual Studios Solution Explorer – hur skönt är inte det?

**.NET Core Web Api Projektanteckningar/Steps**

Dokumentationen som följer här är från README-filen i [projektet som finns här på github](https://github.com/JimiSweden/dotnetcoreBloggserie) men är kraftigt uppdaterad och inkluderar kod för att du ska kunna följa med från start till mål utan att behöva titta i projektet.

Om du vill kan du stega igenom incheckningarna i projektet då jag har försökt att göra så många som möjligt med detaljerad beskrivning om vad som är gjort. Vissa saker som namngivning etc. i den incheckade koden skiljer sig från koden här på sidan, pga. justeringar i samband med genomgången av anteckningarna här.

**Projektkrav**

Applikationen ska visa en lista av konsultprofiler

* En profil består av Förnamn, Efternamn och en Beskrivning

Som en anonym användare kan jag

* Registrera mig som användare
* Se förnamn på profilerna

Som en registrerad användare kan jag

* Se all information i konsultprofilerna

**Teknikval**

Eftersom projektet har som syfte att driva den här bloggserie om [.NET Core](https://docs.microsoft.com/sv-se/dotnet/) och [ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/) faller det sig naturligt att använda följande tekniker, samtliga servade i det stora fluffiga molnet, och även om andra molntjänster skulle kunna användas kommer vi slutligen drifta allt i Azure.

* **API**: Alla moderna system exponerar data via apier, så vi använder **.NET Core Web API**
* ***Klient (senare)****: Kan komma att byggas i flera olika tekniker, men till att börja med blir det en mobilapp i hybridramverket* [***Ionic***](https://ionicframework.com/)*, som bygger på* [*Angular*](https://angularjs.org/) *och Html5.*
* ***Administration (senare)*** *av profildata: Då vi behöver blanda in* [*ASP.NET core MVC*](https://docs.microsoft.com/en-us/aspnet/core/mvc)*någonstans får det bli här.*
* **Autentisering**: med .NET Core känns det självklart att anväda [ASP.NET Core Identity](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity) för hantering av användare. Här skulle vi kunna nöja oss med bara Identity, men eftersom jag är en förespråkare för [IdentityServer](https://identityserver4.readthedocs.io/) gör vi implementationen med IdentityServer4 (IdS) ovanpå core Identity. Den stora fördelen IdS ger oss är att vi kan använda samma autentiseringslösning för olika system, något som inte är ett krav här men jag väljer att göra då jag själv inte har testat IdS4 ännu.

**Utvecklingsmiljö**

Jag använder här Visual Studio 2015 men du kan även använda Visual Studio 2017, detta är de versioner av Visual Studio som fungerar med .NET Core. Du kan självklart använda andra utvecklingsverktyg med .NET Core SDK.

Läs mer i Microsofts dokumentation om ["Windows Prerequisites"](https://docs.microsoft.com/en-us/dotnet/articles/core/windows-prerequisites)

* [Hur du sätter upp din miljö med Visual Studio 2015 hittar du här](https://docs.microsoft.com/en-us/dotnet/articles/core/windows-prerequisites#visual-studio-2015)
* [Hur du sätter upp din miljö med Visual Studio 2017 hittar du här](https://docs.microsoft.com/en-us/dotnet/articles/core/windows-prerequisites#visual-studio-2017-rc)

Om du vill utveckla i macOS läs ["Getting started with .NET Core on macOS, using Visual Studio Code"](https://docs.microsoft.com/en-us/dotnet/articles/core/tutorials/using-on-macos).

**Och nu – dags för implementation!**

**Skapa ett nytt projekt för APIet**

* Lägg till nytt projekt, ASP.NET Core Web Application (.NET Core) med namn "Profil.Api" och Empty template.
* Starta appen och verifiera att du får upp en browser med meddelandet "Hello World!"

**Program.cs**

I Program.cs konfigurerar man webbservern (t ex om vi vill köra IIS Express mm)  
Vi låter inställningarna vara som de är, vill du lära dig mer om hur du konfigurerar hosting finns det en bra beskrivning i dotnet-dokumentationen under [hosting](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/hosting) och mer info om servertyper under [servers](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/)

public static void Main(string[] args)

{

var host = new WebHostBuilder()

.UseKestrel()

.UseContentRoot(Directory.GetCurrentDirectory())

.UseIISIntegration()

.UseStartup<Startup>()

.Build();

host.Run();

}

*lägg märke till att "UseStartup" kallar på "Startup.cs"*

**Startup.cs**

I Startup.cs konfigurerar man "service dependencies" och "HTTP Request pipeline".

