API ASP.NET Core

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

API REST avec ASP.Net Core

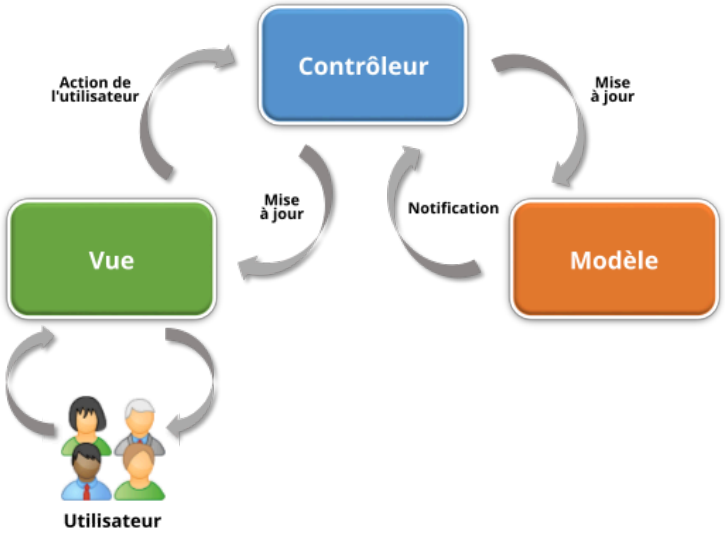
Ce support fait directement suite au support généraliste sur les APIS.

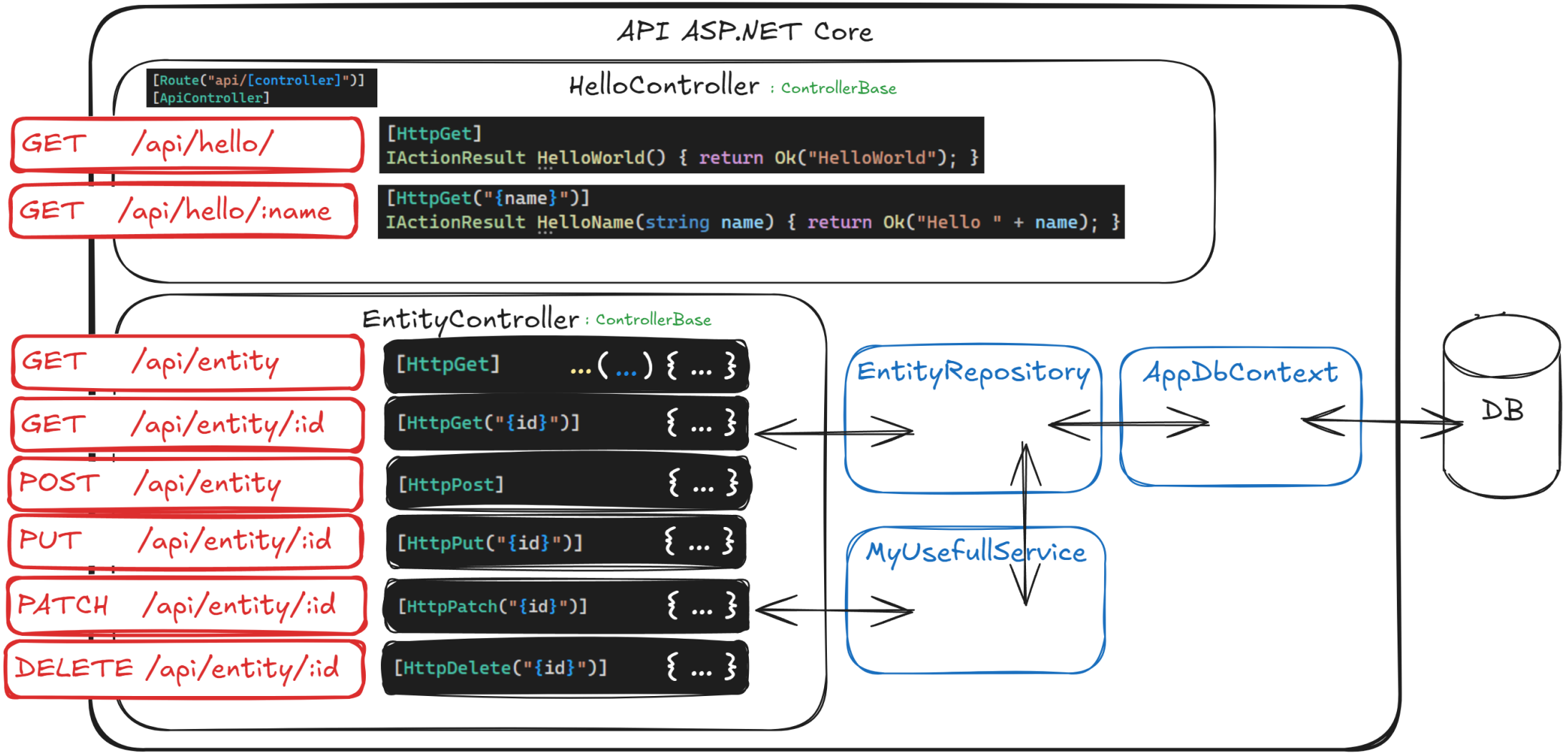
UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction à ASP.NET

