ASP.NET Core

Sommaire

- Introduction
- Startup
  - injection de dépendances
  - Hosting
  - Configuration
  - Static files
- Middleware
- Entity Framework Core
- Razor Page
- MVC
- WebAPI
- Validation
- Blazor
- Authentication

Introduction

- ASP.NET Core est un framework multiplateforme et open source pour:
  - Applications et des services web
  - Applications IoT

- Outils de développement sur Windows, macOS et Linux avec:
  - Visual Studio
  - Visual Studio Code

- Exécuter sur .NET Core et .NET Framework

Introduction

- Framework unifié pour des sites web et des API web
- Testable facilement
- Les razor pages permettent de de faire des sites web orientés page facilement
- Injection de dépendances intégrée
- Leger, performant et modulaire
- Modèle MVC

Introduction

- ASP.NET Core 3.0 et ultérieur s’exécute uniquement sur .NET Core
  - Multiplateforme.
  - S’exécute sur macOS, Linux et Windows
  - Performances améliorées
  - Gestion des versions côte à côte
  - Nouvelles API
  - Open source

Startup et injection de dépendances

- La classe Startup permet
  - L'enregistrement des services dans la méthode ConfigureServices
  - Configuration du pipeline de traitement avec ses middlewares dans la méthode Configure

- ASP.NET propose une infrastructure d'injection de dépendances intégrée
- Le conteneur par default ASP.NET peut être remplacé par un autre conteneur de service

Startup et injection de dépendances

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc()
            .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

        services.AddDbContext<MovieContext>(options =>
                options.UseSqlServer(Configuration.GetConnectionString("MovieDb")));
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseMvc();
    }
}
```

Startup et injection de dépendances

- Une dépendance est un objet qui nécessite un autre objet.
- Les problème de dépendance :
  - Pour remplacer une dépendance par une autre implémentation, la classe doit être modifiée.
  - Si la dépendances possède des dépendances, elles doivent être configurées par la classe.
  - Dans un grand projet comportant plusieurs classes dépendant d'une même dépendance, le code de configuration est disséminé dans toute l'application.
  - Complique le test unitaire.

Startup et injection de dépendances

- L’injection de dépendance permet
  - L’utilisation d’une interface ou classe de base pour extraire l’implémentation des dépendances
  - L’inscription de la dépendance dans un conteneur de service
  - L’injection de la dépendance dans le constructeur de la classe où elle est utilisé

- 3 durée de vie possible
  - Transcient: créés chaque fois qu’ils sont demandés.
  - Scoped: créés une seule fois par requête
  - Singleton: créés la première fois qu’ils sont demandés

Startup et injection de dépendances

- Les applications ASP.NET Core configurent et lancent un hôte responsable de la gestion du démarrage et de la durée de vie des applications.
- 2 hôtes possible: hôte générique (recommandé) et hôte web (pour la retro comptabilité)
- La configuration de l'hôte se trouve dans la classe Program et fournis les services suivant:
  - Injection de dépendances
  - Journalisation
  - Configuration
  - Implémentations de IHostedService (pour une application web, cela démarre le serveur multiplateforme Kestrel)
- Kestrel peut être exécuté et exposé directement à Internet mais il est le plus souvent exécuté dans une configuration de proxy inverse avec Nginx ou Apache

Startup et injection de dépendances

- La configuration d'application dans ASP.NET Core est basée sur des paires clé-valeur
- Plusieurs source possible
  - Azure Key Vault
  - Configuration de Azure App
  - Arguments de ligne de commande
  - Fournisseurs personnalisés (installés ou créés)
  - Fichiers de répertoire
  - Variables d'environnement
  - Objets .NET en mémoire
  - Fichiers de paramètres

Startup et injection de dépendances

- Configuration par défaut pour l'hôte générique
  - Variables d'environnement préfixées avec DOTNET_ (par exemple, DOTNET_ENVIRONMENT)
  - Arguments de ligne de commande
  - Si hôte web, variables d'environnement préfixées avec ASPNETCORE_ (par exemple, ASPNETCORE_ENVIRONMENT)
    - Exemple dans le fichier launchSettings.json
  - Configuration de l'application
    - *appSettings.json*
    - *appsettings.{Environment}.json (remplacera les valeurs de appsettings.json)*

Startup et injection de dépendances

- Environnements (ASPNETCORE_ENVIRONMENT)
  - 3 valeurs sont prises en charge
    - Development
    - Staging
    - Production
  - Vous pouvez définir vos propres valeurs
  - Si aucune valeur n’est définie la valeur par défaut est Production.

Startup et injection de dépendances

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseMvc();
}
```

Startup et injection de dépendances

- Appsettings.json

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "AllowedHosts": "*",
    "Demo": {
        "key1": "value1",
        "group1": {
            "key2": "value2"
        }
    }
}
```

Startup et injection de dépendances

- Startup

```
var val1 = Configuration.GetValue<string>("Demo:key1"); // value 1
var section = Configuration.GetSection("Demo");
val1 = section.GetValue<string>("key1"); // value 1

var demoConfig = Configuration.GetSection("Demo").Get<DemoConfig>();
```

```
class DemoConfig
{
    0 references
    public string Key1 { get; set; }
    0 references
    public Group1 Group1 { get; set; }
}

1 reference
class Group1
{
    0 references
    public string Key2 { get; set; }
}
```

Startup et injection de dépendances

- Le secret Manager stocke les données sensibles pendant le développement d'un projet ASP.NET Core
- Il ne chiffre pas les clés stockées mais évite de les avoir dans code du projet ASP.NET
- Chemin windows

Chemin d'accès au système de fichiers :

```
%APPDATA%\Microsoft\UserSecrets\<user_secrets_id>\secrets.json
```

- Chemin macOS/Linux

Chemin d'accès au système de fichiers :

```
~/.microsoft/usersecrets/<user_secrets_id>/secrets.json
```

Startup et injection de dépendances

- Pour lier le projet au secret manager
  - Windows: cliquer **gérer les secrets d'utilisateur** dans le menu contextuel.
  - macOS / Linux:
    - dotnet user-secrets init
    - dotnet user-secrets set "Auth:Password" "12345"

- Dans les 2 cas cela ajoute le UserSecretsId dans le csproj.