**Service Dependencies - Startup.ConfigureServices(...)**

* Här konfigurerar vi de tjänster vi är beroende av, t ex databas, repositories, autentisering etc.

**Http Request pipeline - Startup.Configure(...)**

* En Http Request pipeline består av ett antal "middlewares", vilka kopplas på IApplicationBuilder (app) – vanliga middelwares hämtar du från nuget, och du kan så klart skriva dina egna. Mer information om Middleware hittar du  i [dokumentationen](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware)
* Ordningen för middlewares är viktig eftersom en middleware tar emot data från föregående middleware, gör sin grej och skickar vidare data till nästa middleware i kedjan. Vissa middlewares lyssnar på returdata från middlewares senare i kedjan (se exempel nedan).

Exempel:

public void Configure(IApplicationBuilder app)

{

app.UseExceptionHandler(); //mw1

app.UseIdentity(); //mw2

app.UseMvc(); //mw3

}

UseExceptionHandler(mw1) skickar vidare request till UseIdentity(mw2) som skickar vidare request till UseMvc(mw3) som visar data från våra controllers - om en controller kräver autentisering kommer den nu att använda Identity som autentiseringsmekanism - om ett fel kastas, i Identity eller Mvc, tar ExceptionHandler hand om detta.

* Vissa middlewares är så kallade "terminal" vilket betyder att de inte skickar data vidare, de är sist i kedjan; några exempel är UseWelcomePage() och Run(), middlewares konfigurerade senare i kedjan kommer alltså inte att köras.

För att kunna köra vårt api som bygger på MVC måste vi lägga till ett nuget-paket för detta och uppdatera konfigurationen med UseMvc.

* Lägg till paketet "Microsoft.AspNetCore.Mvc", antingen via nuget package manager eller direkt i "project.json", vilken därefter ska se ut så här.

"dependencies": {

"Microsoft.NETCore.App": {

***"version": "1.1.0"****,*

"type": "platform"

},

"Microsoft.AspNetCore.Diagnostics": "1.1.0",

**"Microsoft.AspNetCore.Mvc": "1.1.0",**

"Microsoft.AspNetCore.Server.IISIntegration": "1.1.0",

"Microsoft.AspNetCore.Server.Kestrel": "1.1.0",

"Microsoft.Extensions.Logging.Console": "1.1.0",

"IdentityServer4.AccessTokenValidation": "1.0.1"

},

"tools": {

"Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.1.0-preview4-final"

},

"frameworks": {

***"netcoreapp1.1"***: {

"imports": [

"dotnet5.6",

"portable-net45+win8"

]

}

},

Som du kan se har jag uppdaterat .NET Core till version 1.1.0 vilken är den senaste just nu, det är inget krav att du uppdaterar.

Har du inte redan installerat 1.1 följer du dessa instruktioner för att uppdatera både core och nuget.

* uppdatera nuget till senaste beta <https://docs.nuget.org/ndocs/guides/install-nuget>
* uppdatera till .NET Core 1.1 <https://blogs.msdn.microsoft.com/webdev/2016/11/16/announcing-asp-net-core-1-1/>

Nu kan vi lägga till Mvc i konfigurationen

* Uppdatera Startup till att se ut så här

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

**services.AddMvc();**

}

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)

{

loggerFactory.AddConsole();

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

**app.UseMvc();** }   
}

**Lägg till ett repository och en modell för konsultprofilen**

*Vanligtvis gör jag de här stegen i omvänd ordning, men det blir tydligare att förklara så här.*

* Lägg till en ny katalog med namnet "Models" i projektet
* Skapa en datamodell för konsultprofilen, Models/ConsultantProfile.cs

public class ConsultantProfile

{

public string FirstName { get; set; }

public string LastName { get; set; }

public string Description { get; set; }

}

* Skapa ett repository för att hämta konsultprofiler, Models/ConsultantProfileRepository.cs. Vi börjar med en hårdkodad implementation för att inte blanda in databaser än.

public interface IConsultantProfileRepository  
 {  
 IEnumerable<ConsultantProfile> GetAll();  
 }  
  
 //en första implementation utan att blanda in databaser  
 public class InMemoryConsultantProfileRepository : IConsultantProfileRepository  
 {  
 private List<ConsultantProfile> consultantProfiles;  
  
 public InMemoryConsultantProfileRepository()  
 {  
 consultantProfiles = new List<ConsultantProfile>  
 {  
 new ConsultantProfile { FirstName = "Jimi", LastName = "Friis", Description = "Systemutvecklare Fullstack - C# .net, javascript..." },  
 new ConsultantProfile { FirstName = "Pontus", LastName = "Nagy", Description = "Systemutvecklare Frontend - javascript, React, Html..." },  
 new ConsultantProfile { FirstName = "Mikael", LastName = "Sundström", Description = "Sysadmin - Windows, Mac..." },  
 };  
 }  
  
 public IEnumerable<ConsultantProfile> GetAll()  
 {  
 return consultantProfiles;  
 }  
 }

 Med repositoryt på plats kan vi lägga till en controller för att exponera profilerna i vårt api.