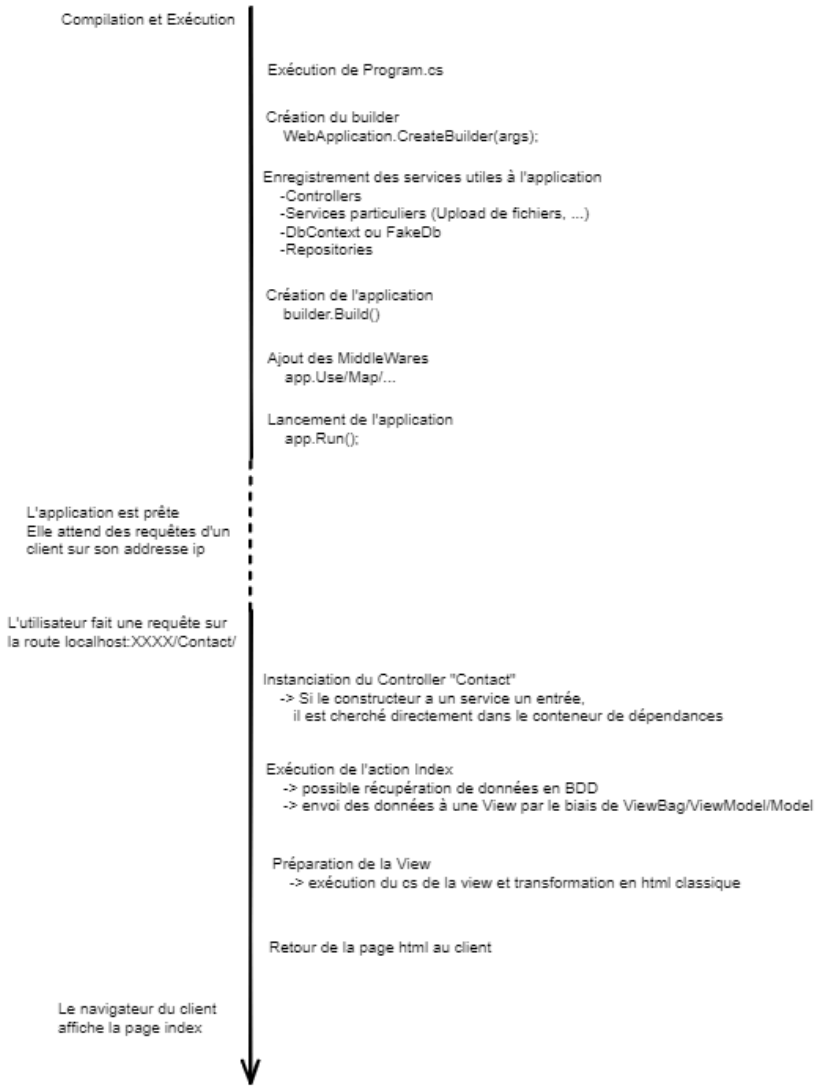
Architecture ASP.NET Core

- Composants principaux
  - `Program.cs` : point d'entrée
  - Pipeline des middlewares
  - Modèle MVC : Contrôleurs, Modèles, Vues




API ASP.NET Core
[Route("api/[controller]")]
[ApiController]
HelloController : ControllerBase
GET /api/hello/
[HttpGet]
IActionResult HelloWorld() { return Ok("HelloWorld"); }
GET /api/hello/:name
[HttpGet("{name}")]
IActionResult HelloName(string name) { return Ok("Hello " + name); }
EntityController : ControllerBase
GET /api/entity
[HttpGet] ...(...) { ... }
GET /api/entity/:id
[HttpGet("{id}")] { ... }
POST /api/entity
[HttpPost] { ... }
PUT /api/entity/:id
[HttpPut("{id}")] { ... }
PATCH /api/entity/:id
[HttpPatch("{id}")] { ... }
DELETE /api/entity/:id
[HttpDelete("{id}")] { ... }
EntityRepository
AppDbContext
DB
MyUsefullService
Endpoint == Action/Méthode d'un Contrôleur
Services (du conteneur de dépendances)

Etapes lors du fonctionnement d'une application ASP.NET CORE



Ignorer les information relative aux Views/html, cela s'applique aux application ***ASP.NET Core MVC***

Provenance des données dans les actions

Attribut	Provenance des données	Explication
`FromRoute`	Paramètres de l'URL	Les données sont extraites de la route ex: `/api/{id}`
`FromQuery`	Paramètres de la query string	Les données proviennent de la query string ex: `/api?id=123`
`FromBody`	Corps de la requête	Les données proviennent du corps de la requête (JSON ou XML **désérialisés**)
`FromForm`	Données d'un formulaire multipart	Les données sont envoyées via un formulaire HTML (en `multipart/form-data`)
`FromHeader`	En-têtes HTTP	Les données proviennent des en-têtes HTTP de la requête
`FromServices`	Service injecté	Injection de dépendances directement dans l'action

Exemples d'utilisation

- **FromRoute** : pour capturer des segments de l'URL

```
[HttpGet("/blabla/{id}/test")] // http://localhost/blabla/32/test
public IActionResult GetById([FromRoute] int id) { ... }
```

- **FromQuery** : pour capturer des paramètres passés en query string.

```
[HttpGet("/blabla")] // http://localhost/blabla?term=ABCD
public IActionResult Search([FromQuery] string term) { ... }
```

- **FromBody** : pour recevoir un objet depuis le body de la requête.

```
[HttpPost]
public IActionResult Create([FromBody] Product product) { ... }
```