```
var password = Configuration.GetValue<string>("Auth:Password");
```

Startup et injection de dépendances

- Les fichiers statiques (HTML, CSS, images, JavaScript etc) sont des ressources qu'une application ASP.NET Core délivre directement aux clients.
- Une configuration est nécessaire pour pouvoir délivrer ces fichiers.

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();
}
```

- Structure par défaut d'une application web pour les fichiers statiques

- **wwwroot**
  - **css**
  - **images**
  - **js**

Startup et injection de dépendances

- Vous pouvez délivrer des fichiers statiques en dehors du wwwroot

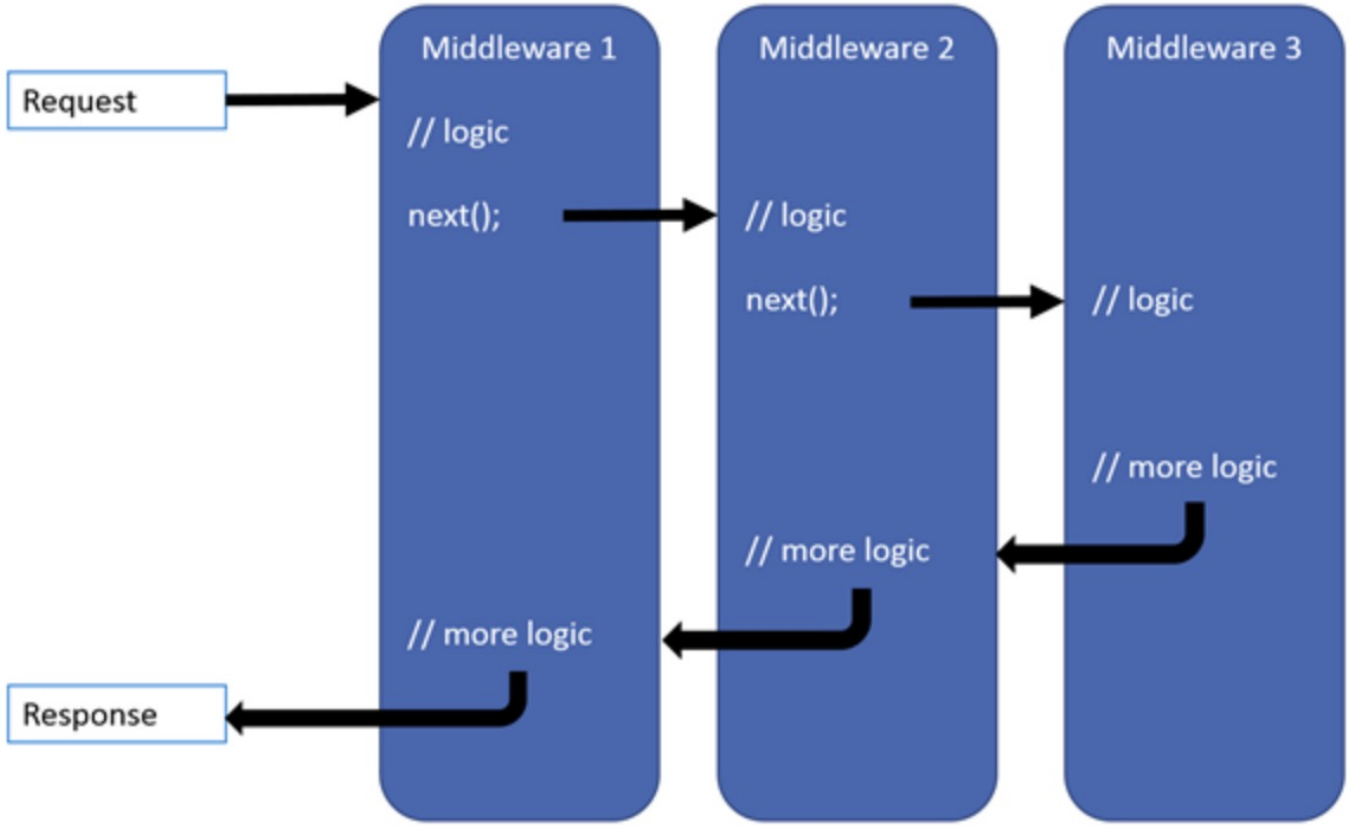
```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles(); // For the wwwroot folder

    app.UseStaticFiles(new StaticFileOptions
    {
        FileProvider = new PhysicalFileProvider(
            Path.Combine(Directory.GetCurrentDirectory(), "MyStaticFiles")),
        RequestPath = "/StaticFiles"
    });
}
```

- **wwwroot**
  - **css**
  - **images**
  - **js**
- **MyStaticFiles**
  - **images**
    - *banner1.svg*

Middleware

- Un middleware est un composant du pipeline d'application pour gérer les requêtes et les réponses
- Chaque middleware
  - Choisit de passer la requête au composant suivant dans le pipeline.
  - Peut exécuter du code avant et après le composant suivant dans le pipeline.
- Les délégués de requête sont configurés à l'aide des méthodes d'extension Run , Map, Use

Middleware


Request
Middleware 1
// logic
next();
// more logic
Response
Middleware 2
// logic
next();
// more logic
Middleware 3
// logic
// more logic

Middleware

- L’ordre dans lequel les middleware sont ajoutés dans la
méthode Configure définit l’ordre dans lequel ils sont appelés sur les requêtes