* Lägg till en ny katalog med namnet "Controllers" i projektet där du lägger till ConsultantProfileController.cs

[Route("api/[controller]")]  
 public class ConsultantProfileController : Controller   
 {  
     private readonly IConsultantProfileRepository consultantProfileRepository;  
  
     public ConsultantProfileController(IConsultantProfileRepository consultantProfileRepository)  
     {  
         this.consultantProfileRepository = consultantProfileRepository;  
     }  
  
     [HttpGet]  
     public IEnumerable<ConsultantProfile> Get()  
     {  
         return consultantProfileRepository.GetAll();  
     }  
 }

 För att controllern ska hittas behövs routing vilket man kan sätta direkt på controllern så som vi visar i koden ovan, alternativt i Startup.Configure (eller både och; controllerns konfiguration väger då tyngst).

Här är ett exempel på en konfiguration av default route – om "controller name" inte hittas i sökvägen skickas anropet till HomeController med Index som vald action (metod).

private void ConfigureRoutes(IRouteBuilder routeBuilder)

{

routeBuilder.MapRoute("Default","{controller=Home}/{action=Index}/{id?}");

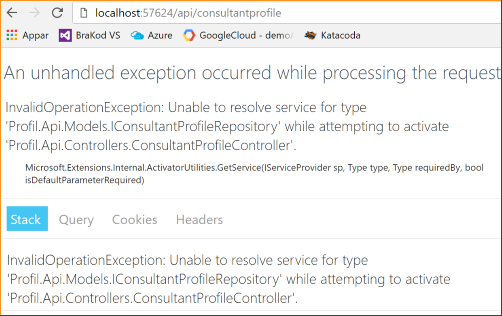
}

// används i Configure enligt, app.UseMvc(ConfigureRoutes)  
  
// Tips! Om du ska ha en default route enligt "{controller=Home}/{action=Index}/{id?}"  
// kan du göra det enkelt för dig genom att använda "app.UseMvcWithDefaultRoute()"  
// istället för "app.UseMvc()"

Med controller, repository och vymodell på plats kan vi testa apiet.

* Starta applikationen i debug och gå till  sökvägen <http://localhost:57624/api/consultantprofile>   
  (portnummer får du så klart anpassa efter vad du har, eller sätta till 57624 i "project properties > debug")

Du ska nu få ett felmeddelande som ser ut som på bilden nedan – och det beror på att vi inte har konfigurerat ConsultantProfileRepository som en service, vilken vi försöker använda i  ConsultantProfileController.



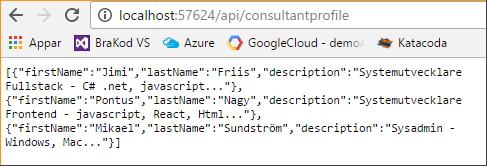
Anledningen till att vi får ett så fint felmeddelande är "app.UseDeveloperExceptionPage()"

* Lägg till ConsultantProfileRepository i Startup.ConfigureServices

...  
public void ConfigureServices(IServiceCollection services)  
{  
 services.AddMvc();  
 **services.AddScoped<IConsultantProfileRepository, InMemoryConsultantProfileRepository>();** //Scoped skapar en instans per http request  
}  
...

* Starta projektet och verifiera att du får tillbaks data när du anropar <http://localhost:57624/api/consultantprofile> innan vi går vidare med autentiseringen.

Resultatet bör se ut så här



Lysande! – Vi har ett fungerande api som kan leverera konsultprofiler. Nästa steg är att lägga till autentisering, vilket vi gör med .NET Core Identity och [Identity Server 4](https://github.com/IdentityServer/IdentityServer4)

**Skapa ett nytt projekt för IdentityServer och .NET Core Identity**

Här följer jag exemplet från [Identity Server Quickstarts > Using ASP.NET Core Identity](https://identityserver4.readthedocs.io/en/release/quickstarts/6_aspnet_identity.html) och eftersom exemplet är så bra tänker jag inte duplicera det utan bara komplettera med mina noteringar. Det finns även ett [exempelprojekt att ladda ner här](https://github.com/IdentityServer/IdentityServer4.Samples/tree/dev/Quickstarts/6_AspNetIdentity), med [deras färdiga UI](https://github.com/IdentityServer/IdentityServer4.Quickstart.UI).