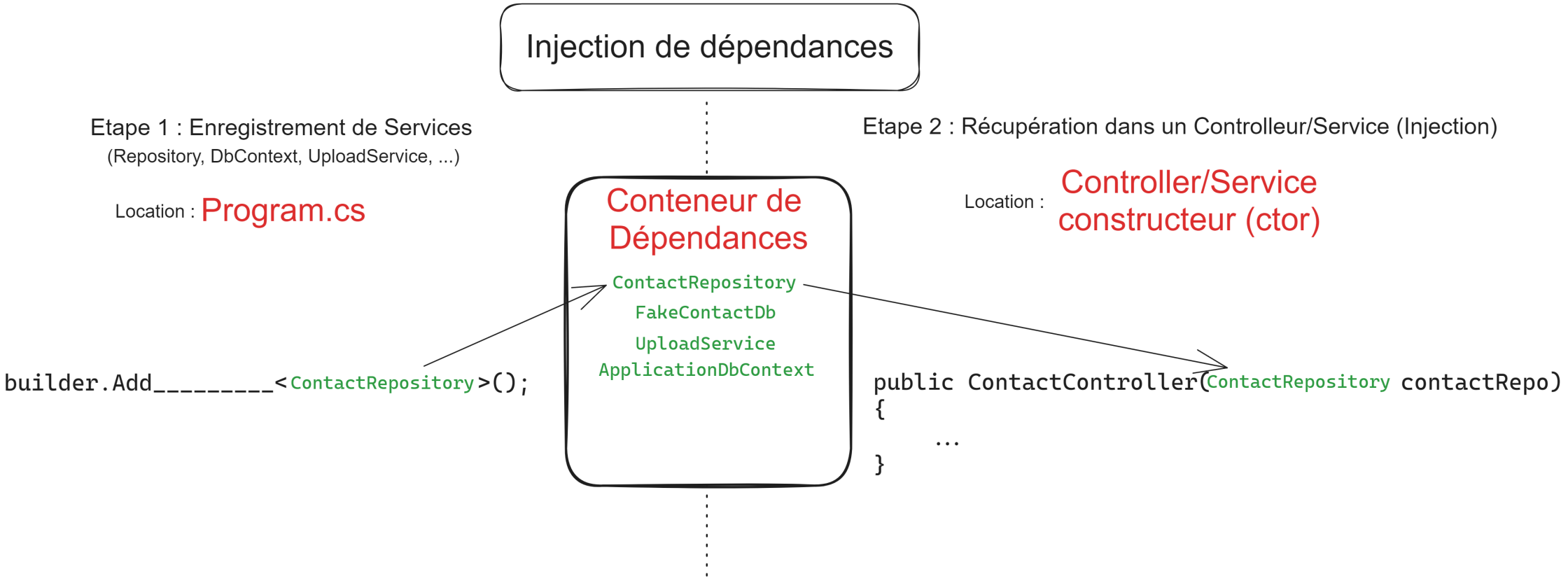
Swagger et OpenAPI

- Intégration de **SwaggerGen** via `Swashbuckle.AspNetCore` pour la **génération automatique de la documentation**
- Accès à l'interface **SwaggerUI** pour tester l'API via `/swagger`.
- Le fichier `swagger.json` est consultable depuis le **lien dans SwaggerUI**, il est à la syntaxe OpenAPI JSON.
- Il est le résultat du travail combiné de L'`APIEndpointExplorer` et du `SwaggerGen`
- C'est sur lui qu'est basé la SwaggerUI.

Injection de Dépendances

- Les services sont les **"briques algorithmiques"** qui serviront à **construire l'application** et qui seront **réutilisés par d'autres services** (/!\ Les controllers sont aussi des services)
- **Conteneur de dépendances**
  - **Enregistrement** des services dans `Program.cs`
  - **Lifetime des Services** : Scoped, Transient, Singleton
- Utilisation dans les contrôleurs
  - **Constructor Injection** et attributs privés