- C’est l’ordre inverse pour la réponse

```
public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

Middleware

- Il est possible de créer une classe de middleware
- Elle doit inclure
  - un constructeur public avec un paramètre de type RequestDelegate
  - une méthode publique nommée Invoke ou InvokeAsync qui doit retourner une Task
  - Avoir un premier paramètre de type HttpContext
- Les paramètres supplémentaires pour le constructeur et Invoke/InvokeAsync seront remplis par injection de dépendances
- Singleton obligatoire pour les paramètres du constructeur

Middleware

- Pour utiliser un middleware il faut appeler la methode UseMiddleware<T>
- On utilise souvent une méthode d'extension pour cela

```
public static class RequestCultureMiddlewareExtensions
{
    public static IApplicationBuilder UseRequestCulture(
        this IApplicationBuilder builder)
    {
        return builder.UseMiddleware<RequestCultureMiddleware>();
    }
}
```

Middleware

- Exemple
  - Le middleware Fichier statique est appelé tôt pour procéder au court-circuit sans passer par les composants suivants
  - Le middleware Fichier statique ne fournit **aucune** vérification d'autorisation et sont disponibles publiquement

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseDatabaseErrorPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseCookiePolicy();
    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
    app.UseSession();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Entity Framework Core

- Configuration d'entity framework avec asp.net core
  - Le context doit utiliser le contructeur avec DbContextOptions
  - Vous pouvez enregister la connection string dans le fichier appsettings.json

```
"ConnectionStrings": {
    "MyContext": "Data Source=Database.db"
}
```

- Outil EF pour macOS / Linux
  - dotnet tool install --global dotnet-ef
  - dotnet add package Microsoft.EntityFrameworkCore.SQLite
  - dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
- Et pour la génération automatique
  - dotnet add package Microsoft.EntityFrameworkCore.Design
  - dotnet add package Microsoft.EntityFrameworkCore.SqlServer

Entity Framework Core

- Enregistrement du context EF

```
services.AddDbContext<MyContext>(options =>
    options.UseSqlite(Configuration.GetConnectionString("MyContext")));
```

- Scaffolding
  - Windows: menu depuis visual studio
  - macOS / Linux
    - dotnet tool install --global dotnet-aspnet-codegenerator
  - Exemple pour RazorPage avec EF
    - dotnet aspnet-codegenerator razorpage -m Contact -dc MyContext -udl -outDir Pages/Contacts –referenceScriptLibraries
    - -udl: use default Layout
    - -referenceScriptLibraries: Ajoute _ValidationScriptsPartial aux pages Modifier et Créer.

Entity Framework Core

- Migration EF
- CLI
  - dotnet ef migrations add init
  - dotnet ef database update
- Visual Studio
  - Add-Migration init
  - Update-Database

Razor Page

- Plus simple et plus productifs que MVC
- Créer un nouveau projet (CLI)

dotnet new webapp -o demo

(macOS: dotnet dev-certs https –trust)

cd demo

donet run

Razor Page

- Les pages Razor sont dérivées de PageModel
- Convention de nommage <MaPage>Model
- Injection de dépendances dans le constructeur de la page
- Les requêtes GET entrantes vont exécuter OnGetAsync ou OnGet
- Les requêtes POST entrantes vont exécuter OnPostAsync ou OnPost
- Razor peut passer du HTML au C# ou à des balises spécifiques à Razor avec le symbole @
- Si @ est suivi d'un mot clé réservé Razor, il est converti en balise spécifique à Razor
- Sinon, il est converti en C#.

Razor Page

- @page doit être la première directive Razor sur une page
- La directive @model spécifie le type du modèle passé à la page Razor.

```
@page
@model demoWebApp.Pages.Contacts.IndexModel

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-page="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].FirstName)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Contact[0].LastName)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model.Contact) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.FirstName)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.LastName)
            </td>
            <td>
                <a asp-page="./Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-page="./Details" asp-route-id="@item.Id">Details</a> |
                <a asp-page="./Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>
```

Razor Page

- L’attribut [BindProperty] permet la liaison de données.
- Quand le formulaire publie les valeurs de formulaire, ASP.NET lie les valeurs publiées au modèle avec l’attribut [BindProperty].
- Le modèle peut ensuite être utilise dans la méthode OnPost / OnPostAsync

Razor Page

- Injection du context entity framework par constructeur
- La propriété contact est expose a la liaison de données grâce à l'attribut [BindProperty]
- Dans la méthode OnPostAsync, le Contact est ajoute a la base de données

```
0 references
public class CreateModel : PageModel
{
    3 references
    private readonly MyContext _context;

    0 references
    public CreateModel(MyContext context)
    {
        _context = context;
    }

    0 references
    public IActionResult OnGet()
    {
        return Page();
    }

    [BindProperty]
    1 reference
    public Contact Contact { get; set; }

    0 references
    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Contact.Add(Contact);
        await _context.SaveChangesAsync();

        return RedirectToPage("./Index");
    }
}
```

Razor Page

- Les Filters (IPageFilter et IAsyncPageFilter) permettent
  - Exécuter le code après la sélection d'une méthode (OnGet, OnPost) mais avant la liaison de données.
    - OnPageHandlerSelected et OnPageHandlerSelectionAsync
  - Exécuter le code avant l'exécution de la méthode, une fois la liaison de données terminée.
    - OnPageHandlerExecuting et OnPageHandlerExecutionAsync
  - Exécuter le code après l'exécution de la méthode.
    - OnPageHandlerExecuted
  - Peuvent être implémentés dans une page ou globalement.
  - Il faut choisir synchrone OU asynchrone mais pas les 2
- Ils ressemblent beaucoup aux filtres d'action ASP.NET MVC

Razor Page

- Enregistrement global d’un filtre

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc(options =>
    {
        options.Filters.Add(new SampleAsyncPageFilter(_logger));
    });
}
```