* Följ nu stegen från "New Project for ASP.NET Identity" till och med "Configure IdentityServer" i "[Using ASP.NET Core Identity](https://identityserver4.readthedocs.io/en/release/quickstarts/6_aspnet_identity.html)" (läs gärna mina kommentarer till varje steg innan du börjar).

[**Add IdentityServer packages**](https://identityserver4.readthedocs.io/en/release/quickstarts/6_aspnet_identity.html#new-project-for-asp-net-identity)

* När jag uppdaterade alla nugetpaket till det senaste misslyckades nuget med restore (dotnet restore), vilket löstes genom att uppdatera nuget till senaste betaversionen <https://docs.nuget.org/ndocs/guides/install-nuget>
* Därefter följde jag instruktionerna i denna länk för att uppdatera till .NET Core 1.1 <https://blogs.msdn.microsoft.com/webdev/2016/11/16/announcing-asp-net-core-1-1/>

[**Configure IdentityServer**](https://identityserver4.readthedocs.io/en/release/quickstarts/6_aspnet_identity.html#configure-identityserver)

Konfiguration av IdentityServer görs med "InMemory" settings, dvs en statisk konfiguration i koden  
– senare kommer vi att flytta detta till en SQL-databas.

**När du är klar med stegen i "Confiure IdentityServer" fortsätter du här**

Än så länge är det fritt fram att hämta data från <http://localhost:57624/api/consultantprofile> men det är dags att begränsa åtkomsten.

Med Identity Server på plats kan vi lägga till autentisering på vårt api – stegen nedan gör vi  alltså i projektet Profil.Api

* Lägg till "Authorize" på ConsultantProfileController.Get()   
  – om du försöker hämta data nu får du tillbaks status code "401 Unauthorized"

...  
 [HttpGet]  
 [Authorize]  
 public IEnumerable<ConsultantProfile> Get()  
 {  
 return consultantProfileRepository.GetAll();  
 }  
...

Vi behöver även konfigurera vårt api till att använda IdentityServer för autentisering, dvs så att just vår IdentityServer är betrodd för att hantera inloggningarna.

* Lägg till "IdentityServer4.AccessTokenValidation" som en dependency i project.json

...  
 "Microsoft.AspNetCore.Mvc": "1.1.0",  
 **"IdentityServer4.AccessTokenValidation": "1.0.1"**  
...

* Lägg till middleware för IdentityServer i Startup.Config, före app.UseMvc

...  
 **//options måste matcha inställningarna i IdentityServer-projektet**  
 **app.UseIdentityServerAuthentication(new IdentityServerAuthenticationOptions**  
 **{**  
 **Authority = "http://localhost:5000",**  
 **AllowedScopes = { "api1" },**  
 **RequireHttpsMetadata = false**  
 **});**  
  
 app.UseMvc();  
...

[**Creating the user database**](https://identityserver4.readthedocs.io/en/release/quickstarts/6_aspnet_identity.html#creating-the-user-database)

Eftersom vi använder Identity för användarhanteringen behöver databasen initieras innan vi kan registrera och logga in användare – tänk på att vi nu är tillbaka i projektet "IdentityServerWithAspNetIdentity".

Om du vill namnge din första migrering innan du skapar databasen kan du köra kommandot dotnet ef migrations add InitialDbMigration innan dotnet ef database update

Har du som jag uppdaterat .NET Core till 1.1 så behöver du lägga till några nugetpaket (läs mer om paketen i [dokumentationen](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet)) – annars får du varningen No executable found matching command dotnet-ef när du försöker initiera databasen.

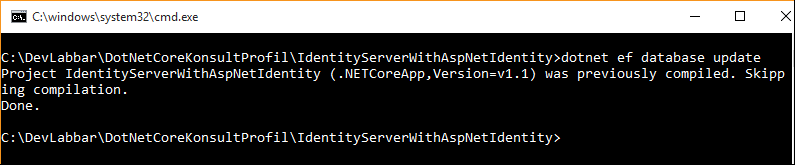
* Lägg till "Microsoft.EntityFrameworkCore.Design" som "build only" dependency i project.json

...  
 "dependencies": {  
 ...  
  **"Microsoft.EntityFrameworkCore.Design": {**  
 **"type": "build",**  
 **"version": "1.1.0"**  
 **},**  
 ...