Injection de Dépendances

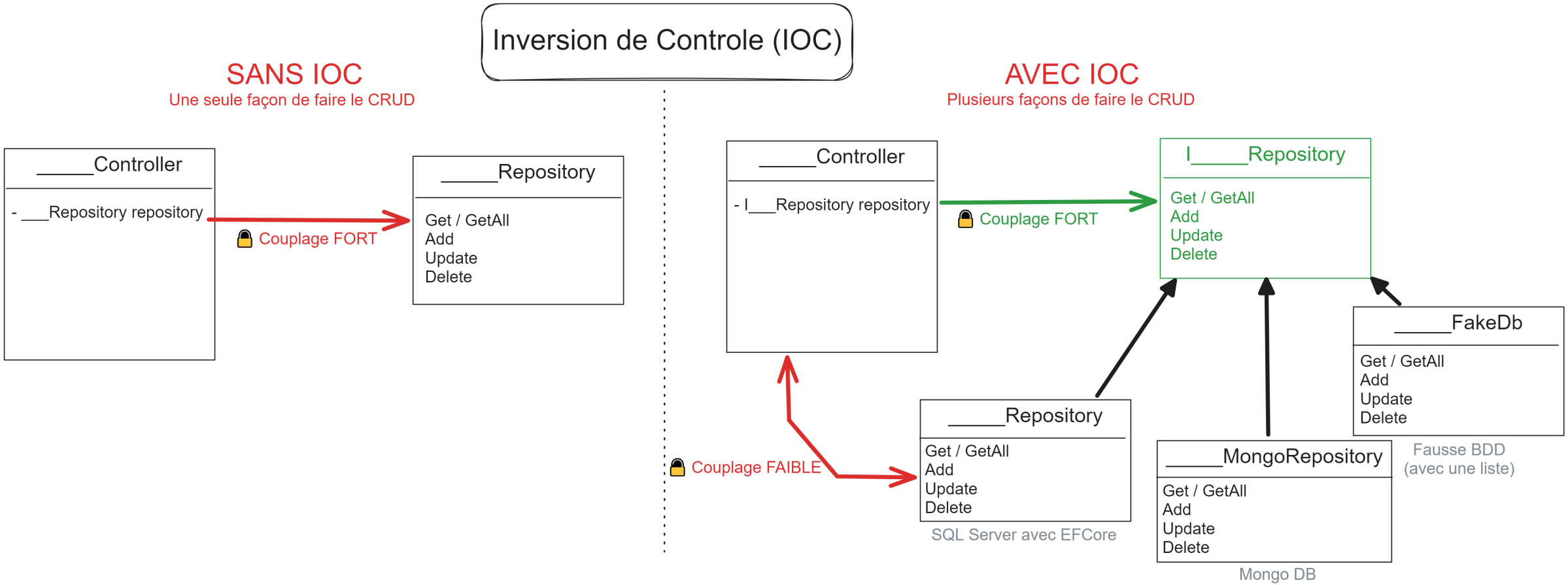


`AddSingleton` ou `AddScoped` ou `AddTransient`

Inversion de Contrôle

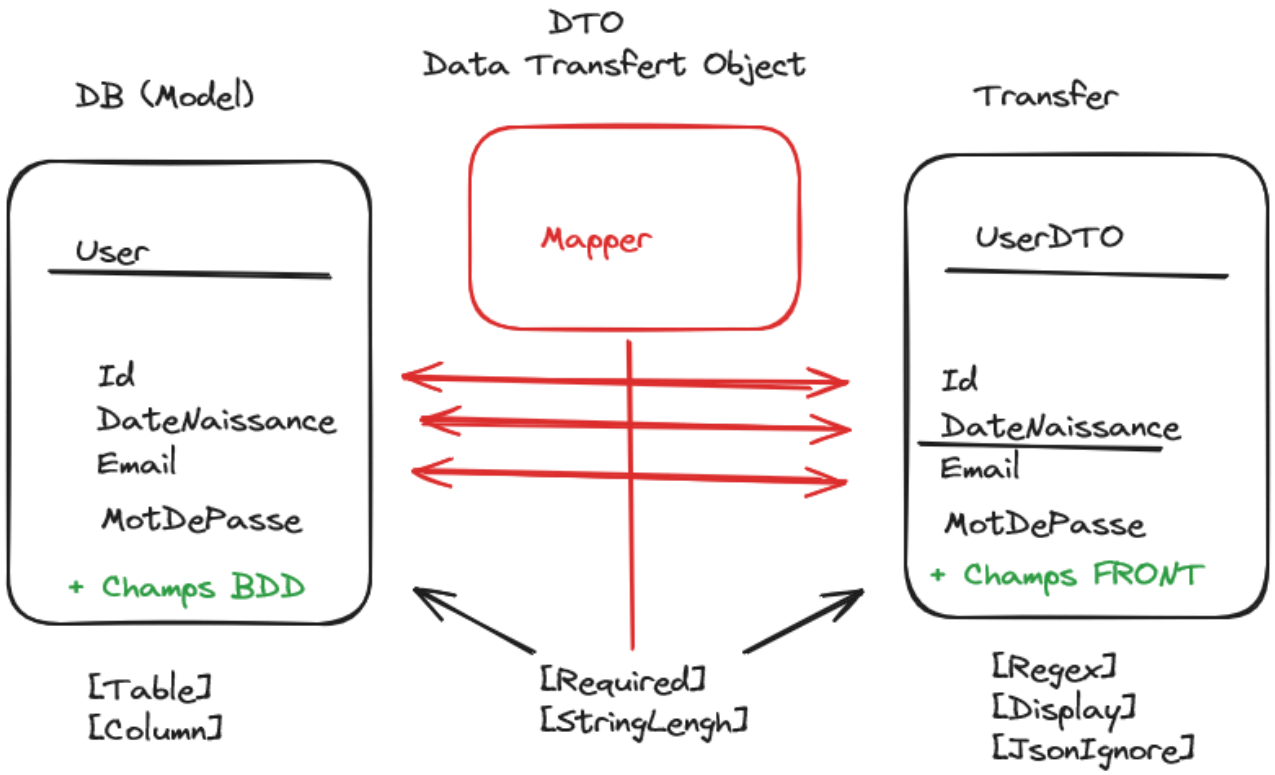
- Couplage **fort** vs couplage **faible**
- **Dépendre d'abstractions plutôt que d'implémentations**
  (cf. principes SOLID)
- **Ce que je veux faire** VS **Comment je fait en réalité**
- Exemple le plus courant :
  - Je veux réaliser un CRUD => interface `IEntityRepository`
  - Je le fais avec MongoDB => service `EntityMongoRepository`
    ou avec EF Core et MySQL => service `EntityEFCoreRepository`
    ou avec une List (FakeDB) => service `EntityFakeDb`