MVC

- Créer un nouveau projet (CLI)

dotnet new mvc -o demo

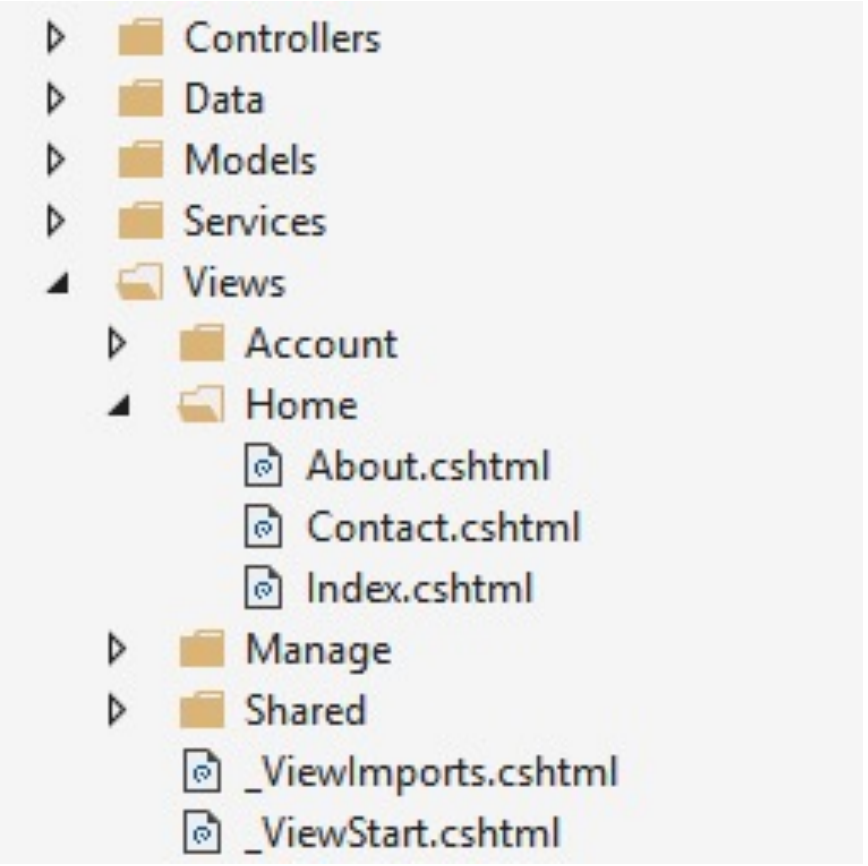
(macOS: dotnet dev-certs https –trust)

cd demo

donet run

MVC

- Template et convention
  - AccountController
  - HomeController
    - Action About
    - Action Contact
    - Action Index



MVC

- Les méthodes publiques sur un contrôleur sont des « Actions », sauf celles qui sont décorées avec l'attribut [NonAction]
- Passage de données du contrôleur vers la vue
  - Le dictionnaire ViewData
    - Controller: `ViewData["Message"] = "Hello " + name;`
    - Vue: `<li>@ViewData["Message"]</li>`

  - L'objet dynamique ViewBag qui est un wrapper autour de ViewData
  - Envoyer un Modèle ou ViewModel à la vue (@model)

MVC

- Scafolding
  - dotnet aspnet-codegenerator controller -name ContactsController -m Contact -dc MyContext --relativeFolderPath Controllers --useDefaultLayout --referenceScriptLibraries
- Injection de dépendances par constructeur dans le controlleur
- Envoi du modèle à la vue

```
public async Task<IActionResult> Index()
{
    return View(await _context.Contact.ToListAsync());
}
```

MVC

- Le routage basé sur les conventions vous permet de définir globalement les formats d'URL acceptés par votre application
  - {controller=Home}/{action=Index}/{id?}

- Le routage par attributs vous permet de spécifier des informations de routage en décorant vos contrôleurs et vos actions avec des attributs

```
[Route("api/[controller]")]
public class ProductsController : Controller
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
      ...
    }
}
```

MVC

- Placez les routes « globales » le plus loin dans la table de routage.
- Le routage par attributs nécessite plus de configuration que la route conventionnelle par défaut
- Le routage par attributs force un contrôle plus précis
- Les applications MVC peuvent combiner l'utilisation du routage conventionnel et du routage par attributs
- Souvent on utilise le routage conventionnel pour la partie Html et le routage par attributs pour les API REST
- Tout attribut de route sur le contrôleur a pour effet que toutes les actions du contrôleur sont routées par attributs et ne sont plus accessibles via les routes conventionnelles

MVC

- Les vues partielles réduisent la répétition du code et permettent de découper des fichiers cshtml volumineux
- Si du code C# est necessaire vous devez utiliser plutôt des composants de vue
- En MVC, le ViewResult d'un contrôleur peut retourner une vue ou une vue partielle
- Par convention le nom des vues partielles commencent par _
- Avec razor vous pouvez utiliser <partial> ou @await Html.PartialAsync(…)

MVC

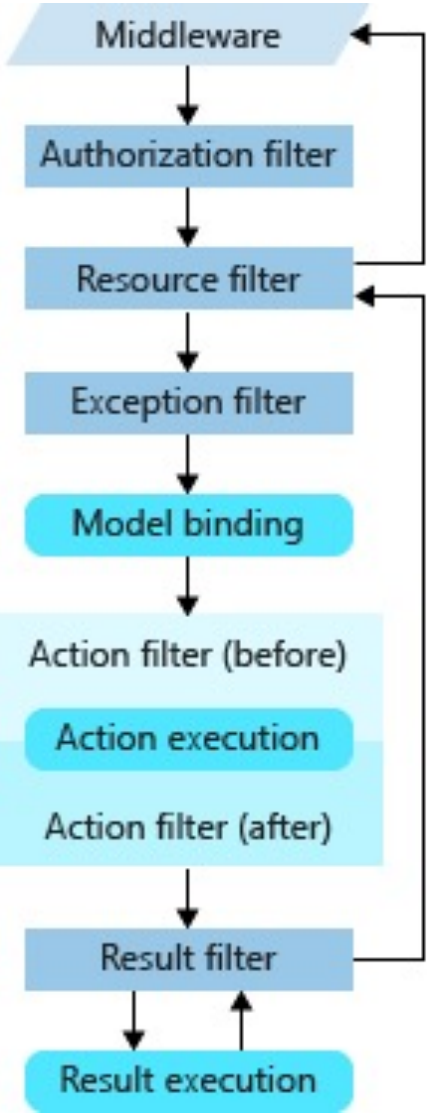
- Ordre de découverte des vues partielles
  - /Areas/<Area-Name>/Views/<Controller-Name>
  - /Areas/<Area-Name>/Views/Shared
  - /Views/Shared
  - /Pages/Shared