* Lägg till "Microsoft.EntityFrameworkCore.Tools" och  "Microsoft.EntityFrameworkCore.Tools.DotNet" som tools i project.json

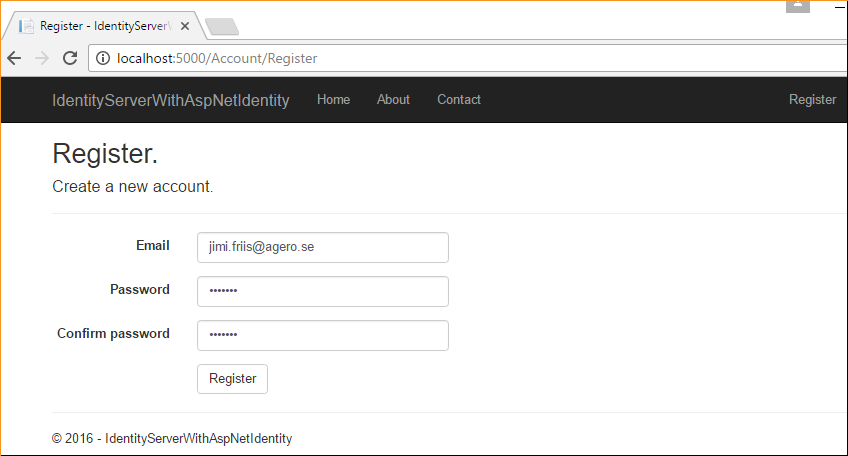
...  
"tools": {  
 ...  
 **"Microsoft.EntityFrameworkCore.Tools": "1.1.0-preview4-final",**  
 **"Microsoft.EntityFrameworkCore.Tools.DotNet": "1.1.0-preview4-final",**  
 ...

* Verifiera att verktygen installerades korrekt genom att köra dotnet restore från package manager eller kommandoprompten.
* Skapa databasen genom att köra kommandot dotnet ef database update

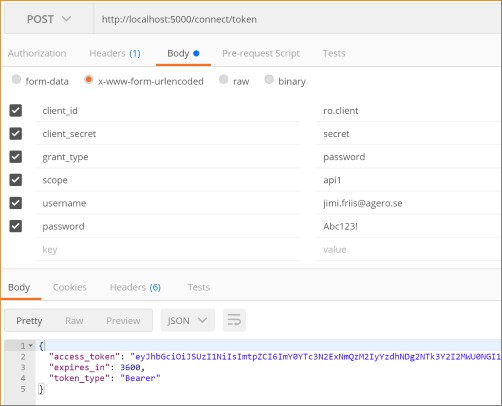
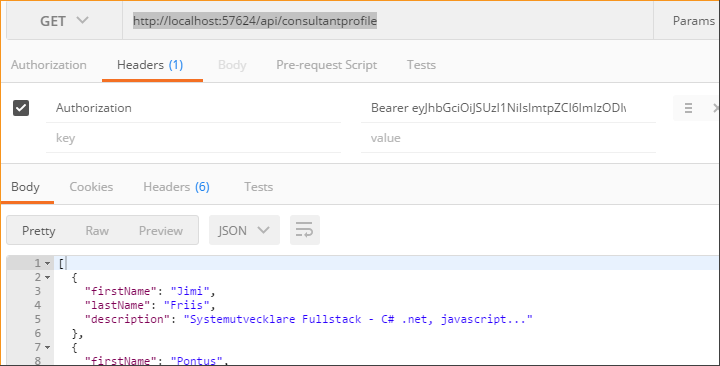


[**Creating a user**](https://identityserver4.readthedocs.io/en/release/quickstarts/6_aspnet_identity.html#creating-a-user)

* Starta både Api och IdentityServer
* Öppna identityserver (<http://localhost:5000>) i en webbläsare och registrera en användare (som standard krävs komplexa lösenord, t ex Abc123!)



Nu när vi har en användare kan vi testa vårt api från ett verktyg som [Postman](https://www.getpostman.com/)

* Hämta en token genom att anropa IdentityServer, <http://localhost:5000/connect/token,> från postman med inställningar enligt bilden. Värdena för client\_id, client\_secret, grant\_type och scope kommer från Config.GetClients – username och password är från användaren du nyss registrerade.   
  