Inversion de Contrôle

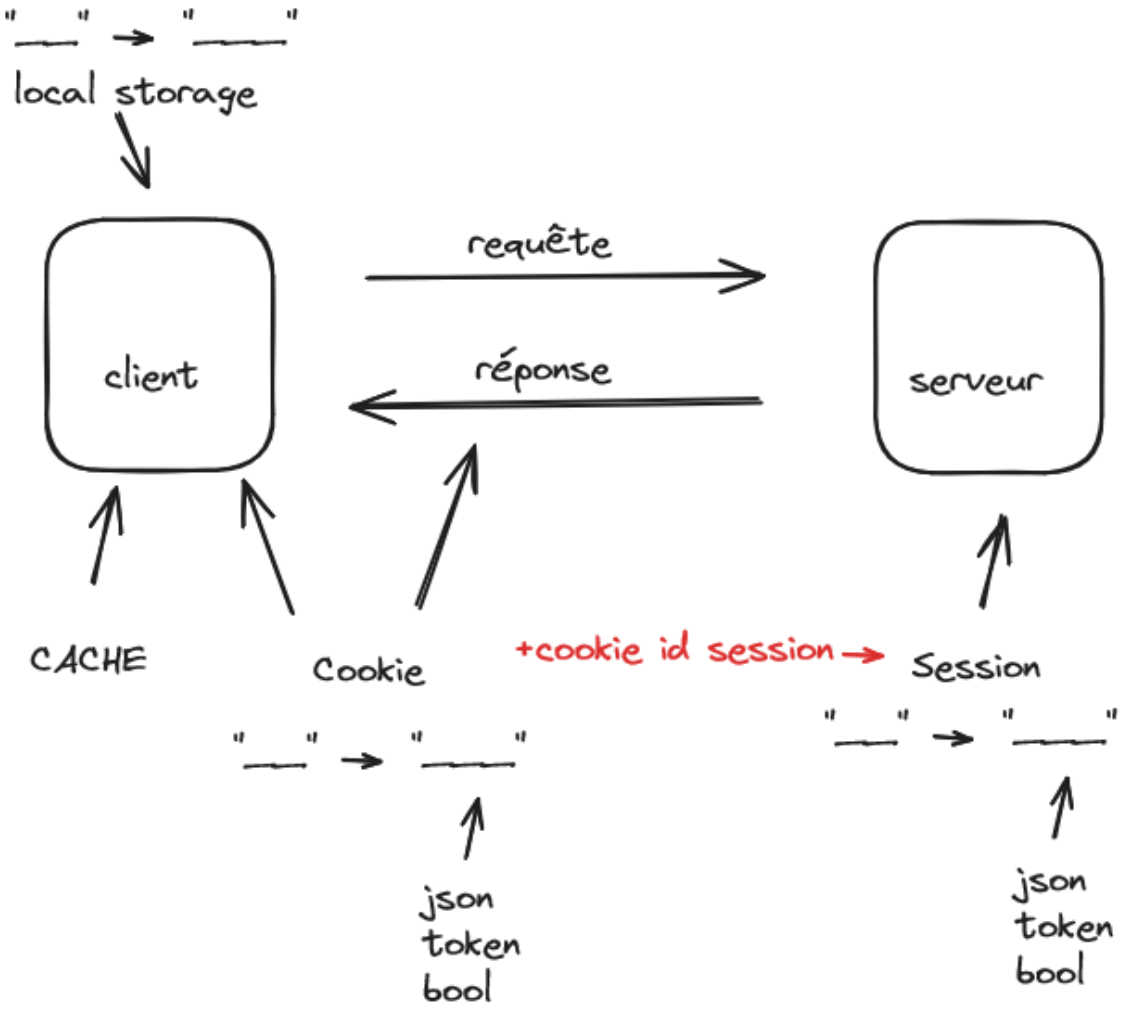


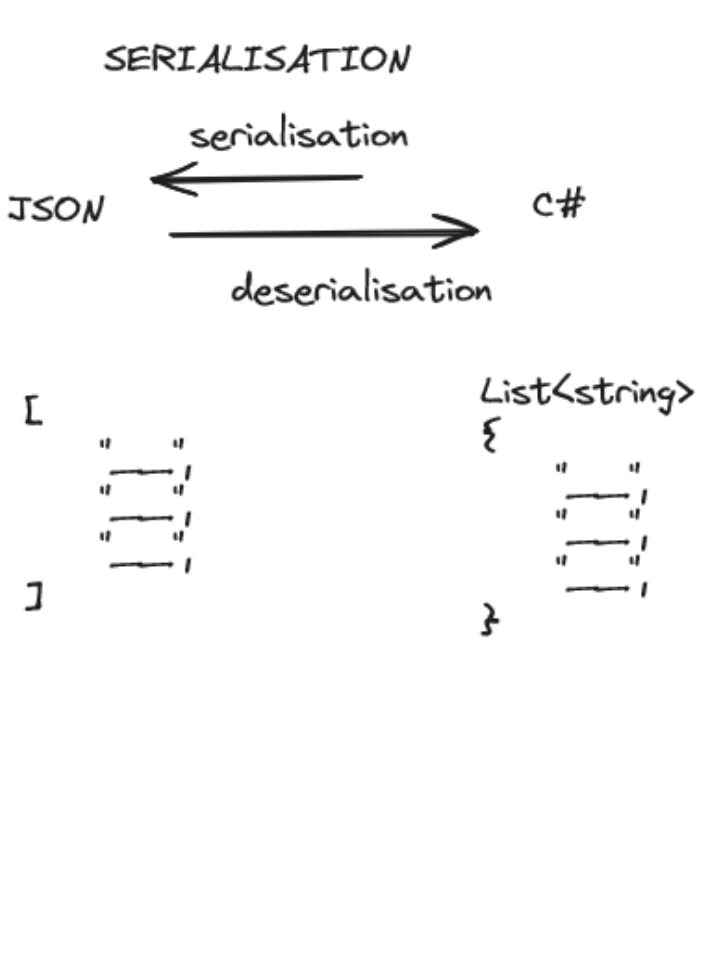
DTO Pattern

- Séparer les données de transfert des données stockées via 2 classes différentes et un Mapper pour la correspondance



Stockages du Web





Tests automatisés

- Types de tests : Tests **unitaires**, tests d'**intégration**
- **Mocking des dépendances** (Services) simplifié par l'inversion de contrôle
- Utilisation de bibliothèques comme **MSTest/NUnit/xUnit** et **Moq**
- Sujet connexe à connaître : TDD, BDD

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Sécuriser son API ASP.NET Core

CORS (Cross-Origin Resource Sharing)

- **CORS** : Mécanisme de **sécurité du navigateur** qui **bloque les requêtes HTTP** provenant d'un **domaine différent du serveur d'origine**.
- Exemple : Une application front-end sur `https://frontend.com` veut appeler une API située sur `https://api.com`.
- Par défaut, les navigateurs bloquent ces requêtes pour des raisons de sécurité (politique de **same-origin**).
- **CORS** permet d'autoriser explicitement ces requêtes cross-origin.

Configurer CORS dans ASP.NET Core

- **Activer CORS** dans `Program.cs` avec une policy (réglementation) :

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

// Configuration des règles CORS
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowAllOrigins",
        policy => policy.AllowAnyOrigin() // ou .WithOrigins("https://frontend.com") par exemple
                        .AllowAnyMethod()
                        .AllowAnyHeader());
});

var app = builder.Build();
app.UseCors("AllowAllOrigins"); // Utiliser la politique définie
// ...
```

Autoriser toutes les requêtes CORS sans politique

- La manière la plus simple d'autoriser toutes les requêtes CORS est d'utiliser le middleware par défaut, sans configurer de politique.

```
// ...
var app = builder.Build();