MVC

- Les filtres permettent d’exécuter du code avant ou après des étapes spécifiques dans le pipeline de traitement des requêtes.
- Les filtres synchrones peuvent exécuter du code avant (On-?-Executing) et après (On-?-Executed) leur étape de pipeline.
- Par exemple, OnActionExecuting est appelé avant l’appel de la méthode d’action. OnActionExecuted est appelé après le retour de la méthode d’action.

MVC

- Comment ces types de filtres interagissent dans le pipeline de filtres
- Filtres de ressources
  - IResourceFilter ou IAsyncResourceFilter.
- Filtres d'actions
  - IActionFilter ou IAsyncActionFilter.
- Filtres d’exceptions
  - IExceptionFilter ou IAsyncExceptionFilter.
- Filtres de résultats :
  - IResultFilter ou IAsyncResultFilter



Validation

- Les DataAnnotations fournissent un ensemble d'attributs de validation et de mise en forme
- assembly: System.ComponentModel.DataAnnotations
- Les plus courant
  -> Required
  -> StringLength
  -> Regex
  -> Range
  -> DataType

- La validation est appliquée à la fois côté client (jQuery) et côté serveur.

Validation

- Model

```
public class Session
{

    public int Id { get; set; }

    [Required]
    [StringLength (10)]
    public string Title { get; set; }

    [Required]
    public string Description { get; set; }

    [MasterValidationAttribute]
    public string Speaker { get; set; }

    [Range(100, 500, ErrorMessage = "Insufficient number of attendees")]
    public int NbAttendees { get; set; }
}
```

Validation

- Controller

```
[HttpPost]
public ActionResult Create(Session session)
{
    if (ModelState.IsValid)
    {
        context.Sessions.Add(session);
        context.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(session);
}
```

Validation

- Vue

```
<form asp-action="Create">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label asp-for="Title" class="control-label"></label>
        <input asp-for="Title" class="form-control" />
        <span asp-validation-for="Title" class="text-danger"></span>
    </div>
```

Validation

- Les DataAnnotations fournissent un ensemble d'attributs de validation et de mise en forme
- assembly: System.ComponentModel.DataAnnotations
- Les plus courant
  -> Required
  -> StringLength
  -> Regex
  -> Range
  -> EmailAddress
  -> DataType
  -> Display
  -> DisplayFormat

- La validation est appliquée à la fois côté client (jQuery) et côté serveur.

Validation

- Attributs personnalisés avec la classe ValidationAttribute et la methode IsValid

- Validation au niveau de la classe avec l'interface IValidatableObject (exécuté après la validation des propriétés)

Validation

- Exemple de ValidationAttribute

```
public class MasterValidationAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        if (value != null && value.ToString() == "master")
            return true;
        else
            return false;
    }
}
```

Validation

- Exemple avec IValidatableObject

```
public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
{
    //test for something
    yield return new ValidationResult("no no no");

    //test for something else
    yield return new ValidationResult("no no no no");
}
```

Web API

- Créer un nouveau projet (CLI)

dotnet new webapi -o demo

(macOS: dotnet dev-certs https –trust)

cd demo

donet run

Web API

- Scafolding
  - dotnet aspnet-codegenerator controller -name ContactsController -m Contact -dc MyContext -async –api

- Injection de dépendances par constructeur dans le controlleur
- Les contrôleurs web api dérive de ControllerBase et utilise l’attribut [ApiController] qui
  - force le routage par attributs
  - Renvoi automatiquement une erreur 400 si le modèle n’est pas valide
  - Evite de devoir utiliser de partout les attributs [FromBody], [FromForm], [FromRoute], [FromQuery]

Web API

- Pour utiliser l'API Web vous pouvez utiliser OpenAPI (Swagger)
- Package Swashbuckle.AspNetCore
- Ajout du service

```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });
});
```