* Som svar ska du få ett json-objekt innehållande en "access\_token". Kopiera access\_token och öppna en ny flik i Postman för att skapa ett anrop mot vårt api
* Gör ett api-anrop enligt bilden, mot <http://localhost:57624/api/consultantprofile,> med headern "Authorization" "Bearer [tokenstring]" där tokenstring är access\_token. Som svar ska du få konsultprofilerna.  
  

Har du inte redan haft en ordentlig paus är det hög tid för det innan vi går vidare – en kafferast eller kanske lunch, en promenad eller ett uppfriskande pass på gymmet?

**Reflektion**

Ok, nu har vi satt upp ett api som levererar konsultprofiler innehållande Förnamn, Efternamn och Beskrivning. Men just nu måste jag vara en registrerad användare för att kunna se profildata – och enligt kravet ska jag kunna se profilernas förnamn även om jag inte är registrerad.

Här kan vi välja olika lösningar

* Den ena är separata api actions (controller-metoder) där en action levererar data till icke autentiserade användare och en annan action levererar data till autentiserade användare.
* En annan lösning är att ta bort attributet "Authorize" från vår "Get"-action och kontrollera behörighet innuti metoden för att där returnera olika data beroende på...

Eftersom jag anser att den som frågar efter data ska veta vad den frågar efter samt att jag vill ha renare metoder i mitt api väljer jag alternativet med separata actions.

Visst hade det varit trevligt om klienten bara behövde anropa en metod och sen kunde visa upp data oavsett om det var hela profilerna eller bara förnamnen.  – Men det är mycket tydligare mot klienten att returnera "401 Unauthorized" än att returnera en begränsad datamängd om klienten förväntar sig all data. Dessutom blir som sagt koden renare med separata actions – och ren kod leder till bra kod.

**Uppdatera api enligt krav**

* Skapa en ny vymodell för att visa namn

public class ConsultantProfileNameViewModel  
 {  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 }

* Uppdatera repot med en metod för att hämta namn

public interface IConsultantProfileRepository  
 {  
 IEnumerable<ConsultantProfile> GetAll();  
 **IEnumerable<ConsultantProfileNameViewModel> GetAllNameOnly();**  
 }  
  
 public class InMemoryConsultantProfileRepository : IConsultantProfileRepository  
 ...  
   
  **public IEnumerable<ConsultantProfileNameViewModel> GetAllNameOnly()**  
 **{**  
 **return consultantProfiles.Select(p => new ConsultantProfileNameViewModel {FirstName = p.FirstName});**  
 **}**  
 ...

* Uppdatera controllerns actions enligt följande

//denna tidigare Get ersätts.  
//[HttpGet]  
//public IEnumerable<ConsultantProfile> Get()  
//{  
// return consultantProfileRepository.GetAll();  
//}  
  
[HttpGet]

[Route("")]//default Getmetod, bara för att jag vill ge metoden ett tydligare namn än "Get",

public IEnumerable<ConsultantProfileNameViewModel> GetConsultantProfileNames()

{

return consultantProfileRepository.GetAllNameOnly();

}

[HttpGet]

[Authorize]

[Route("GetFull")]

public IEnumerable<ConsultantProfile> GetFullConsultantProfiles()

{

return consultantProfileRepository.GetAll();

}

Nu kan vi testa implementationen

* Perfekt, <http://localhost:57624/api/consultantprofile/getfull> returnerar hela profilen och kräver autentisering – skulle du här få ett "Http 500 Internal Server Error" har du troligen glömt att starta IdentityServer.
* Default action <http://localhost:57624/api/consultantprofile> kräver ingen autentisering – men.. oops – vi får både för och efternamn, det stämmer inte helt med kraven...

Att det ska vara så svårt att hålla reda på så få krav – det är hög tid att införa tester innan det blir för krångligt och innan kraven ändras, för det vet man ju att de alltid gör.

Generellt förespråkar jag att utveckling görs testdrivet, men  – när man lär sig nya saker eller experimenterar med något tycker jag att det oftast är bättre att bara fokusera på det nya – å andra sidan kan det ibland underlätta att sätta upp tester för att enklare kunna debugga med breakpoints via testerna.

**Nytt projekt för att testa vårt api med xUnit**

Själv är jag ganska bekväm med NUnit men eftersom deras [testrunner för .NET Core fortfarande är i alphastadiet](https://github.com/nunit/dotnet-test-nunit) är det äntligen dags att jag sätter mig in i [xUnit](https://xunit.github.io/docs/getting-started-dotnet-core.html). xUnit är för visso i beta men jag har fått intrycket av att fler använder xUnit än NUnit mot .NET Core.