// Autoriser toutes les requêtes CORS (toutes origines, méthodes et en-têtes)
app.UseCors(cors => cors.AllowAnyOrigin()
                        .AllowAnyMethod()
                        .AllowAnyHeader());
// ...
```

Sécurité des Web Services REST

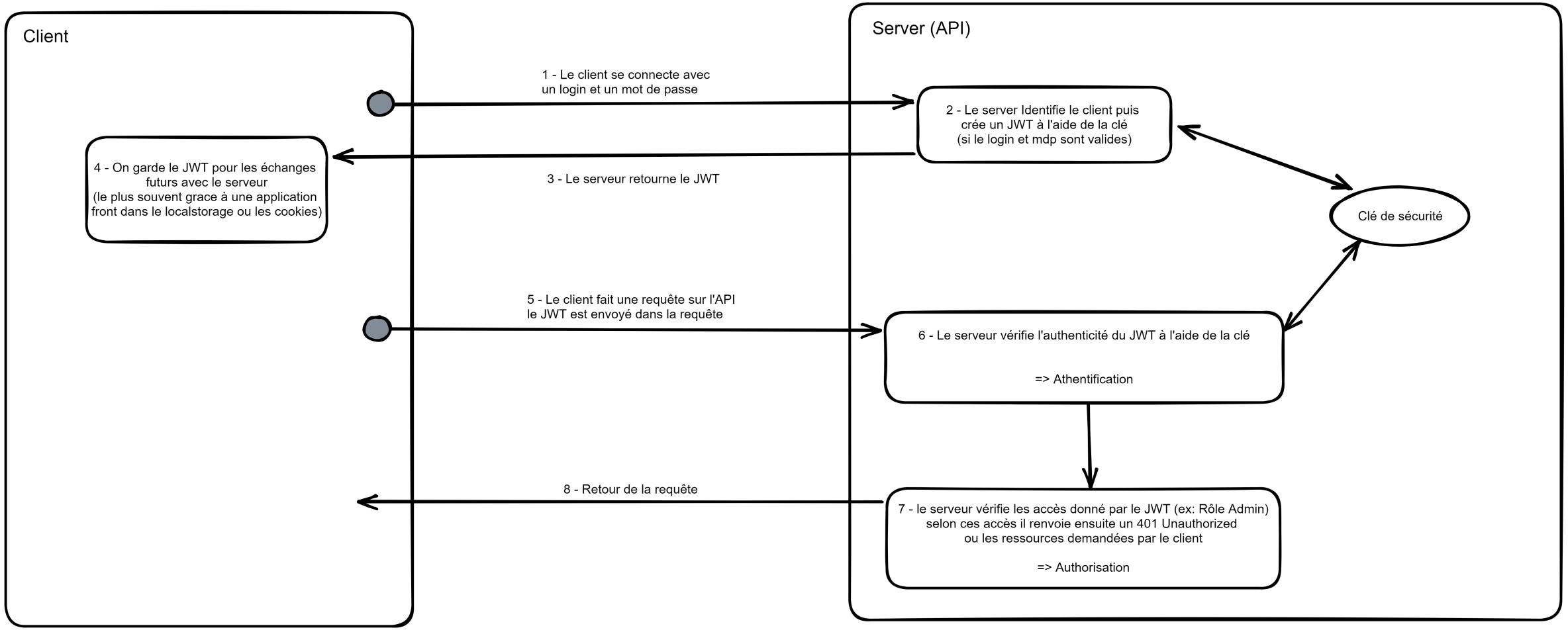
- 3 alternatives populaires :
  - **JSON Web Tokens** (JWT)
  - OAuth 2.0
  - OpenID Connect

Les 3 phases de la connexion sécurisée:

1. **Identification** : On **identifie une entité** et on lui donne de quoi affirmer cette identité en retour (token/credentials/clé).
2. **Authentification** : On **vérifie l'authenticité d'un message**/une action envoyé (peut contenir token/credentials/clé à vérifier).
3. **Autorisation** : On **définit les accès à une ressources** en fonction des règles définis dans les **A.C.L.** (Access Control Lists). Ces accès sont donnés en fonction des information Authentiques.

- Les règles A.C.L. ne sont **pas forcément associées à une entité**.
  Le porteur d'un "token" n'est pas forcément identifié (anonyme).

Fonctionnement JWT



C'est quoi un JWT ?

Un JWT est **toujours lisible par tout le monde** (**lecture seule**), cependant on ne peut l'**écrire** que si on possède la **clé de sécurité**.
Il est donc **Infalsifiable** sans la clé de sécurité.

1 Token = 3 parties :

- Headers : type de token et algorithme de chiffrement
- Payload (Body/Data) :
- Signature : ce qui permet de valider et déchiffrer le token

Exemple de JWT :

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjI0MyIsIm5hbWUiOiJHdWlsbGF1bWUgTWFpcmVzc2UiLCJlbnRlcnByaXNlIjoiVXRvcGlvcyIsIm1lc3NhZ2Ugc2VjcmV0IjoiTGUgQyMgZG9taW5lIHRvdXQgbGVzIGF1dHJlcyBsYW5nYWdlcyBjJ2VzdCDDqXZpZGVudCJ9.b0ZbcGLQT-uxWEDvI-vlPpq1qq3IzfR0k5kUt0_bR-E

Déchiffrer ce token via jwt.io

Comment travailler avec le JWT

- Une fois le token généré, il sera nécessaire de le **garder côté front-end** car il sert à affirmer **qui** envoie les requêtes.
- Le token JWT est généralement stocké dans le **localStorage** ou dans un **cookie** sécurisé.

Où stocker le JWT

- **localStorage** : Accessible en JavaScript et persistant entre les sessions du navigateur.
  - **Avantages** : Facile d'accès pour les requêtes.
  - **Inconvénients** : Vulnérable aux attaques XSS.
- **Cookies** (avec l'attribut `HttpOnly` et `Secure`) :
  - **Avantages** : Sécurisé et peut être envoyé automatiquement avec chaque requête HTTP.
  - **Inconvénients** : Vulnérable aux attaques CSRF si mal configuré.

Utilisation dans les requêtes

- Le JWT est généralement envoyé dans les **en-têtes HTTP** de chaque requête en utilisant l'en-tête **Authorization** :

```
Authorization: Bearer <token>
```

- Dans **fetch** ou **HttpClient** (Javascript):

```
fetch('https://api.example.com/data', {
  method: 'GET',
  headers: {
    'Authorization': 'Bearer ' + localStorage.getItem('token')
  }
});
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

WebSockets

Introduction à WebSocket

- **WebSocket** est un protocole qui permet une **communication bidirectionnelle en temps réel** entre le client et le serveur.
- Contrairement à HTTP, où la connexion est fermée après chaque requête/réponse, **WebSocket** maintient une connexion ouverte.
- Idéal pour des applications nécessitant des mises à jour instantanées (chat, jeux en ligne, notifications en temps réel, etc.).

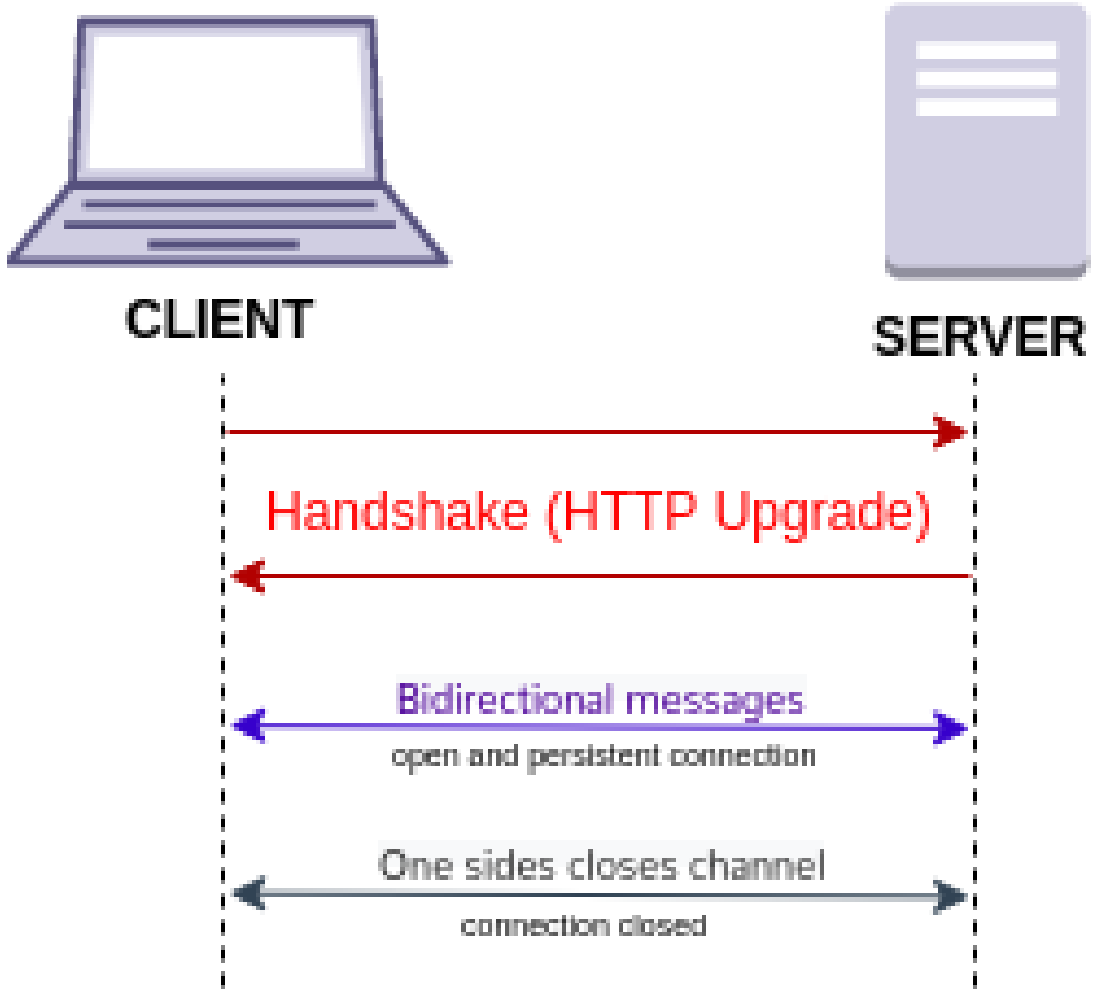
Pourquoi utiliser WebSocket ?