- Activer les middlewares

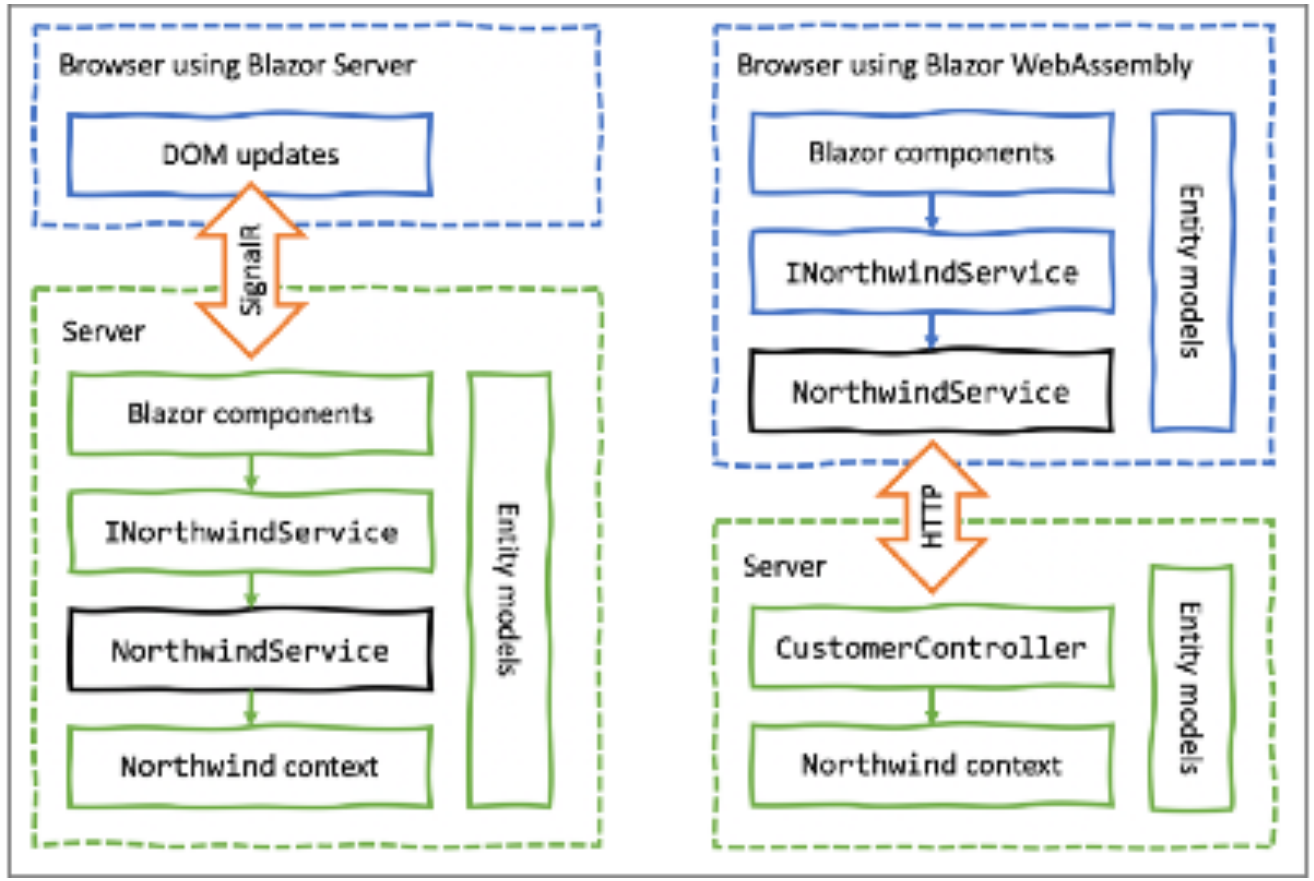
```
// Enable middleware to serve generated Swagger as a JSON endpoint.
app.UseSwagger();

// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
// specifying the Swagger JSON endpoint.
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
});
```

Web API

- Pour décrire votre API, vous pouvez utiliser les attributs
  - [Produces("application/json")]
  - [ProducesResponseType(400)]
  - [ApiConventionType]
- ApiConventionType peut être utiliser
  - sur une action
    - [ApiConventionMethod(typeof(DefaultApiConventions), nameof(DefaultApiConventions.Put))]
  - sur un controlleur
    - [ApiConventionType(typeof(DefaultApiConventions))]
  - sur un namespace
    - [assembly: ApiConventionType(typeof(DefaultApiConventions))]

Blazor

- Interfaces utilisateur riches et interactives en C# au lieu de JavaScript
- Partager du code côté serveur et côté client
- 2 types de projet:
  - Blazor WebAssembly (code client seulement avec possibilité d'avoir 1 autre projet pour le server web api)
  - Blazor Server


Browser using Blazor Server
DOM updates
SignalR
Server
Blazor components
INorthwindService
NorthwindService
Northwind context
Entity models
Browser using Blazor WebAssembly
Blazor components
INorthwindService
NorthwindService
Entity models
HTTP
Server
CustomerController
Northwind context
Entity models

Blazor Server

- Avec Blazor server le code est exécuté cote serveur.
- Il utilise une connexion SignalR
- Les changements sont envoyés sur le serveur et le nouveau rendu est envoyé en retour
- Le client ne contient que du HTML, CSS et JavaScript

Blazor WebAssembly

- Blazor WebAssembly utilise des normes Web ouvertes sans plug-ins
- Fonctionne dans tous les navigateurs Web modernes, mobile compris
- Les fichiers de code C# et les fichiers Razor sont compilés dans des assemblys .net
- On peut accéder aux fonctionnalités complètes du navigateur via JavaScript grâce à l'interopérabilité js
- Les assemblys et le runtime .net sont téléchargés dans le navigateur
  - Le code inutilisé est supprimé de l'application
  - Les réponses HTTP sont compressées
  - Le runtime .NET et les assemblys sont mis en cache dans le navigateur
- Permet de réaliser des applications hors ligne
- Support pour les Progressive Web Apps

Blazor WebAssembly

Routing:

Le Router se trouve dans le composant App et permet de router les composant avec la directive @page

Le layout par défaut se trouve dans le router

Vous pouvez utiliser plusieurs assemblies pour trouver des composants

```
<Router
    AppAssembly="@typeof(Program).Assembly"
    AdditionalAssemblies="new[] { typeof(Component1).Assembly }">
    @* ... Router component elements ... *@
</Router>
```

Blazor WebAssembly

Paramètre de routage:

- @page /mapage/{param}
- @code {

```
        [Parameter]
        public string Param { get; set; }
}
```

- Paramètre optionel: {text?}
- Contrainte: {id:int}

Blazor WebAssembly

Component:

- Tout les fichiers .razor sont des composants (même les pages)
- Un composant c’est donc un fichier .razor avec du code C# dans @code ou dans un fichier .razor.cs
- OnInitialized{Async} permet d’initialiser un composant, après avoir reçu ses paramètres initiaux
- OnParametersSet{Async} pour setter un parameter, après avoir été initialisé
- @layout pour choisir le layout

Blazor WebAssembly

Les composants peuvent utiliser l'injection de dependance avec @inject ou [Inject]:

- Scopes (singles dans WebAssembly)
- Singleton
- Transient

Les composants peuvent contenir d'autres composants en les declarant avec une balise HTML