Jag använder mig av [Resharpers](https://www.jetbrains.com/help/resharper/sdk/Features/UnitTest/DotNetCore.html) Unit Test Sessions men du kan lika gärna köra tester i Visual Studios Test Explorer eller i en konsol, vilket jag beskriver i slutet av den här artikeln.

* Skapa ett nytt projekt, "Profil.Api.Test", av typen Class library (.NET Core)
* Lägg till nuget packages och config för xUnit enligt nedan i project.json (ersätt hela filen med detta)

{

"testRunner": "xunit",

"dependencies": {

"xunit": "2.2.0-beta2-build3300",

"dotnet-test-xunit": "2.2.0-preview2-build1029"  
 "moq": "4.6.38-alpha",  
 "FluentAssertions": "4.18.0",  
 "Newtonsoft.Json": "9.0.1",  
 "Profil.Api": "1.0.0-\*"

},

"frameworks": {

"netcoreapp1.1": {

"dependencies": {

"Microsoft.NETCore.App": {

"type": "platform",

"version": "1.1.0"

}

}

}

}

}

Nu ska du kunna köra tester från en kommandoprompt med kommandot dotnet test.

* Öppna kommandoprompten i projektet och verifiera att kommandot fungerar.

*Tips! Följ instruktionerna i* [*xUnit getting started*](https://xunit.github.io/docs/getting-started-dotnet-core.html) *för att göra ett första test om du inte är van med xUnit.*

**Skapa tester enligt kraven**

* Lägg till en ny testklass, ConsultantProfileControllerTest, och bygg upp tester enligt kraven

public class ConsultantProfileControllerTest  
 {  
 [Fact]  
 public void Unauthorized\_user\_can\_get\_profiles\_containing\_only\_first\_names\_from\_api()  
 {  
 var repo = new InMemoryConsultantProfileRepository();  
 var controller = new ConsultantProfileController(repo);  
  
 //eftrersom vi vill se hela objektet som json, precis så som klienten får data, behöver vi validera det serialiserade objektet  
 var expectedJson = JsonConvert.SerializeObject(new { FirstName = "Jimi" });  
  
 //serialisera första objektet (första konsultens namn)  
 var actualJson = JsonConvert.SerializeObject(controller.GetConsultantProfileNames().First());  
  
 actualJson.ShouldBeEquivalentTo(expectedJson, "\_ Endast förnamnet får lämnas ut till icke registrerade användare");  
 }  
  
 [Fact]  
 public void Authorized\_user\_can\_get\_full\_profile\_info\_from\_api()  
 {  
 var repoMock = new Mock<IConsultantProfileRepository>();  
 repoMock.Setup(x => x.GetAll()).Returns(GetMockedConsultantProfiles);  
  
 //För att kunna hämta data som en autentiserad användare behöver vi manipulera CurrentUser, vilken finns på ControllerContext.HttpContext.   
 //I "klassiska" .NET görs detta med "Thread.CurrentPrincipal = new GenericPrincipal..."  
 var currentUser = GetMockedCurrentUser("Jimi");  
 var mockedControllerContext = GetMockedHttpContext\_WithCurrentUser(currentUser);  
  
 var controller = new ConsultantProfileController(repoMock.Object)  
 {  
 ControllerContext = mockedControllerContext  
 };  
  
 var expectedJson = JsonConvert.SerializeObject(new  
 {  
 FirstName = "Jimi",  
 LastName = "Friis",  
 Description = "skriver ibland på BraKod.se"  
 });  
  
 var actualJson = JsonConvert.SerializeObject(controller.GetFullConsultantProfiles().First());  
  
 actualJson.ShouldBeEquivalentTo(expectedJson, "\_ En registrerad och autentiserad användare ska kunna se all information i konsultprofilerna");  
 }  
  
  
 private static GenericPrincipal GetMockedCurrentUser(string name)  
 {  
 return new GenericPrincipal(new GenericIdentity(name), new[] { "RegistreradAnvändare" });  
 }  
  
 private ControllerContext GetMockedHttpContext\_WithCurrentUser(ClaimsPrincipal currentUser)  
 {  
 //se svaren i denna tråd för mer info om hur du kan mocka IPrincipal och HttpContext i .NET Core  
 //http://stackoverflow.com/questions/38557942/mocking-iprincipal-in-asp-net-core  
 return new ControllerContext  
 {  
 HttpContext = new DefaultHttpContext  
 {  
 User = currentUser  
 }  
 };  
 }  
  
 private IEnumerable<ConsultantProfile> GetMockedConsultantProfiles()  
 {  
 return new List<ConsultantProfile>  
 {  
 new ConsultantProfile  
 {  
 FirstName = "Jimi", LastName = "Friis", Description = "skriver ibland på BraKod.se"  
 }  
 };  
 }  
  
 }

Jag väljer här att endast testa på controllernivå eftersom det är där gränsen går mot användarna/klienterna.

När jag försökte köra testet första gången fick jag följande fel

Unhandled Exception: System.IO.FileNotFoundException: Could not load file or assembly 'Microsoft.DotNet.InternalAbstractions, Version=1.0.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60'. The system cannot find the file specified.

at Xunit.Runner.DotNet.Program.GetAvailableRunnerReporters()

at Xunit.Runner.DotNet.Program.Run(String[] args)

at Xunit.Runner.DotNet.Program.Main(String[] args)

och lösningen var, helt enligt felmeddelandet, att lägga till "Microsoft.DotNet.InternalAbstractions": "1.0.0" i project.json

{

"testRunner": "xunit",

"dependencies": {  
 ...  
  **"Microsoft.DotNet.InternalAbstractions": "1.0.0"**  
 ...

Jag stötte även på ett problem med mockningen av repositoryt när jag använde Moq CallBase för att anropa den riktiga "GetAll"-metoden, vilket inte fungerade – det kan bero på en bugg och är inget jag löste eftersom jag tills vidare valde att gå direkt mot "InMemory"-implementationen. Har du lösningen på problemet får du gärna kontakta mig.

Mockningen såg ut så här:  
  
[Fact]  
public void Unauthorized\_user\_can\_get\_profiles\_containing\_only\_first\_names\_from\_api()  
...  
//i det här stadiet hade jag satt upp repot så att GetAllNameOnly() hämtade från GetAll().  
var repoMock = new Mock<IConsultantProfileRepository>();  
repoMock.Setup(x => x.GetAll()).Returns(GetMockedConsultantProfiles().AsQueryable);  
//CallBase anropar den riktiga metoden, måste dock sättas upp här eftersom klassen är mockad (man kan även sätta "CallBase = True" på mocken för att defaulta beteendet på alla metoder som inte är konfigurerade).  
repoMock.Setup(x => x.GetAllNameOnly()).CallBase();  
var controller = new ConsultantProfileController(repoMock.Object);  
...  
  
Felmeddelandet jag fick var:  
"This is a DynamicProxy2 error: The interceptor attempted to 'Proceed' for method 'System.Collections.Generic.IEnumerable`1[Profil.Api.Models.ConsultantProfileNameViewModel] GetAllNameOnly()'   
which has no target.When calling method without target there is no implementation to 'proceed' to and it is the responsibility   
of the interceptor to mimic the implementation(set return value, out arguments etc) at Castle.DynamicProxy.AbstractInvocation.ThrowOnNoTarget()"

Åter till testet.

* Gör korrigeringar i koden för att få testerna gröna, vill du ladda ner den färdiga koden finns den [här på github](https://github.com/JimiSweden/dotnetcoreBloggserie).
* Efter implementationen av testerna uppfyller vi kraven för åtkomst av data.

Härligt! – Vi är nu färdiga med första implementationsdelen i den här serien, i nästa del fortsätter vi med att bygga administrationen för profilerna i ASP:NET Core MVC.

*Tips! Om du inte redan prenumererar på bloggen så anmäl dig nu för att få uppdateringar genom att skriva in din e-postadress här till höger.*

**Att köra testerna från kommandoprompten**

Jag använder normalt sett Resharpers Unit Test Sessions – *men det går lika bra med selleri...*

Tester från kommandoprompten kör du med kommandot dotnet test i testprojektet (dotnet test --help för att se alternativ).

Något jag har vant mig vid från tester i javascript och typescript, med [Jasmine](https://jasmine.github.io/) och [Karma](https://karma-runner.github.io), är att ha en "watcher" som övervakar projektfilerna och kör testerna när något ändrats, detta kan du även göra här.

* För att köra tester kontinuerligt lägger du till nugetpaketet nedan i testprojektets "tools" i project.json
* "tools": {
* "Microsoft.DotNet.Watcher.Tools": "1.1.0-preview4-final"

},

och kör kommandot dotnet watch test, testrunnern triggas då automatiskt vid filändringar i testprojektet och de projekt som refereras av det – mycket trevligt så länge du inte kör tunga integrationstester :-)