- **Temps réel** : Permet d'envoyer et de recevoir des données sans délai, sans avoir à ré-ouvrir la connexion.
- **Bidirectionnel** : Le client et le serveur peuvent s'envoyer des messages à tout moment.
- **Économique** : Moins de surcharge en comparaison à l'HTTP traditionnel (pas besoin de rouvrir la connexion à chaque requête).

Fonctionnement de WebSocket

- La connexion **WebSocket** démarre avec une requête HTTP traditionnelle (handshake).
- Une fois l'handshake réussi, la connexion passe en mode **full-duplex**.
- Communication en temps réel :
  - Le serveur peut envoyer des données au client **sans que le client fasse une requête**.
  - Le client peut envoyer des données au serveur **à tout moment** lui aussi.

Fonctionnement de WebSocket



Étapes principales

1. Le client envoie une requête HTTP avec l'en-tête `Upgrade: websocket`.
2. Le serveur accepte la connexion et l’upgrade en WebSocket.
3. Les messages sont échangés via un canal ouvert tant que la connexion reste active.

Démo: Configurer WebSocket dans `Program.cs` :

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

app.UseWebSockets();

app.Use(async (context, next) =>
{
    if (context.Request.Path == "/ws")
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            var webSocket = await context.WebSockets.AcceptWebSocketAsync();
            await HandleWebSocketCommunication(webSocket);
        }
        else
            context.Response.StatusCode = 400;
    }
    else
        await next();
});

app.Run();
```

Démo : Gérer la communication WebSocket

```
private async Task HandleWebSocketCommunication(WebSocket webSocket)
{
    var buffer = new byte[1024 * 4];
    WebSocketReceiveResult result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);

    while (!result.CloseStatus.HasValue)
    {
        var serverMessage = Encoding.UTF8.GetBytes("Message reçu");
        await webSocket.SendAsync(new ArraySegment<byte>(serverMessage, 0, serverMessage.Length),
                                  result.MessageType, result.EndOfMessage, CancellationToken.None);

        result = await webSocket.ReceiveAsync(new ArraySegment<byte>(buffer), CancellationToken.None);
    }

    await webSocket.CloseAsync(result.CloseStatus.Value, result.CloseStatusDescription, CancellationToken.None);
}
```

Démo : Client WebSocket avec JavaScript

```
const socket = new WebSocket('ws://localhost:5000/ws');

socket.onopen = function() {
    console.log("Connexion établie");
    socket.send("Hello Server");
};

socket.onmessage = function(event) {
    console.log("Message reçu du serveur :", event.data);
};