Pour appliquer un attribut: @attribute [MonAttribut]

Les balises tag Helper ne sont pas prises en charge dans les composants

Liaison de données avec @bind

Ecouter un evenement avec @on{Event}

On peut exposer un evenement depuis un composant enfant avec le type EventCallback

Blazor WebAssembly

```
<p>
    <button @onclick="OnClickCallback">
        Trigger a Parent component method
    </button>
</p>

@code {
    [Parameter]
    public string Title { get; set; }

    [Parameter]
    public RenderFragment ChildContent { get; set; }

    [Parameter]
    public EventCallback<MouseEventArgs> OnClickCallback { get; set; }
}
```

```
@page "/parent"

<h1>Parent-child example</h1>

<Child Title="Panel Title from Parent" OnClickCallback="@ShowMessage">
    Content of the child component is supplied by the parent component.
</Child>

<p>@message</p>

@code {
    private string message;

    private void ShowMessage(MouseEventArgs e)
    {
        message = $"Blaze a new trail with Blazor! ({e.ScreenX}:{e.ScreenY})";
    }
}
```

Blazor WebAssembly

Formulaire et validation:

- Le composant EditForm aide a créer un formulaire et fonctionne avec les DataAnnotations
- Le Model du formulaire permet de créer un EditContext qui track les metadata de l'edition (par exemple quel champs a changé)

```
<EditForm Model="@exampleModel" OnValidSubmit="@HandleValidSubmit">
    <DataAnnotationsValidator />
    <ValidationSummary />

    <InputText id="name" @bind-Value="exampleModel.Name" />

    <button type="submit">Submit</button>
</EditForm>
```

Blazor WebAssembly

- Composants de formulaire intégrés:
  - InputText
  - InputCheckbox
  - InputNumber
  - InputSelect
  - InputDate
  - InputFile
  - InputRadio

- Ces composants fournissent un comportement par défaut pour la validation
- Tous les composants d’entrée, y compris EditForm , prennent en charge les attributs perso et ils seront ajoutés à l’élément HTML rendu.

Blazor WebAssembly

Javascript interop:

- Ajouter le script dans le fichier wwwroot/index.html après _framework/blazor.{webassembly|server}.js
- Injecter IJSRuntime dans la page et utiliser la methode InvokeAsync
- L'identificateur de la fonction est relatif à l'objet js window. Pour appeler window.someScope.someFunction , l'identificateur est someScope.someFunction.

Blazor WebAssembly

Pour appeler du .NET depuis JS:

- Exposer une méthode public et static avec l'attribut [JSInvokable]
- Utiliser la fonction js DotNet.InvokeMethodAsync
- Pour utiliser une méthode d'instance, la fonction javascript doit prendre en paramètre l'objet
  - (window.myFunc = (obj) => obj.invokeMethodAsync(…))
- Pour créer une référence d'objet: DotNetObjectReference.Create(…)
- Il faut penser à Dispose la référence d'objet dans la méthode dispose de la page

Blazor WebAssembly

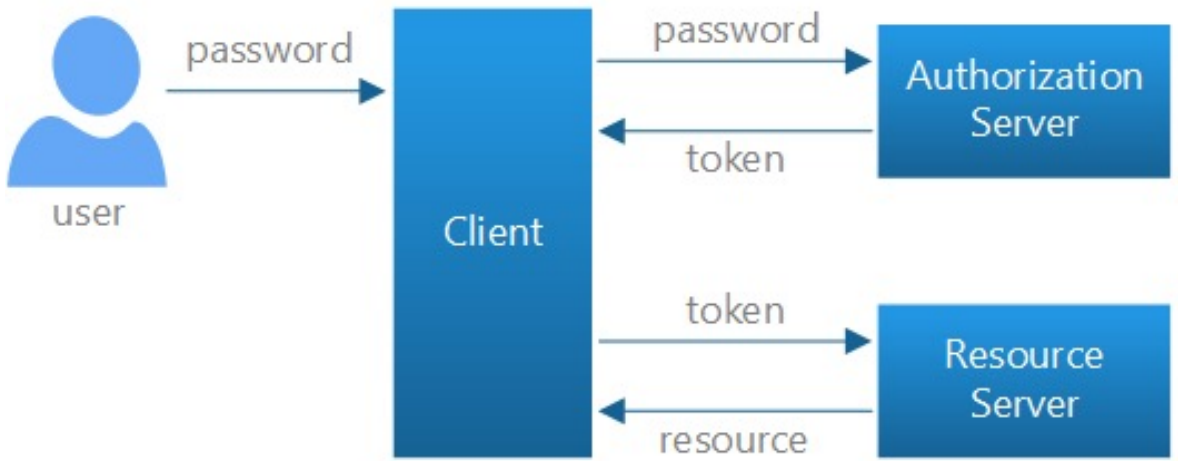
```
private DotNetObjectReference<MyComponent> objRef;

public async Task TriggerDotNetInstanceMethod()
{
    objRef = DotNetObjectReference.Create(this);
    result = await JS.InvokeAsync<string>("myFunc", objRef);
}

[JSInvokable]
public string HelloWorld() => "hello world";

public void Dispose()
{
    objRef?.Dispose();
}
```

```
<script>
    window.myFunc = (dotNetHelper) => {
        return dotNetHelper.invokeMethodAsync('HelloWorld');
    };
</script>
```

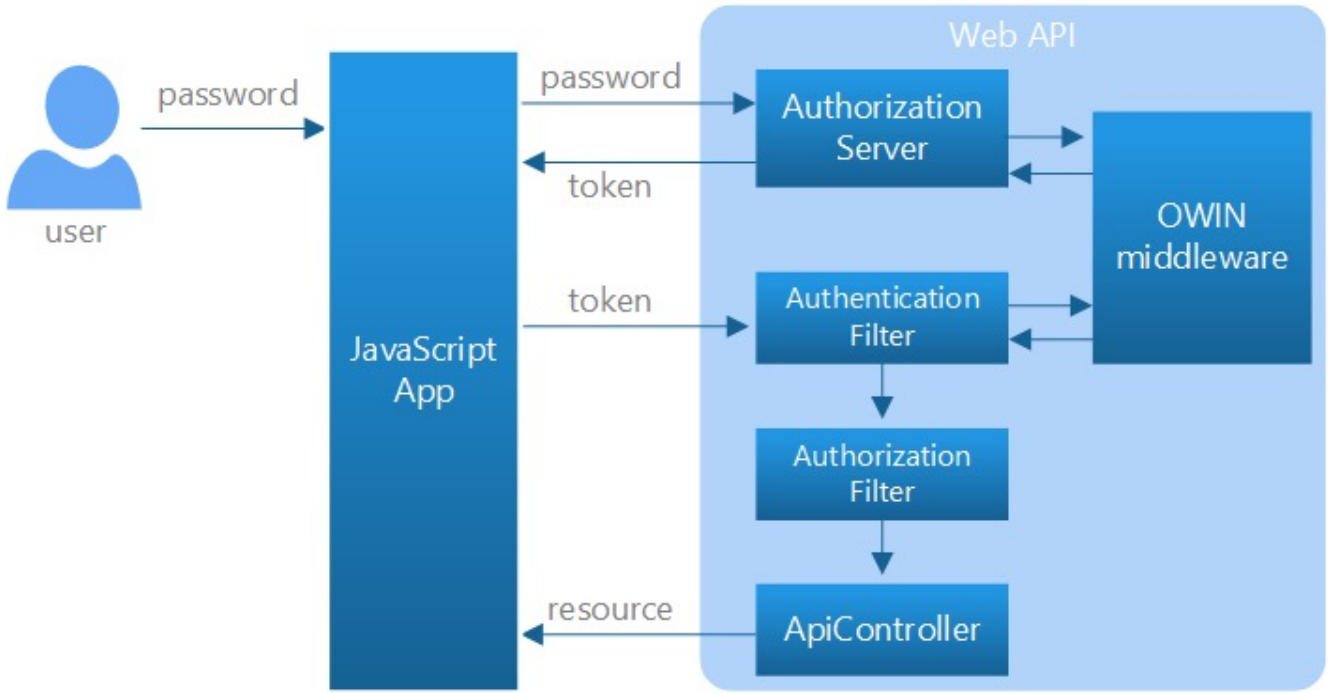
OAuth

- OAuth est un protocole qui permet aux applications de demander des jetons d'accès à un service de jetons de sécurité et de les utiliser pour communiquer avec des API.
- Terminologie
  - Resource: donnée à protéger
  - Resource server: le serveur qui à les ressources
  - Resource owner: peut accorder la permission d'accéder à une ressource
  - Client: application qui veut accéder aux ressources
  - AccessToken: jeton qui accorde l'accès à une ressource
  - Bearer Token: jeton d'accès particulier (pas de secret, besoin de https)
  - Authorization server: serveur fournissant le jeton d'accès

OAuth


password
user
Client
password
token
Authorization Server
token
resource
Resource Server

OAuth avec web api



OAuth avec web api

- Authentification avec Identity framework:
  - Package Microsoft.AspNetCore.Identity.EntityFrameworkCore

OAuth avec web api

- **IdentityContext**: entity framework DbContext
- **IdentityUser**: représente un utilisateur (IUser)
- **IdentityRole**: représente un Role (IRole)
- **IdentityClaim**: une revendication d'utilisateur spécifique
- **IdentityUserLogin**: identifiant utilisateur externe (facebook, google…)
- **IdentityUserRole**: représente un utilisateur appartenant à un rôle

OAuth avec web api

```
1 reference | 0 changes | 0 authors, 0 changes
public class MyUser : IdentityUser
{ }

0 references | 0 changes | 0 authors, 0 changes
public class DemoContext : IdentityDbContext<MyUser>
{ }
```

Asp.net core Identity

- Types utiles:
  - UserManager
  - SignInManager
  - RoleManager
  - IUserClaimsPrincipalFactory: Fournit des méthodes pour créer un principal de revendications pour un utilisateur donné.

Asp.net core Identity

- ConfigureServices

```
services.AddDbContext<MyContext>(options =>
    options.UseSqlite(Configuration.GetConnectionString("MyContext")));
services.AddIdentity<User, IdentityRole>()
    .AddEntityFrameworkStores<MyContext>()
    .AddDefaultTokenProviders();
```

Asp.net core Identity

- ASP.NET Core contient un middleware qui permet à une application de recevoir un jeton
- Install-Package
  - Microsoft.AspNetCore.Authentication.JwtBearer
  - System.IdentityModel.Tokens.Jwt

Json Web Token

- AddAuthentication et AddJwtBearer dans la classe Startup

```
services.AddAuthentication(x =>
{
    x.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    x.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(x =>
{
    x.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(key),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

Json Web Token

- Generate JWT:

```
private string GenerateJWT(User user)
{
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
    var claims = new[] {
        new Claim(JwtRegisteredClaimNames.Sub, user.Username),
        new Claim(JwtRegisteredClaimNames.Email, user.Email),
        new Claim(ClaimTypes.Role, "Admin"),
        new Claim(ClaimTypes.Role, "Users"),
    };

    var token = new JwtSecurityToken(_config["Jwt:Issuer"],
      _config["Jwt:Issuer"],
      claims,
      expires: DateTime.Now.AddMinutes(20),
      signingCredentials: credentials);

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

OAuth avec web api

- AuthorizeAttribute
  - l'accès à un contrôleur ou à une méthode d'action est limité aux utilisateurs qui remplissent les conditions d'autorisation

```
[Authorize(Roles = "Administrators")]
1 reference | 0 changes | 0 authors, 0 changes
public class TodoController : ApiController
{
```