socket.onclose = function() {
    console.log("Connexion fermée");
};
```

Démo : Lancer la connexion

- Ouvrir la console du navigateur, observer l'établissement de la connexion et l'envoi/réception des messages en temps réel.

Cas d'usage de WebSocket

- **Chats en temps réel** : Permet de maintenir une conversation fluide sans requêtes répétées.
- **Notifications push** : Envoie de mises à jour instantanées (météo, finance, etc.).
- **Jeux en ligne** : Assure des interactions rapides et sans latence.
- **Collaborations en direct** : Synchronisation de documents ou d'actions en temps réel (ex : Google Docs).

Conclusion

- **WebSocket** est une solution puissante pour les applications en **temps réel** nécessitant des mises à jour constantes.
- Très utile pour les scénarios où les mises à jour du serveur doivent être instantanément communiquées au client.
- Attention à bien gérer la **sécurité** et les **ressources** du serveur pour des connexions WebSocket persistantes.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Déploiement

Déploiement d'une API ASP.NET Core

- Une API ASP.NET Core peut être déployée de différentes façons, en fonction des besoins de l'application et de l'infrastructure disponible.
- Voici quelques alternatives courantes pour le déploiement :
  1. **Internet Information Services (IIS)**
  2. **Auto-hébergement (Kestrel)**
  3. **Visual Studio & Azure**
  4. **Docker**

Déploiement avec IIS

- **IIS (Internet Information Services)** est un serveur web de Microsoft, largement utilisé pour héberger des applications ASP.NET.

  Avantages :

  - Intégration native avec l'écosystème Windows.
  - Support de la gestion des certificats SSL.
  - Possibilité de redémarrage automatique en cas de défaillance.

Déploiement avec IIS

Étapes :

1. Publier l'API depuis Visual Studio.
2. Configurer un site sur IIS pour l'application.
3. Utiliser **ASP.NET Core Module** pour relier Kestrel (serveur web interne de .NET) à IIS.

- Convient aux environnements **Windows** avec une infrastructure d'entreprise.

Déploiement Auto-hébergé (Kestrel)

- **Kestrel** est le serveur web léger intégré à ASP.NET Core, idéal pour les scénarios d'auto-hébergement.

  Avantages :

  - Pas besoin de serveur web externe comme IIS.
  - Haute performance pour les applications simples ou en microservices.
  - Multi-plateforme : fonctionne sous **Windows**, **Linux** et **macOS**.

Déploiement Auto-hébergé (Kestrel)

Étapes :

1. Publier l'API en mode **auto-hébergé**.
2. Lancer l'application directement en exécutant le fichier compilé (ex. `dotnet YourApp.dll`).
3. Exposer Kestrel directement ou placer un serveur proxy (ex : Nginx, Apache) devant Kestrel pour gérer les requêtes HTTP.

- Convient aux petites applications ou architectures **microservices**.

Déploiement avec Visual Studio et Azure

- **Visual Studio** permet un déploiement direct sur **Microsoft Azure** (Cloud provider) via **App Service** par exemple.

  Avantages :

  - Déploiement **facile** directement depuis l'IDE avec l'option "Publier".
  - Intégration avec les **services cloud** d'Azure : **App Service**, **Azure Functions**, **Azure DevOps**.
  - Gestion automatique du **scalabilité** et du **monitoring**.

Déploiement avec Visual Studio et Azure

Étapes :

1. Choisir **Azure App Service** comme cible de déploiement.
2. Configurer les paramètres de l'environnement (nom, plan de service, etc.).
3. Cliquer sur "Publier" pour déployer l'application.

- Idéal pour des applications nécessitant une montée en charge automatique ou l'usage de services cloud.

Déploiement avec Docker

- **Docker** permet de créer des **conteneurs** pour isoler et déployer des applications. On parle alors de **virtualisation** et **conteneurisation**.

  Avantages :

  - **Portabilité** : Fonctionne de la même manière sur n'importe quel environnement (local, cloud, serveur).
  - **Isolation** : Chaque conteneur a ses propres dépendances, ce qui évite les conflits entre environnements.
  - Parfait pour les architectures **microservices**.

Déploiement avec Docker

Étapes :

1. Créer un **Dockerfile** pour l'API ASP.NET Core.
2. **Construire l'image** avec `docker build`.
3. Lancer un conteneur avec `docker run`.
4. Pousser l'image sur un **registre de conteneurs** (ex. Docker Hub ou Azure Container Registry) pour déploiement futur.

- Solution flexible pour des environnements **multi-plateformes**.

Comparaison rapide des options de déploiement

Méthode	Avantages	Inconvénients
IIS	Intégration Windows native, SSL	Dépendant de Windows
Auto-hébergé	Haute performance, léger	Nécessite un proxy pour HTTPS
Azure	Facilité, intégration cloud	Coût du cloud
Docker	Portabilité, isolation	Complexité de configuration

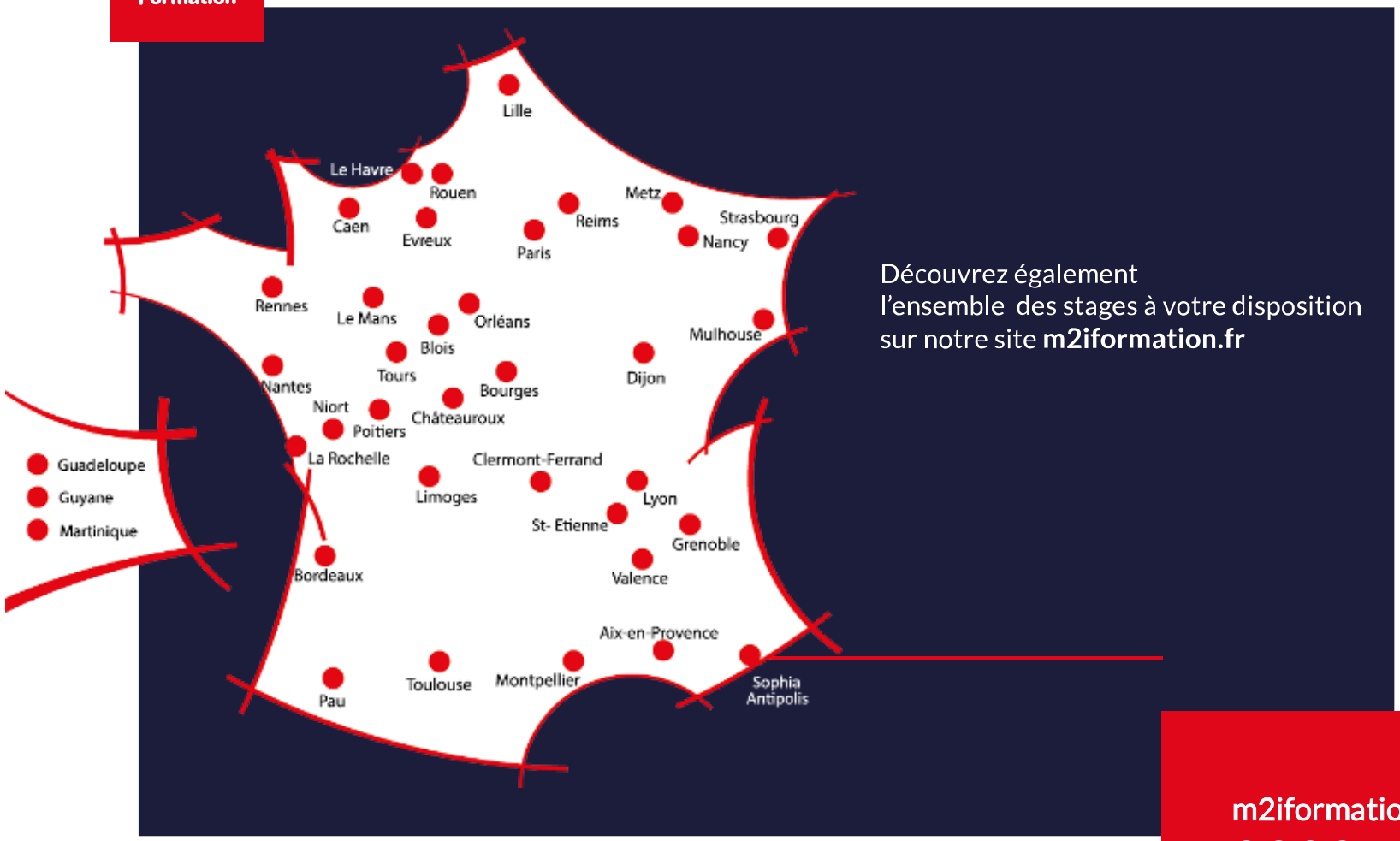
Conclusion

- Chaque option a ses avantages en fonction des besoins.
- **IIS** et **Azure** sont parfaits pour des environnements Windows ou cloud.
- **Kestrel** et **Docker** sont adaptés aux applications nécessitant de la flexibilité et de la portabilité, en particulier pour les microservices.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr